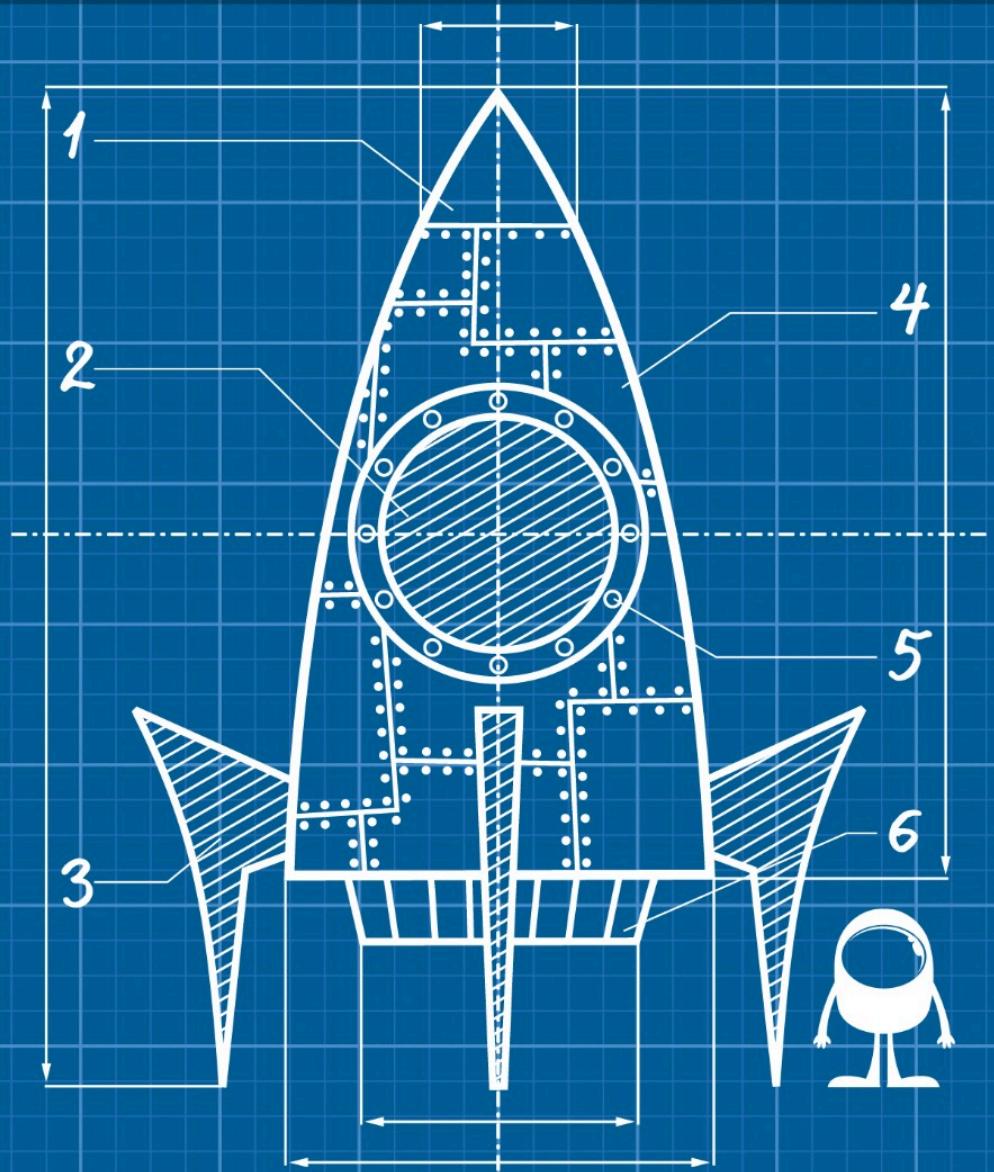


# Julia for Beginners

From Romans to Rockets



Erik Engheim

SixtyNORTH



# Preface

I began programming as a teenager with fun books containing comic book strips with wizards and turtles. I read magazines that showed me how to make my own simple games or make silly stuff happen on the screen. I had fun.

But when I went to university, books started talking about bank accounts, balances, sales departments, employees and employers. I wondered if my life as a programmer would mean putting on a gray suit and writing code handling payroll systems. Oh the horror!

At least half of my class hated programming with a passion. I could not blame them. Why did programming books have to be so boring, functional and sensible?

Where was the sense of adventure and fun? Fun is underrated. Who cares if a book is silly and has stupid jokes if it makes you learn and enjoy learning?

That is one of the reasons I wrote this book. I wanted the reader to enjoy learning programming. Not through cracking jokes, but by working through programming examples that are interesting or fun to do.

I promise you, there will be no examples modeling a sales department. Instead we will simulate *rocket launches*, pretend to be Caesar sending a secret message to his army commanders using old Roman *encryption techniques*, as well as simulating a beautiful old handheld mechanical calculator, the Curta, and many other things.

The second important reason I wanted to write this book is because people keep telling me: “Julia? Isn’t that a language only for science and scientists?”

Julia has had major success in this area, which is why the Julia community today is full of brainy people working on hard problems such as developing new drugs, modeling the spread of infectious diseases, climate change or the economy.

But no, you don’t need to be a genius or a scientist to use Julia. Julia is a wonderful *general purpose* programming language for everyone! I am not a scientist and I have enjoyed using it for over 7 years now. With Julia you will find that you can solve problems more quickly and elegantly than you have done in the past. And as a cherry on top, computationally intensive code will run blisteringly fast.



# Introduction

Software is everywhere around us. Every program you see on your computer, smart phone or tablet has been made by someone, who wrote code in a programming language.

But that isn't the only places you'll find software. You may think of a computer as box that sits on your desk or a laptop computer, but there are tiny computers we call micro controllers inside almost any kind of technical device we use. In cars for instance, these little computers figure out how much gasoline needs to be injected into the engine cylinder.

Have you ever seen the Falcon 9 rocket land at sea on a barge by firing its rocket engines right before it smashes into the deck? There are numerous computers inside this rocket keeping track of how fast the rocket is going, how far it is from the ground and exactly how much thrust it has to apply, and for how long, to avoid crashing.

All of these computers run programs that somebody wrote.

Programs are written in many different programming languages. People sometimes ask me what the best programming language is. There is no best language! It is like asking what the best car is. Some cars are better for certain things than others. Ferraris are good at going really fast, but not good at transporting your IKEA furniture. But most of the time, car choices come down to personal preferences and personality. Programming languages are the same. It is both about the job, but also about how you like to work.

## The Julia Programming Language

I have chosen to teach you to the Julia programming language. Why did I pick this language, when there are hundreds of others to chose from, some which are much better known?

1. It is a **fun language!** The language plays on your team instead of against you.
2. **Easy to learn.** Some languages languages require you do learn a myriad of details before you get to do anything at all. Julia lets you learn one small thing at a time.
3. **Powerful.** You can get a lot done with very small amounts of code.



Figure 1: A Falcon 9 booster coming in for landing.

4. **Batteries included.** Don't you hate it when you unpack a cool new thing and it doesn't work, because batteries are sold separately? A lot of programming languages are like that, but not Julia.
5. **Fast.** You can write code that runs slow in any language, but Julia gives you the *ability* to write very high performance code.



# Overview

The chapters in this book are meant to be read in sequence, and build upon each other. However not every single chapter needs to be read.

This book tries to balance the needs of three different kinds of readers:

1. **The beginner**, with minimal programming experience. For this reason the first chapters try to keep things simple and assume limited prior knowledge of common programming concepts.
2. **The experienced developer** who wants to move faster and have more in depth coverage of material specific to Julia. For this reason, later chapters will assume more prior knowledge of various programming concepts.
3. **The curious technology enthusiast**. Some chapters go into more technical and historical details, which some readers may find fascinating or interesting. However these chapters cover material which may not be *strictly necessary* to master Julia.

The inspiration for this organization is my own experience reading technology and science books for fun. Sometimes you want to learn something fast and get to the meat quickly. Other times you enjoy immersing yourself with geeky details.

1. Installing and Setting up Tools. We cover how to install Julia, setup tools to work with Julia and a basic introduction to the Julia programming environment.
2. Working with Numbers. What makes a computer different from a mere calculator? Exploring the ability to automate complex number calculations using a programming language.
3. Working with Text. We make a simple program to allow somebody to practice multiplication. This ties together reading input from the user and writing out messages.
4. Storing Data in Dictionaries. We work through an example of how to convert Roman numerals to decimal numbers by using an important data structure called a dictionary.
5. How Does a Computer Work? Is an introduction to the *binary number system*, binary operations and how to work with integers in different representations. Go through how a *microprocessor* does arithmetic.
6. More on Types. We explain the Julia type system and how it ties in with *multiple dispatch*.
7. Defining Your Own Types by building a space rocket in code.

8. Static vs Dynamic Typing. Julia has type annotations, so what makes Julia a dynamic language, and how is it different from a statically typed language such as C/C++, Java or C#?
9. Conversion and Promotion of different number types.
10. Different Kinds of Nothing. How to represent objects which are not found, missing or undefined in Julia.
11. Strings. Working with text strings. What is Unicode and UTF-8 encoding?
12. Object Collections. Shared abilities for different types of object collections and how you make your own.
13. Working with Sets. Creating sets, set operations and what makes a set different from an array.
14. Your Own Spreadsheet. Use Julia like a spreadsheet, or how to work with multi-dimensional arrays. A simple introduction to linear algebra.
15. Moving a Rocket using affine transformation matrices (2D arrays) to simulate movement and rotation of a space rocket. A geometric look at matrices, vectors and points.
16. Functional Programming concepts such as closures and higher order functions. A deeper look at using functions in Julia and how to think functional.
17. Object-Oriented Programming. How do you apply your object-oriented thinking to a language which isn't object-oriented? What does a design pattern look like in Julia?
18. Code Organization in files, modules and packages. Creating dependencies between packages.
19. Input and Output to files and network sockets. Asynchronous network socket communication. Representing objects as strings.
20. Shell Scripting in Julia. Instead of Bash, Julia can be used to do common scripting operations.
21. Parametric Types allow you to write safer and more high performance code.
22. Testing focuses on how to write unit tests in Julia.
23. Logging covers the standard logging framework in Julia. The purpose of logging, logging levels and making your own loggers.
24. Debugging is centered on how to use an interactive Debugger in Julia.

# Installing and Setting up Tools

- Installation
- Setup
- Text editor
- The Julia Read-Eval-Print loop
- Use Julia as a calculator

Before writing any programs, you've got to get used to the environment and the tools we are going to use. In this first chapter we will focus on the basics of writing and running Julia code. We will do that by starting Julia and using it as a calculator.

And we will do the equivalent of having some food samples. Just a tiny taste of later topics such as *variables*, *functions* and *types*.

Don't worry if you don't understand a topic at first. We will get into more details later in the book. In the beginning it is just about giving you an overview, not about understanding every detail.

## Install Julia

These are the steps to download and install Julia.

### Download

1. Go to the website [julialang.org/downloads](https://julialang.org/downloads).
2. Select the right Julia version for your operating system. We recommend Julia 1.4. However any 1.x version should work.

### Install on Mac

1. Open your downloaded .dmg file named something like `julia-1.4.2-mac64.dmg`.
2. Drag and drop the Julia application bundle to /Applications folder.

This completes the install. It is that easy. The next steps are for convenience.

1. Open the Terminal.app console applicaton.



Figure 2: The Julia programming language homepage

2. Create symbolic link from installation location to a directory in your path, e.g. /usr/local/bin.

```
$ ln -s /Applications/Julia-1.4.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia
```

## Install on Linux

1. Unpack the .tar.gz file named something like julia-1.4.2-linux-x86\_64.tar.gz.
2. Move unpacked directory to /opt. If /opt does not exist, create it.

```
$ sudo mkdir /opt
$ cd $HOME/Downloads
$ sudo mv julia-1.4.2 /opt
```

Next to be able to easily run Julia from the terminal we will make a symbolic link.

```
$ sudo rm /opt/bin/julia # remove any old link
$ sudo ln -s /opt/julia-1.4.2/bin/julia /opt/bin/julia
```

## Install on Windows

You will get an .exe file as the download, which is installed like any other Windows software by double clicking it.

## Configure on Mac and Linux

To make it easy to run Julia from the terminal, it is useful to configure your shell environment for Julia.

Here is an example of configuring Julia for use with bash shell on Linux, where Sublime (subl) is used as the text editor for Julia code.

```
export JULIA_EDITOR=subl
export PATH=/usr/opt/bin:$PATH
```

How this is configured depends on the shell you use. E.g. here is an example of another configuration using the fish shell, this time on Mac. Most Mac user will be using bash or Z shell.

```
set -x JULIA_EDITOR atom
set -x PATH /usr/local/bin $PATH
```

In this case we are using the Atom (atom) editor, and the link to the Julia executable is in /usr/local/bin.

## Configure on Windows

On Windows, environment variables are configured through the graphical user interface you see below.

The JULIA\_EDITOR environment variable is not necessary to set on Windows as the operating system will open a dialog and ask you what editor to use when needed. Windows will then associate an application with .jl files.

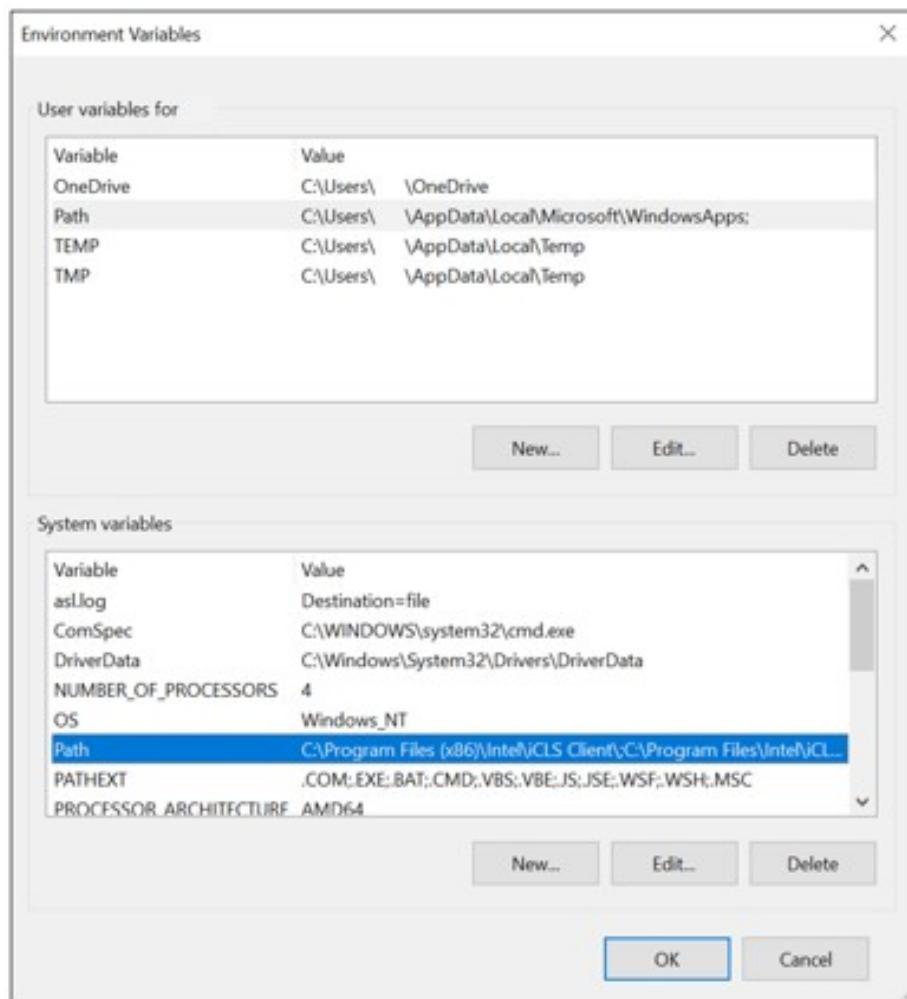


Figure 3: Windows 10 environment variables dialog

Configuring the shell environment is likely less needed on Windows than on Linux/macOS because command line interfaces are not as frequently used by Windows developers. Developers interested in the command line on Windows may prefer to use the Window Subsystem for Linux (WSL).

## Run

1. Start Julia
2. Write some code to check that it works

```
$ julia
julia> println("hello world")
hello world
```

**NOTE How do you exit Julia?**

You can interrupt anything you are doing in Julia by holding down the **Ctrl** key and pressing **C**. We write this as **Ctrl+C**. To exit Julia you hold down **Ctrl+D**.

## Install a Julia Code Editor

For little snippets of code it is fine using the Julia REPL environment. However as soon as we need to write more than five lines of code, it is nicer to use a proper text editor.

You cannot use any kind of text editor for this. MS Word or Apple's Pages would not work. That is because these programs store all sorts of information about the text you write, such has colors, fonts, margins etc.

Julia does not understand anything but plain text. So you need an editor which can edit UTF-8 or ASCII encoded text. It is best to pick one that has plugin support for Julia. You don't need to, but it makes it more convenient and nicer to work with.

A proper code editor with Julia support will colorize<sup>1</sup> your code, so you can quickly see important parts of your code. Some editors have a lot more features. All the editors I list below has plugins for working with Julia.

## Multi Platform

- Atom
- VS Code
- Sublime 3
- Vim
- Jupyter
- Pycharm

---

<sup>1</sup>Syntax highlighting is a popular feature in many programming editors which will colorize code based on its syntax.

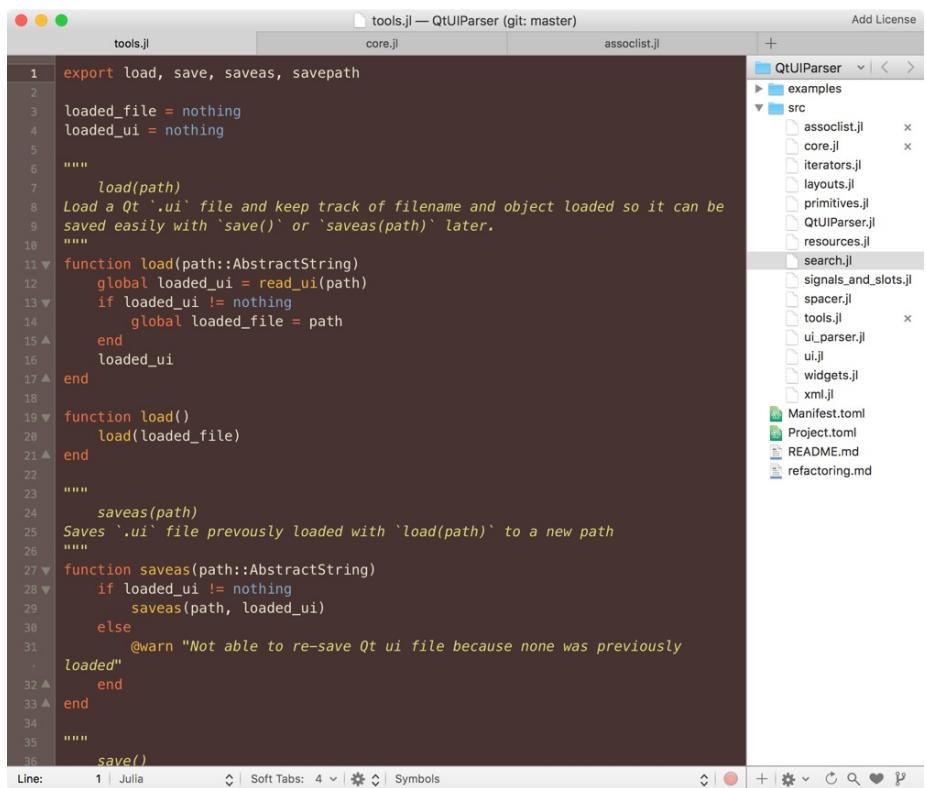


Figure 4: TextMate code editor configured with Julia plugin

The screenshot shows the Juno (atom) IDE interface. On the left, there's a 'Project' sidebar with a 'src' folder containing files like basiclexer.jl, lexer.jl, parser.jl, etc. The main area is a code editor with tabs for 'parser.jl' and 'basiclexer.jl'. The 'basiclexer.jl' tab is active, displaying Julia code for lexing numbers, strings, and identifiers. The code uses functions like `lex\_number`, `lex\_hexbinary`, and `lex\_identifier`. At the bottom, there's a status bar showing 'Slime not connected.' and various file and plugin icons.

```

3 "Lex a number. Could be hexadecimal, scientific or start with +"
4 function lex_number(l::Lexer)
5     scan_number(l)
6     emit_token(l, NUMBER)
7     lex_basic
8 end
9
10 function lex_hexbinary(l::Lexer)
11     accept_char(l, "<")
12     accept_char_run(l) do ch
13         isxdigit(ch) || isspace(ch)
14     end
15     if accept_char(l, ">")
16         emit_token(l, HEXBINARY, filter(isxdigit, lexeme(l)))
17     else
18         return error(l, "Hexadecimal data must end with >")
19     end
20     return lex_basic
21 end
22
23 "Lex a string enclosed in \" quotes"
24 function lex_string(l::Lexer)
25     if scan_string(l)
26         emit_token(l, STRING)
27         lex_basic
28     else
29         error(l, "Not a valid quoted string")
30     end
31 end
32
33
34 "lex an identifier such as a variable or function name"
35 function lex_identifier(l::Lexer)
36     ...
37 end

```

Figure 5: Juno (atom) IDE configured with Julia plugin

## Mac

- TextMate
- BBedit
- TextWrangler

## Windows

- Nodepad++

## Using Julia as a Calculator

Startup Julia by either clicking the icon of Julia or by writing `julia` on the command line.

I never use a calculator or calculator application on my computer. Instead I keep a terminal window with Julia open at all times. Even if you do not care to learn how to program, Julia works great as a calculator.

Below you can see how there are many ways to write numbers. There are even symbols for numbers with many digits such as  $\pi$ , so you don't have to remember all the digits. Actually  $\pi$  has an infinite number of digits.

```
julia> 32
32
```

```
julia> 45
45

julia> 1.2e3
1200.0

julia> pi
π = 3.1415926535897...
julia> π
π = 3.1415926535897...
```

However not all letters are placeholders for numbers. Say you write  $\theta$ , Julia will respond that nobody has defined which number  $\theta$  should be yet.

```
julia> θ
ERROR: UndefVarError: θ not defined

julia> 3.4
3.4

julia> -23
-23
```

All common mathematical operations such as addition, subtraction and multiplication that you have learned in school are possible to do in Julia.

```
julia> 3 + 4
7

julia> 3.4 + 5
8.4

julia> 2^2
4

julia> 2^3
8

julia> 3*3 + 1
10
```

You can even do trigonometric functions such as sine and cosine.

```
julia> sin(θ)
0.0

julia> cos(θ)
1.0
```

# Working with Numbers

- **Variables.** Getting Julia to remember long numbers and strings of text for you.
- **Functions.** Store how a calculation is done for later reuse.
- **Control flow.** Using loops and if-statements to decide what code to run and how many times to do it.
- **Types.** The kind of objects we can work with in Julia.

What is programming and why is it useful? To explain that I am going to tell you a story, about what one of the first programs in history were made to do. Perhaps you have wondered what those first computers did, back when computers had no screens, but instead had large walls of blinking lights?

Our story starts in World War II. Mathematics started to become very important to winning the war. The Germans sent secret messages through radio telegraphs. These messages were sent by the infamous Enigma machines which encrypted and decrypted secret messages. Figuring out what the secret message said was like solving a mathematical puzzle.

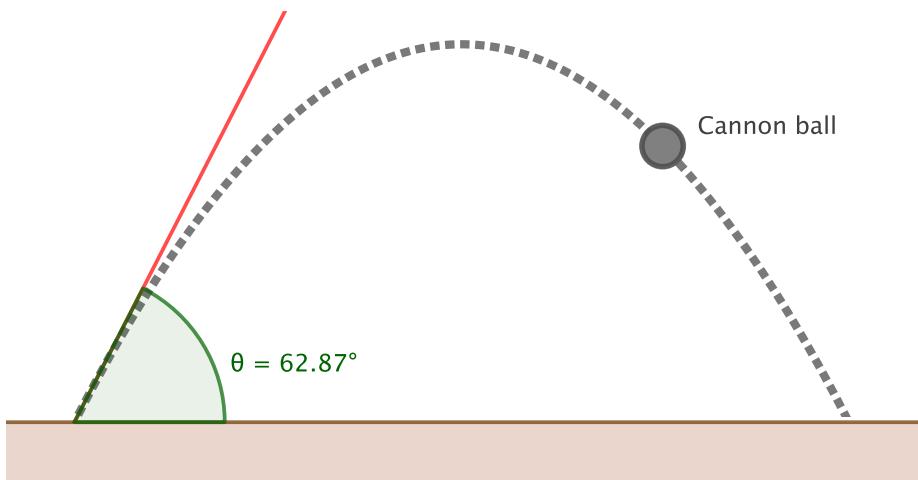


Figure 6: Distance traveled by a cannon ball depends on the elevation of the cannon

Another problem was to figure out the correct angle to orient the artillery cannons to hit their targets. Say you want to hit an enemy bunker 8 km away. At

what angle  $\theta$  should you fire your cannon?

This is a mathematical problem, and the poor grunts fighting the war could not bust out a pen and paper and start doing complicated math calculations each time they wanted to fire a cannon. How do you think they did it?

## Calculating Artillery Trajectories

They used books with lots of tables. Before handheld calculators, mathematicians used scientific tables. There was tables for everything: trigonometric functions (sine, cosine), logarithms, normal distributions, etc.

TABLE OF FIRE. LIGHT 12-POUNDER GUN. MODEL 1857.							
SHOT.		SPHERICAL CASE SHOT.			SHELL		
Charge $2\frac{1}{4}$ Pounds.		Charge $2\frac{1}{4}$ Pounds.			Charge 2 Pounds.		
ELEVATION In Degrees.	RANGE In Yards.	ELEVATION In Degrees.	TIME OF FLIGHT. Seconds.	RANGE In Yards.	ELEVATION In Degrees.	TIME OF FLIGHT In Seconds.	RANGE In Yards.
0°	323	0°50'	1"	300	0°	0°75	300
1°	620	1°	1°75	575	0°30	1°25	425
2°	875	1°30'	2°5	635	1°	1°75	615
3°	1200	2°	3"	730	1°30'	2°25	700
4°	1325	3°	4"	960	2°	2°75	785
5°	1680	3°30'	4°75	1080	2°30'	3°5	925
		3°40'	5"	1135	3°	4"	1080
					3°45'	5"	1300

Use SHOT at masses of troops, and to batter, from 600 up to 2,000 yards. Use SHELL for firing buildings, at troops posted in woods, in pursuit, and to produce a moral rather than a physical effect; greatest effective range 1,500 yards. Use SPHERICAL CASE SHOT at masses of troops, at not less than 500 yards; generally up to 1,500 yards. CANISTER is not effective at 600 yards; it should not be used beyond 500 yards, and but very seldom and over the most favorable ground at that distance; at short ranges, (less than 200 yards,) in emergency, use double canister, with single charge. Do not employ RICCHET at less distance than 1,000 to 1,100 yards.

**CARE OF AMMUNITION CHEST.**

1st. Keep everything out that does not belong in them, except a bunch of cord or wire for breakage: beware of loose tacks, nails, bolts, or scraps.  
2d. Keep friction primers in their papers, tied up. The pouch containing those for instant service must be closed, and so placed as to be secure.  
Take every precaution that primers do not get loose: a single one may cause an explosion. Use plenty of low in packing.  
(This sheet is to be glued on to the inside of Limber Chest Cover.)

Figure 7: Example of table used to decide correct elevation of a cannot to fire a given range.

The soldier manning the artillery cannons had books with numerous tables. The tables would tell them what elevation (angle) to put the cannon in order to fire the artillery projectile (cannon ball) the desired distance. But these tables could get very complicated, because so many things affect how far the cannon ball would go:

- wind
- amount of gunpowder
- the particular kind of cannon (artillery) used.

This meant they needed countless tables, which is why during WWII the allies had huge rooms filled with people calculating these tables. People doing these calculations were called computers. That was what a computer was before electronic computers. It was a person doing lots of calculations.

The first computers were made to replace thousands of human computer, so that all these tables could be quickly calculated by a machine. Let us look at how these calculations where done.

## Angle of reach

Say you got a cannon and you want to shot an enemy. Your enemy is at a distance  $d$  from you. The cannon ball you fire, exits the cannon with a velocity  $v$ . What angle  $\theta$  does your cannon need to be elevated?

That angle is called the “angle of reach” and is calculated as follows:

$$\theta = \frac{1}{2} \arcsin \left( \frac{gd}{v^2} \right)$$

In this equation  $g$ , is the acceleration gravity gives a falling object. On earth that is  $9.81 \text{ m/s}^2$ . That means if you fired your cannon on Mars or the moon the result would be different because object don't fall as fast there, due to lower gravity.

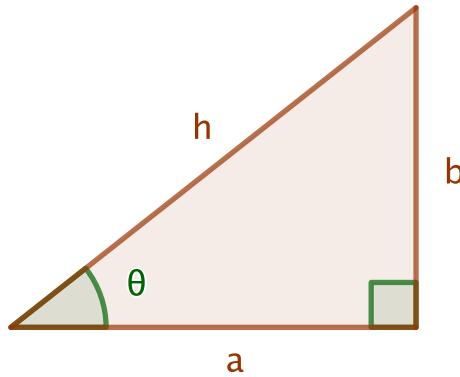


Figure 8: A triangle with sides of length  $a$ ,  $b$  and  $h$ . The longest side  $h$  is called the hypotenuse

The arcsin function is the reverse of the sin function:

$$\frac{a}{b} = \sin \theta \quad \theta = \arcsin \left( \frac{a}{b} \right)$$

Normally you give an angle to the sin function. The result is  $a/b$ , where  $b$  is the hypotenuse and  $a$  the length of the triangle opposite the angle  $\theta$ .

Thus if we want to calculate multiple trajectories for our artillery book we could use Julia as a calculator.

**NOTE Muzzle velocity and range**

Artillery cannons typically have muzzle velocities of 700 to 900 m/s, and shoot 20 to 30 km. In our simplified case we get almost 60 km max distance since we ignore things like air resistance which would significantly reduce artillery range.

Imagine a start velocity  $v = 762.425\text{m/s}$  and we want to know the angles for the distances 8, 12, 16 and 25 km. We could write in the Julia REPL:

```
julia> 0.5*asin(9.81*8000/762.425^2)
0.06771158922454301
```

```
julia> 0.5*asin(9.81*12000/762.425^2)
0.10196244313304187
```

```
julia> 0.5*asin(9.81*25000/762.425^2)
0.217772776595389
```

The angles here are in radians so they will be from  $-\pi$  to  $\pi$ , rather than from 0 to 360.

Imagine doing thousands of these calculations. No wonder they needed rooms full of people calculating! This is going to get boring and tedious even with a calculator. And they did not have electronic calculators but mechanical calculators which could basically only add and subtract.

This poses several problems. We have to keep writing the same long numbers over and over again. It is boring and time consuming to write 762.425 again and again. Sooner or later we will get the wrong answer, mixing up one digit or forgetting another one.

Is there perhaps some way we can get Julia to remember 762.425 for us?

That is how what we do in natural language. Remember how I talked about our ability to talk about context? Humans talking to each other don't need to keep repeating 762.425 when talking about the velocity. They can just say "initial velocity".

## Variables and Constants

That is what we do in the math equations as well. We use the letter  $v$  as a sort of placeholder. Remember I talked about *variables* as one of the important

concepts in programming?

In this case it will actually be a *constant* as our initial velocity will be fixed to the same value every time we do the calculations. But in programming we still usually refer to it as a variable.

So what would be a good way of telling Julia that we want to use a letter or a word to refer to a number?

There are many possible ways of doing this, which are equally valid. It is the designers of the programming language who decide how to do it. What is most important, is that it is done in a manner which is easy for programmers to remember.

Some languages will write it like this:

```
v <- 762.425
```

An early programming language called Pascal made you write it like:

```
v := 762.425
```

But in Julia we write:

```
v = 762.425
```

#### **NOTE**

This is potentially confusing, because it is exactly the same symbol used in mathematics for comparing two numbers. Keep in mind that in Julia (and most other programming languages) it is used for assignment and **not** for comparing values. The mathematical expression  $x = y$  is written  $x == y$  in Julia.

Thus on the Julia REPL (command line) we can input the values for velocity v and the acceleration g caused by gravity:

```
julia> v = 762.425
762.425
```

```
julia> g = 9.81
9.81
```

Once written, Julia will remember the values of v and g and we can keep writing:

```
julia> 0.5*asin(g*12000/v^2)
0.10196244313304187
```

However it is still tedious to write this whole equation. Imagine writing this a thousand times and the only thing you really change is the velocity. Everything else stays the same each time, yet we have to keep writing it.

Is there perhaps a way in which we can get Julia to not just give a name to a number, but to give a name to a whole equation or calculation?

## Functions

You guessed it, functions. Functions allow you to give names to whole calculations. Thus instead of having to remember the details of how something is calculated, you can simply refer to a calculation by name.

Lets make a naive attempt at writing such a function:

```
julia> angle = 0.5*asin(g*12000/v^2)
```

Nope, that won't work. All we did was calculate an angle and stick it in a variable named `angle`. We can write

```
julia> angle
```

Over and over again, but the problem is we get the same result each time. We haven't told Julia yet which variable we want to keep changing the value of. We want to change the distance from 8, 12 to 25 km.

Thus somehow, when we define a function with a name, we have to tell Julia which variable will keep changing. Having used variables already, we can use a variable name to refer to this distance. But how can we tell Julia that the variable named `distance` should keep changing but not the other variables?

## Function arguments

When we define the function we specify function arguments. That is a list of the variables which we want to change each time we use our function.

```
julia> angle(distance) = 0.5*asin(g*distance/v^2)
angle (generic function with 1 method)
```

This defines a function `angle`, with one single argument named `distance`. Now I can write:

```
julia> angle(8000)
0.06771158922454301
```

```
julia> angle(12000)
0.10196244313304187
```

```
julia> angle(25000)
0.217772776595389
```

That is a lot better. Now we can calculate angles much faster and we don't have to memorize the gravitational acceleration, initial velocity and the equation anymore.

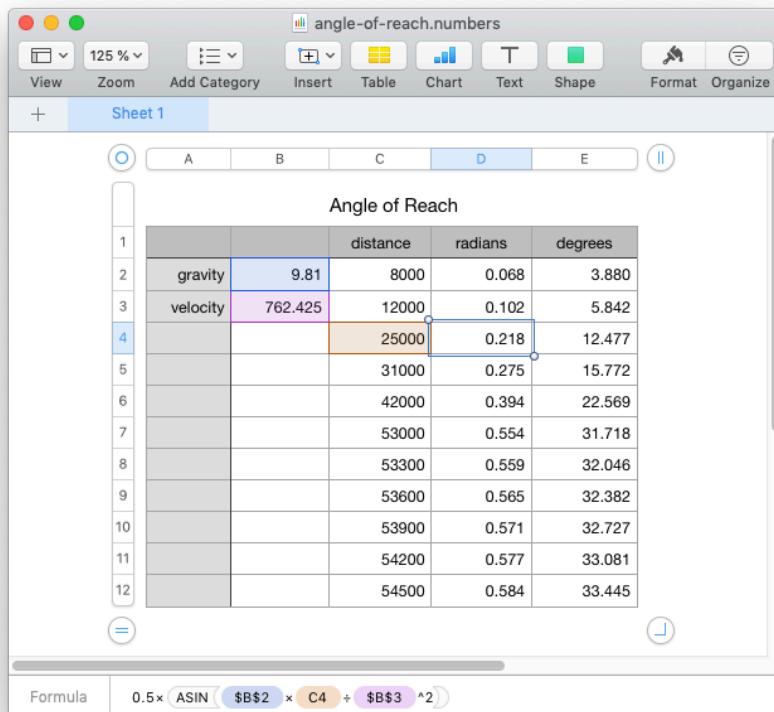
## Arrays

But it is still tedious to do the calculations. Say we have multiple distances in a list:

8, 12, 25, 31, 42

and you want to calculate the angle for each of these distances. Is there a simpler way than writing `angle()` five times? Or what if we had a hundred or thousand numbers. Manually calling `angle` for each of these numbers would be very time consuming.

You may already be familiar with spreadsheet applications such as Microsoft Excel or Apple's Numbers. These programs excel at performing a calculation on multiple numbers stored in tables. Below is an example from a spreadsheet for calculating angles for different distances.



The screenshot shows a Numbers spreadsheet titled "angle-of-reach.numbers". The table is titled "Angle of Reach" and has columns for gravity, distance, radians, and degrees. The formula bar at the bottom shows the formula `0.5 * ASIN($B$2 * C4 / $B$3^2)`.

		distance	radians	degrees
1	gravity	9.81	0.068	3.880
2	velocity	762.425	0.102	5.842
3		12000	0.218	12.477
4		25000	0.275	15.772
5		31000	0.394	22.569
6		42000	0.554	31.718
7		53000	0.559	32.046
8		53300	0.565	32.382
9		53600	0.571	32.727
10		53900	0.577	33.081
11		54200	0.584	33.445
12		54500		

Figure 9: Spreadsheet application Numbers, used to calculate angles

At the bottom of the image you can spot the formula used to calculate angles given distances.

Julia also offers a way of working with tables of numbers. In Julia you create the equivalent of tables with the `Array` data type. Julia arrays allow you to work with numbers in rows and columns. When an array is just a row or column, we call it one-dimensional. When an array is made up of several rows or columns we call it a two-dimensional array or a matrix.

Below are examples of one-dimensional and two-dimensional arrays. We will cover more details of how to work with arrays in the Object Collections chapter.

element indices				column indices			
1	2	3	4	1	2	3	4
5	7	8	9	11	12	13	14
<b>v[2]</b>				21	22	23	24
				31	32	33	34
				<b>A[:, 3]</b>			

A 1D array **v** and a 2D array **A**. We have marked in blue the 2nd element in **v**. It contains value 7. The 3rd column of **A** is marked containing values 12, 23 and 33

Figure 10: 1D and 2D arrays

We can store an array of numbers in a variable just like single numbers (scalars). Notice how we can use `_` to separate digits in long numbers.

```
julia> distances = [8_000, 12_000, 25_000, 31_000, 42_000]
5-element Array{Int64,1}:
 8000
12000
25000
31000
42000
```

#### NOTE Assigning values to variables

Remember assignment means putting a value into a variable. Another way of looking at it, is that you are fixing a label to a value. You can have all sorts of long and complicated numbers and arrays you need to remember. Sticking a label on the number helps you remember it. Just like we can make it easier to remember a calculation by giving it a name. Thus function names are labels stuck on calculations and variables are labels stuck on numbers, arrays or other data types.

The benefit of having numbers in arrays is that you can work with the numbers collectively. For instance you can tell Julia to find the average of all the numbers in an array or the sum of them.

```
julia> sum(distances)
118000

julia> sum([8_000, 12_000, 25_000, 31_000, 42_000])
118000
```

```
julia> 8_000 + 12_000 + 25_000 + 31_000 + 42_000
118000
```

All three expressions are equivalent.

To get the median or mean we would need to use the statistics package.

```
julia> using Statistics
```

```
julia> mean(distances)
23600.0
```

```
julia> median(distances)
25000.0
```

## Accessing Elements

You can get hold of one or more values stored in an array in different ways. If we deal with a 1 dimensional array, then the position of the element is defined by an index. If we have a 2 dimensional array, the position is given by a row and column.

```
julia> distances[1]
8000
```

```
julia> distances[2]
12000
```

```
julia> distances[5]
42000
```

```
julia> distances[end]
42000
```

```
julia> two_dim = [2 4 8; 10 12 14]
2×3 Array{Int64,2}:
 2   4   8
 10  12  14
```

```
julia> two_dim[1, 2]
4
```

```
julia> two_dim[2, 2]
12
```

## Higher Order Functions

You can also use functions which don't produce single numbers as result (single values are called scalars). Some functions take arrays as input and produce

arrays as output.

```
julia> angles_rad = map(angle, distances)
5-element Array{Float64,1}:
 0.06771158922454301
 0.10196244313304187
 0.217772776595389
 0.27527867645775067
 0.3938981904086532
```

`map` is a function which takes two arguments. The first argument is a function and the second is an *array*. When you run `map(f, xs)` it will give you a new array, where each number is the result of applying the function `f` given as first argument to each successive number in the array `xs` given as second argument.

#### NOTE

Higher order functions are functions which take other functions as arguments. So `map` is a higher-order function, while `sum` isn't. Higher-order functions will be covered more in detail in Functional Programming

Let us apply `map` one more time. We got our angles in radians. How about turning all your angles into degrees? You can use the Julia function `rad2deg` for this purpose.

```
julia> rad2deg(pi)
180.0

julia> rad2deg(pi/2)
90.0

julia> angles_deg = map(rad2deg, angles_rad)
5-element Array{Float64,1}:
 3.8795882866898173
 5.842017660365961
 12.477460991761141
 15.772306350976407
 22.56870386825631
```

Functions such as `sum` and `map` allow us to do what early computers did, when they made artillery trajectory tables: perform the same calculation repeatedly. This makes a computer different from a calculator. Calculators must be manually operated for every repeated calculation.

These functions may seem magical. Somehow they are able to look at each individual element in an array and do something with it. How do you do that? Can you build functions like this yourself? Yes you can!

## Loops

Most programming languages today have what we call loop constructs or statements. The common ones are the *for-loop* and the *while-loop*.

Now we will do something we haven't done thus far. We will write functions spanning multiple lines. When writing statements or functions which span multiple lines we need to inform Julia of where they begin and end. All the code between the function and end keyword is part of the function. Here is the angle function written over multiple lines:

```
function angle(distance)
    0.5*asin(g*distance/v^2)
end
```

This is more typical Julia syntax. We usually start a programming statement with a keyword such as `for` or `function` telling Julia what sort of statement it is. A statement is vaguely the same thing as a sentence in a natural language. Statements can be nested. A Julia program is basically a long list of statements in which each of the statements in the list may contain other statements.

Here we write our own `sum` function called `addup` using a *for-loop*

```
function addup(numbers)
    total = 0
    for num in numbers
        total = total + num
    end
    total
end
```

The first statement is a `function` statement which Julia can figure out by looking at the first keyword. The very last `end` keyword indicates the end of the function statement.

The *for-loop* starts with the `for` keyword and ends with `end`. Every statement between these two lines are performed multiple times. One way to think about how a function is executed (performed) is to imagine a recursive substitution of variables and function calls for values.

## A Step by Step Evaluation of a Function

Let us look at how a function call like this is evaluated:

```
nums = [2, 4, 6]
total = addup(nums)
```

### Step 1

We substitute `nums` for its value, the array `[2, 4, 6]`.

```
total = addup([2, 4, 6])
```

### Step 2

When calling the function we've got to imagine substituting the arguments of the function with the passed value:

```
function addup([2, 4, 6])
    total = 0
    for num in [2, 4, 6]
        total = total + num
    end
    total
end
```

### Step 3

Lets focus on the *for-loop* alone. We will successively assign the variable `num` a value in the array `[2, 4, 5]`.

```
for 2 in [2, 4, 6]
    total = 0 + 2
end
```

### Step 4

`total` has a new value `2` on next iteration.

```
for 4 in [2, 4, 6]
    total = 2 + 4
end
```

### Step 5

`total` is now `6`. While `num` is also `6` as that is the last value in the `numbers` array.

```
for 6 in [2, 4, 6]
    total = 6 + 6
end
```

### Step 6

The last value in a function evaluates to the whole value of the function, so `total = addup(nums)` becomes:

```
total = 12
```

## Different Ways of Looping Over an Array

If you write e.g. `for x in [2, 4, 6]` then `x` will successively take on the values `2, 4` and `6`. That is, it will take on the values of the elements in the given array.

However it is also possible to iterate over a range of numbers rather than the values of an array. In Julia we describe a range as `i:j` where `i` is the start of

the range and `j` is the last index (number) in the range. `2:6` is an example of a range.

```
total = 0
for x = 2:6
    total = total + x
end
```

`total` will get the value `2 + 3 + 4 + 5 + 6`, as those are the values `x` will successively assume.

You can use ranges in all sort of circumstances. You can use them instead of specifying an array.

```
julia> sum(2:6)
20

julia> sum([2, 3, 4, 5, 6])
20
```

A more basic but equivalent way of writing the first iteration is to use a *while-loop*

```
total = 0
x = 2
while x <= 6
    total = total + x
    x = x + 1
end
```

In this case we will keep performing the lines between `while` and `end` until the condition `x <= 6` is no longer true. So when `x` turns into 7, it will no longer be true as 7 is not less than or equal to 6. We can compare numbers with these operators:

- `<` less than
- `>` greater than
- `<=` less than or equal
- `>=` greater than or equal
- `==` equal to
- `!=` not equal to

You might wonder where I am going with all this? This is prerequisite knowledge to be able to explain how you can write a `map` function yourself. Still we have only looked very briefly at how you access individual values in an array.

## Make Your Own Map Function

We now have the building blocks to create our own map function. Let me take you through some of the individual parts we need to make.

We get some input array, and we need to create an array for the results. Let us make an empty array:

```
julia> angles = []
0-element Array{Any,1}
```

That doesn't look right. It says we made an empty array to hold Any value. That means we could put Bool, AbstractString or whatever there. Since we didn't put any numbers into it, Julia isn't able to figure out what we intend to use it for. So we need to help out Julia by writing the Type of the items as a prefix:

```
julia> angles = Float64[]
0-element Array{Float64,1}
```

We can use a for loop to iterate over distances and calculate angles.

```
julia> for dist in distances
           push!(angles, angle(dist))
       end
```

`push!` is a function which pushes values at the end of an array. You can see the previously empty `angles` array has been filled with the same results as we got previously with `map`.

```
julia> angles
5-element Array{Float64,1}:
 0.06771158922454301
 0.10196244313304187
 0.217772776595389
 0.27527867645775067
 0.3938981904086532
```

When we put all these parts together we get our own `map` function. Let us call the function `transform`. This map function is of course not as flexible as the one bundled with Julia. For instance ours assume the output is always floating point numbers.

We cannot solve that problem until we have covered more about types. The only types you know of thus far are different types of numbers.

```
function transform(fun, xs)
    ys = Float64[]
    for x in xs
        push!(ys, fun(x))
    end
    ys
end
```

## Picking Array Elements

With our `transform` function we took every element in the array and transformed it to another value, resulting in a new array of the same size.

However another useful task is to go over all the elements of an array and only pick particular values that we like to keep. Usually that means we get an array back which is smaller than the one we started with.

When might this be useful?

The firing tables for artillery cannons are usually calculated from more complicated equations that we have used here. When creating these equations, engineers and scientists have to make sure that they get accurate results.

One can do that by firing the cannons repeatedly and recording the range for different angles. One way to estimate the range is to calculate the average distance of several cannon balls fired at the same angle.

When writing down lots of measurements it is common that people make mistakes such as forgetting the decimal point, writing what looks like a 7 where there should have been a 1 and so on.

Say these were the actual **measured** distances in kilometers, which should have been recorded:

11.5, 10.8, 11.4, 12.1, 12.2, 10.9

However due to sloppy writing, these are the **recordings** the scientists got:

11.5, 10.8, 71.4, 12.1, 122, 10.9

We can write our own mean function to calculate the average.

```
mean(xs) = sum(xs)/length(xs)
```

Then we assign the accurate measured distances and poorly recorded distances to two separate variables.

```
measured = [11.5, 10.8, 11.4, 12.1, 12.2, 10.9]
recorded = [11.5, 10.8, 71.4, 12.1, 122, 10.9]
```

We can then calculate the average in the REPL.

```
julia> mean(measured)
11.48333333333334
```

```
julia> mean(recorded)
39.78333333333334
```

We get a result which is almost 4x off. Imagine we got hundreds of numbers in a table. It may not be easy to spot the numbers that are off. This is when Julia's `filter` function is handy.

Remember how `map(f, xs)` function applies a function `f` on every element in `xs`? Function `f` takes a value `x` and returns another value `y`:

$$y = f(x)$$

`filter` is similar but it expects a function that returns `true` or `false` instead of a number. We call this a *boolean* value. Where do these values come from?

```
julia> x = 4
4

julia> x > 5
false

julia> x < 5
true

julia> x == 4
true

julia> x != 4
false
```

You can see that expression using comparison operators such as `>`, `<` and `==` gives `true` or `false` as result. Functions which give a boolean result instead of a number are called *predicates*. Let us define a *predicate* to use with our filter function.

```
isValid(x) = x < 14
```

This is an example of using it.

```
julia> isValid(11)
true
```

```
julia> isValid(15)
false
```

We can use it with `filter` to get only valid *distances* and calculate the average.

```
julia> filter(isValid, recorded)
4-element Array{Float64,1}:
 11.5
 10.8
 12.1
 10.9
```

```
julia> mean(filter(isValid, recorded))
11.325
```

For the actual measured values this filtering will have no impact on the result.

```
julia> filter(isValid, measured)
6-element Array{Float64,1}:
 11.5
 10.8
 11.4
 12.1
 12.2
 10.9
```

```
julia> mean(filter(isValid, measured))
```

```
11.48333333333334

julia> mean(filter(isvalid, measured)) == mean(measured)
true

julia> mean(filter(isvalid, recorded)) == mean(recorded)
false
```

## Make Your Own Filter Function

Based on what we have learned making the `transform` function we could begin by writing this function. We are no longer transforming the input values, but the problem with this version is that it is adding every value `x` in the array `xs`.

```
function pick(pred, xs)
    ys = Float64[]
    for x in xs
        push!(ys, x)
    end
    ys
end
```

What we need is a way to make a *decision* on whether the `x` should be added or not. To make decision on what code to run we use what is called an *if-statement*.

In the code segment below, the `x` is only added to the end of the `ys` array if it is below 14.

```
if x < 14
    push!(ys, x)
end
```

The code between `if` and `end` is only run if the expression `x < 14` evaluates to the boolean value `true`. This if-statement is equivalent to:

```
if isvalid(x)
    push!(ys, x)
end
```

It does not matter what `isvalid` does with `x` as long as it evaluates to (returns) a *boolean* value (`true` or `false`).

With this knowledge we can modify the `pick` function to make it work.

```
function pick(pred, xs)
    ys = Float64[]
    for x in xs
        if pred(x)
            push!(ys, x)
        end
    end
    ys
end
```

In this version we use the predicate function `pred` passed in as first argument with the if-statement to decide whether an element should be added or not.

Let us test out our function and see if it works.

```
julia> pick(isvalid, [4, 14, 18, 3, 1])
3-element Array{Float64,1}:
 4.0
 3.0
 1.0
```

```
julia> pick(isvalid, [16, 18])
0-element Array{Float64,1}
```

We can compare it with Julia's built-in `filter` function to see if it gives the same result

```
julia> xs = [4, 14, 18, 3, 1]
5-element Array{Int64,1}:
 4
 14
 18
 3
 1

julia> pick(isvalid, xs) == filter(isvalid, xs)
true
```

# Working with Text

- **Strings** for working with text.
- **Printing output** to the user with and without colors.
- **Random Number**. Different ways of generating random numbers.
- **If statement** . Execute code only if a condition is met.
- **Debugging**. Locating and fixing flaws in your code.

In this chapter we will go through a code example for practicing the multiplication table. The purpose of the example is to show different ways of working with text strings. We will look at a data type called `String`, for storing and manipulating text and look at ways of showing text to the user in different colors.

I have intentionally introduced mistakes in the code (bugs), so we can later practice debugging it. *Debugging* is what programmers call looking for bugs in their programs. In later chapters we will look at more advance methods of debugging

## Teaching Multiplication

Our next example program is for teaching children to do multiplication. Or you can use it yourself to get quicker at doing multiplication in your head.

We will write this using loops and by defining functions. This way you get to practice what you learned in previous chapter. We will also introduce some new functionality such as writing text to user and reading input from the user.

Above you can see an example of using the program. It repeatedly asks you to multiply some numbers, check your answer and keeps track of your score. It even tracks how fast you do it, so you can practice doing multiplication faster.

## Creating Random Numbers

The program works by repeatedly asking the user to give the answer the multiplication of two random numbers. We use `rand()` to create a random number in Julia:

```
julia> rand()  
0.8579770262264026
```

Without arguments `rand` gives a value between 0 and 1.

```
julia> practice(4)
4*9 = 36
Correct!

9*2 = 18
Correct!

9*6 = 3
9*6 = 54 Wrong

5*5 = 25
Correct!

3/4
Time: 10.0

julia>
```

Figure 11: Program for practicing multiplication

However we don't want to multiply numbers with lots of decimals. We want whole numbers (integers) to multiple. Fortunately we can do that.

```
julia> rand(3:6)
3

julia> rand(3:6)
4
```

This creates random numbers in a range you specify. From 3 to 6 in this case. The ranges you specify are inclusive. That means they include both the starting number and ending number. If you write e.g. 7:16 that would mean a number from 7 to 16 including both 7 and 16.

## Asking User To Answer a Question

Let us pick two numbers, call them `a` and `b`, which we will use to ask our user to multiply. We use `rand` to pick these numbers at random.

```
a = rand(1:10)
b = rand(1:10)
```

Once we get the numbers, we need a way of inserting them into a question, which we display to the user.

```
print(a, "*", b, " = ")
```

We need to place all these 3 lines of code into a text file to run the code. Put it in a file named `arithmetic-practice.jl` for example. Use a code editor which can read and write UTF-8 encoded text files, as I've described earlier.

The `.jl` filename extension helps your code editor guess that this is a Julia source code file. If it had ended with `.py` it would have guessed it was a Python source code file, while if it had ended with `.js` it would have guessed JavaScript.

After you save the file, you can run the code in the file by writing at the command line:

```
$ julia arithmetic-practice-1.jl  
7*4 = ↵  
$ julia arithmetic-practice-1.jl  
8*1 = ↵  
$ julia arithmetic-practice-1.jl  
9*6 = ↵
```

**NOTE**

Notice you only write `julia` `aritmetic-practice.jl` and hit enter. I write `$` and `julia>` prompts to help you see which program you are entering commands into.

The `print` function is used to display text strings on the screen. It can also convert other data types such as numbers to text. Here are some examples. Print a single string:

```
julia> print("hello world")  
hello world
```

`print` can take multiple arguments separated by comma. Thus you can input multiple strings:

```
julia> print("hello", "world")  
helloworld
```

Combining string literals, variables and numbers:

```
julia> name = "Joe"  
"Joe"
```

```
julia> age = 50  
50
```

```
julia> print(name, " are you really ", age, "?")  
Joe are you really 50?
```

## Reading a Reply

Of course there is not much point in asking a question if you can't get an answer. You can read from files, such our code file `aritmetic-practice.jl` with the `readline()` function.

```
julia> readline("aritmetic-practice.jl")  
"a = rand(1:10)"
```

But we don't want to read from a file, but rather what the user writes on the keyboard. Fortunately as a heritage from Unix systems, the keyboard is treated as just another file. In Julia this file is called `stdin`. Your screen or more specifically a console window (command line text interface) is represented by a special file called `stdout`. Writing text to this file would make the text pop up in a console window.

```
julia> readline(stdin)
```

Now Julia will wait for input from your keyboard. Let us say we write:

```
Hello world
```

As soon as you hit the enter key, you will return to the Julia REPL. The `readline()` function will return what you wrote as a text string:

```
"Hello world"
```

## If-Else Statement

Once we got a reply from the user, we have to check if it is correct or not, to be able to tell the user and update his/her score. For this we are going to use the *if-statement* we previously used to implement `pick` in the previous chapter, but with a twist.

The *if-statement* comes in different forms. Here we will use the *if-else* form.

```
c = readline(stdin)
if a*b == c
    println("Correct!")
else
    println("Wrong")
end
```

The comparison we perform `a*b == c` is called the condition. If the condition is `true`, we will run all the code between `if` and `else`. If the condition is `false` we instead run the code between `else` and `end`.

We then update our whole `julia arithmetic-practice.jl` source code file to contain:

```
a = rand(1:10)
b = rand(1:10)
print(a, "*", b, " = ")
c = readline(stdin)
if a*b == c
    println("Correct!")
else
    println(a, "*", b, " = ", a*b, " Wrong")
end
```

Let us run the program and see how it works.

```
$ julia arithmetic-practice.jl
9*9 = 81
```

```
9*9 = 81 Wrong  
$ julia arithmetic-practice.jl  
7*8 = 56  
7*8 = 56 Wrong
```

Notice that, even when you write the correct answer, Julia prints out “Wrong.”

## Debugging

We will have to debug this code to find out what is wrong. Debugging is the process of looking for a defect in a program. I will show you a primitive way of doing debugging.

In the editor window with your source code, you copy one line of code at a time, and paste it into the Julia REPL. This way you get to see the result of executing every line of code in your program. It allows you to simulate stepping through the code, one line at a time.

```
julia> a = rand(1:10)  
7  
  
julia> b = rand(1:10)  
6  
  
julia> a*b  
42  
  
julia> c = readline(stdin)  
42  
"42"
```

Now that we have the inputs, let us check if the if-statement behaves as we expect.

```
julia> a*b == c  
false
```

Hmm... this doesn't make sense. Lets check more thoroughly:

```
julia> a*b == 42  
true  
  
julia> c == 42  
false  
  
julia> c  
"42"  
  
julia> 42 == "42"  
false
```

Do you see the problem? Julia distinguishes between the number `42` and the text string “`42`”. We need to turn the text into a number. We use the `parse()` function to turn text strings into other types of objects.

```
julia> parse(Int, "42")
42
```

So we got to change this line of code:

```
c = readline(stdin)
```

Into:

```
c = parse(Int, readline(stdin))
```

Now you can re-run the program and it works:

```
$ julia aritmetic-practice.jl
2*3 = 6
Correct!
```

## Repeatedly Asking Questions

Of course the program is rather useless at the moment as it can only ask the user for the correct answer **once**. What we need is a way of repeating the same code multiple times.

Here is a chance to utilize the for-loop again to repeatedly ask the user to give the correct answer to a multiplication question.

```
for i in 1:4
    a = rand(1:10)
    b = rand(1:10)
    print(a, "*", b, " = ")
    c = parse(Int, readline(stdin))
    if a*b == c
        println("Correct!")
    else
        println(a, "*", b, " = ", a*b, " Wrong")
    end
end
```

You can run this with `$ julia aritmetic-practice.jl` and see that you get asked to answer four times.

## Keep Track of the Score

If we ask a lot of questions, it is useful to keep track of the number of correct answers.

We'll keep track of the score in a variable called `score`. Each time we got a correct answer we add `1` to it. We can write that as:

```
score = score + 1
```

But in most programming languages we have a short cut for modifying a variable:

```
score += 1
```

Let us change our program to keep track of the score. You'll notice I am using `n` to keep track of number of questions, since we need to know this when telling the user how many correct answers they got out of the total number of questions.

```
n = 4    # number of questions
score = 0
for i in 1:n
    a = rand(1:10)
    b = rand(1:10)
    print(a, "*", b, " = ")
    c = parse(Int, readline(stdin))
    if a*b == c
        println("Correct!")
        score += 1
    else
        println(a, "*", b, " = ", a*b, " Wrong")
    end
end
println("Score $score/$n")
```

In the last line, we use something called string interpolation to put the values of variables `score` and `n` into the text string.

## Writing Text in Color

You'll notice in the picture of my original program you can see colors. For that I am using the function `printstyled()` like this:

```
printstyled(" Wrong\n", color=:red)
printstyled("Correct!\n", color=:green)
```

The `\n` at the end of each text string means a new line. Usually I use `println()` instead of `print()` when I want a new line, but that is not an option in this case.

`color=:green` is a named function argument. The argument is named `color` and given the value `:green`. Named arguments are often used for optional arguments. They are a benefit when you have many arguments because named arguments can be put in any order. Their name, rather than their position identifies them.

**NOTE What does 'foo' and 'bar' mean?**

Throughout this book you will notice the usage of nonsense names such as `foo`, `bar`, `baz` and `foobar`. This can trip up beginners who wonder about the special significance of these names.

They have no special significance. It is simply that they have been used in code examples for so many years.

The utility of these names is that, as soon as you see one of them you are made aware of that the name used is completely arbitrary.

For instance if I write `pi` in code it refers to an actual predefined constant, while no programming language ever has `foo`, `bar` and `foobar` as predefined constants or functions. Hence these words are useful in code examples.

Some languages such as Python have their own conventions, preferring to use the words `spam`, `ham` and `eggs` in their code examples. This conventions is a humorous reference to a 1972 Monty Python sketch called ``Spam''.

Colors are indicated with names such as `:blue`, `:cyan`, `:green` and `:light_black`. You might wonder about the colon in front? Just like strings are encapsulated with quotation marks ", symbols are prefixed with a colon. Symbols are almost the same as a string. However you don't use symbols to write messages, but to give unique names to options or choices.

#### **NOTE Symbols**

Only for the very curious. Most programming languages don't have a `Symbol` type like Julia. However there are others such as LISP and Ruby that do. Usually they exist in languages which supports representing their own code as data structures you can manipulate. Consider the code:

```
foo = "bar"
```

When storing all the parts of this code, we get the objects `:foo`, `:=` and `"bar"`. While this line:

```
foo = bar
```

Gives us the objects `:foo`, `:=` and `:bar`, thus allowing us to distinguish between two different cases.

I wont show you how to add colors to the program. Look at the chapter exercises to solve it for yourself.

## Using Strings

Now that you have been introduced to strings, let us look at their usage in more detail. Strings are made of individual characters.

```
julia> chars = ['H', 'e', 'l', 'l', 'o']
5-element Array{Char,1}:
```

```
'H'  
'e'  
'l'  
'l'  
'o'
```

A character is written in single quotes, so 'H' is a character while "H" is a string with a single character. An array of characters can be combined using `join()`, to create a text string.

```
julia> s = join(chars)  
"Hello"
```

The characters making up a string are numbered. You can get hold of an individual character by giving an index number. The first character is at index 1.

```
julia> s[1]  
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)  
  
julia> s[5]  
'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
```

The last character is at index 5 in this example, but there is a shortcut if you don't want to count characters, using the `end` keyword.

```
julia> s[end]  
'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
```



# Storing Data in Dictionaries

- **Dictionary** for storing key-value pairs.
- **Pairs**. A datatype for storing two related values.
- **Tuples**. Collection of one or more values of different types.

We will look at a new data type called a dictionary, by working through a code example for converting roman numerals to decimal values and back. We will use a dictionary to keep track of what value a letter such as I, V and X corresponds to in the decimal system.

## Roman Numerals

While roman numerals are not very practical to use today, they are useful to learn about in order to understand number systems. In particular when programming you will encounter various number systems.

Both Roman numerals and the binary, system used by computers, may seem very cumbersome to use. However it often appears that way because we don't use the numbers as they were intended.

It is hard to make calculations using Roman numerals with pen and paper compared to Arabic numerals (which is what we use). However the Romans did not use pen and paper to perform calculations. Rather they performed their calculations using a roman abacus.

It is divided into multiple columns. You can see the I, X and C column:

- In the I column every pebble is a 1.
- In X, every pebble represent 10.
- In C, every pebble represent 100.

Above each of these columns we got the V, L and D columns, which represent the values 5, 50 and 500.

### NOTE

The beauty of the Roman system is that you can quickly write down exactly what the pebbles on the abacus say. Likewise it is



Figure 12: Roman abacus

quick to arrange pebbles on a Roman abacus to match a Roman numeral you have read. For this reason Roman numerals were used all the way into the 1500s in Europe, long after Arabic numerals had been introduced.

Let us look at how we can use this knowledge to parse roman numerals and turn them into Arabic numerals. Put the code below into a text file and save it.

```
roman_numerals =
    Dict('I' => 1, 'X' => 10, 'C' => 100,
        'V' => 5, 'L' => 50, 'D' => 500,
        'M' => 1000)

function parse_roman(s)
    s = reverse(uppercase(s))
    vals = [roman_numerals[ch] for ch in s]
    result = 0
    for (i, val) in enumerate(vals)
        if i > 1 && val < vals[i - 1]
            result -= val
        else
            result += val
        end
    end
    result
end
```

Load this file into the Julia REPL environment to test it out. This is an example of using `parse_roman` with different roman numerals as input.

```
Roman Numerals — ⌘2
julia> parse_roman("II")
2
julia> parse_roman("IV")
4
julia> parse_roman("VI")
6
julia> parse_roman("IX")
9
[julia> parse_roman("XI")]
11
julia> ]
```

Let us go through how the code works.

## The Dict Type

We map or translate the Roman letter I, V, X etc to numbers using what is called a dictionary. A dictionary is made up of multiple pairs.

```
julia> 'X' => 10          # 1
'X' => 10

julia> pair = 'X' => 10    # 2
'X' => 10

julia> dump(pair)          # 3
Pair{Char,Int64}
  first: Char 'X'
  second: Int64 10

julia> pair.first          # 4
'X': ASCII/Unicode U+0058 (category Lu: Letter, uppercase)

julia> pair.second
10
```

1. A pair of the letter X and 10.
2. Pairs can be stored in a variable and examined later.
3. Dump allows us to look at the fields of any value. The fields of a pair value in this case.
4. Extracting the first value in the pair.

We provide a list of these pairs to create a dictionary.

The code below shows how we create a dictionary to map letters used by Roman numerals to their corresponding decimal value.

```
julia> roman_numerals =
    Dict('I' => 1, 'X' => 10, 'C' => 100,
         'V' => 5, 'L' => 50, 'D' => 500,
         'M' => 1000)
Dict{Char,Int64} with 7 entries:
'M' => 1000
'D' => 500
'I' => 1
'L' => 50
'V' => 5
'X' => 10
'C' => 100
```

When used in a dictionary we refer to the first values in each pair as the keys in the dictionary. The second values in each pair form the values of the dictionary. So I, X and C are keys, while 1, 10 and 100 e.g. are values.

We can ask a dictionary for the value corresponding to a key. This takes a Roman letter and returns the corresponding value.

```
julia> roman_numerals['C']
100

julia> roman_numerals['M']
1000
```

## Looping over Characters

We can use this dictionary to help us convert roman letters to corresponding values. In the `parse_roman` function we do this conversion with `[roman_numerals[ch] for ch in s]`. This is called an **array comprehension**. We will look at a regular for-loop doing exactly the same thing first. This makes it easier to understand what the array comprehension does.

In this example we start with roman numerals “XIV” which we want to convert.

```
julia> s = "XIV"
"XIV"

julia> vals = Int8[]
0-element Array{Int8,1}
```

```
julia> for ch in s
           push!(vals, roman_numerals[ch])
       end

julia> vals
3-element Array{Int8,1}:
 10
  1
  5
```

“XIV” is turned into the array of values [10, 2, 5] named `vals`. However the job is not quite done. Later we need to combine these values into one number.

Before converting input strings, our code turns every letter into uppercase. “xiv” would not get processed correctly, because all the keys to our dictionary are uppercase.

We reverse the input, so we can process the lowest values first.

```
julia> s = "xiv"
"xiv"

julia> s = reverse(uppercase(s))
"VIX"
```

## Enumerate

In our for-loop we need to keep track of the index of the value `val` of each loop iteration. To get the index we use the `enumerate` function. That is what you see used in the line `for (i, val) in enumerate(vals)`. Here is a simple demonstration of how it works:

```
julia> collect(2:3:11)
4-element Array{Int64,1}:
 2
 5
 8
11

julia> collect(enumerate(2:3:11))
4-element Array{Tuple{Int64,Int64},1}:
 (1, 2)
 (2, 5)
 (3, 8)
 (4, 11)
```

The `collect` function will simulate looping over something, just like a for-loop. Except it will `collect` all the values encountered into an array, which it returns. So you can see with `enumerate` you get a pair of values upon each iteration: an integer index and the value at that index.

## Conversion

We cannot simply add up the individual roman letters converted to their corresponding values. Consider the roman number XVI. It turns into [10, 5, 1]. We could add that and get the correct result 16. However XIV is supposed to mean 14, because with Roman numerals when you got a smaller value in front of a larger one, such as IV, then you subtract the smaller value from the larger.

So we cannot just sum up the corresponding array [10, 1, 5]. Instead we reverse it and work our way upwards. At every index we ask if the current value is lower than the previous one. If it is, we subtract from the result. Otherwise we add.

```
if i > 1 && val < vals[i - 1]
    result -= val
else
    result += val
end
```

That is what `val < vals[i - 1]` does. It compares the current value `val`, to the previous value `vals[i - 1]`. `result` is used to accumulate the value of all the individual Roman letters.

## Using Dictionaries

Now that we have looked at a practical code example utilizing the dictionary type `Dict` in Julia, let us explore some more ways of interacting with a dictionary.

### Creating Dictionaries

There are a multitude of ways to create a dictionary. Here are some examples. Multiple arguments, where each argument is a pair object:

```
julia> Dict("two" => 2, "four" => 4)
Dict{String,Int64} with 2 entries:
  "two"  => 2
  "four" => 4
```

Pass an array of pairs to the dictionary constructor (a function named the same as the type it makes instances of).

```
julia> pairs = ["two" => 2, "four" => 4]
2-element Array{Pair{String,Int64},1}:
  "two"  => 2
  "four" => 4

julia> Dict(pairs)
Dict{String,Int64} with 2 entries:
  "two"  => 2
  "four" => 4
```

Pass an array of tuples to the dictionary constructor. Unlike pairs, tuples may contain more than two values. For dictionaries they must only contain a key and a value though.

```
julia> tuples = [("two", 2), ("four", 4)]
2-element Array{Tuple{String,Int64},1}:
 ("two", 2)
 ("four", 4)

julia> Dict(tuples)
Dict{String,Int64} with 2 entries:
 "two"  => 2
 "four" => 4
```

Create an empty dictionary and fill it up later with key-value pairs.

```
julia> d = Dict()
Dict{Any,Any} with 0 entries
```

Notice the {Any, Any} part. This describes what Julia has inferred is the type of the key and value in the dictionary. Compare this with the other examples where you see {String, Int64}. When you provide some keys and values upon creation of the dictionary, Julia is able to guess the type of the key and value. When you create an empty dictionary, Julia cannot guess the types anymore and assumes the key and value could be Any type.

You can however explicitly state the type of the key and value:

```
julia> d = Dict{String, Int64}()
Dict{String,Int64} with 0 entries

julia> d["five"] = 5
5
```

Which means if you try to use values of the wrong type for key and value, you will get an error (something called an exception is thrown). In this case we are trying to use an integer 5, as key when a text string key is expected.

```
julia> d[5] = "five"
ERROR: MethodError: Cannot `convert` an object of type Int64 to an object of type String
Closest candidates are:
 convert(::Type{T}, !Matched::T) where T<:AbstractString at strings/basic.jl:209
 convert(::Type{T}, !Matched::AbstractString) where T<:AbstractString at strings/basic.jl:210
 convert(::Type{T}, !Matched::T) where T at essentials.jl:171
```

Types will be covered in the More on Types chapter.

Sometimes you get keys and values in separate arrays. However you can still combine them into pairs, to create dictionaries using the `zip` function.

```
julia> words = ["one", "two"]
2-element Array{String,1}:
 "one"
 "two"
```

```
julia> nums = [1, 2]
2-element Array{Int64,1}:
 1
 2

julia> collect(zip(words, nums))
2-element Array{Tuple{String,Int64},1}:
 ("one", 1)
 ("two", 2)

julia> Dict(zip(words, nums))
Dict{String,Int64} with 2 entries:
 "two" => 2
 "one" => 1
```

## Element Access

We have already looked at one way of getting and setting dictionary elements. But what happens if we try to retrieve a value for a key that does not exist?

```
julia> d["seven"]
ERROR: KeyError: key "seven" not found
```

We get an error. We can of course simply add it:

```
julia> d["seven"] = 7;
```

```
julia> d["seven"]
7
```

But how do we avoid producing an error when we are not sure if a key exists? One solution is the `get()` function. If the key does not exist, a sentinel value is returned instead. The sentinel can be anything. The example below uses `-1`.

```
julia> get(d, "eight", -1)
-1
```

Or we could simply ask the dictionary if it has the key.

```
julia> haskey(d, "eight")
false
```

```
julia> d["eight"] = 8
8
```

```
julia> haskey(d, "eight")
true
```

# How Does a Computer Work?

- **CPU** or Central Processing Unit. The brain of a computer, carrying out program instructions.
- **ALU** or Arithmetic Logic Unit of a *CPU*. Carries out all calculations.
- **Abacus**. Exploring arithmetic on this old device to help explain how a computer performs calculations.
- **Mechanical Calculators**. How does a mechanical calculator perform arithmetic and how does that relate to the workings of a computer?

In the first chapter we looked at how computers came about to automate large number of calculations.

When you look at a modern computer with fancy graphics, animations and audio it may not be immediately apparent that a computer fundamentally only manipulates numbers.

Almost anything can be represented by numbers. The colorful screen you look at is made up of millions of tiny pixels, each producing its own color. Numbers are used to tell the computer how much current it should send to each red, green and blue light embedded within every pixel on the screen. The current affects how bright each of these colors shine.

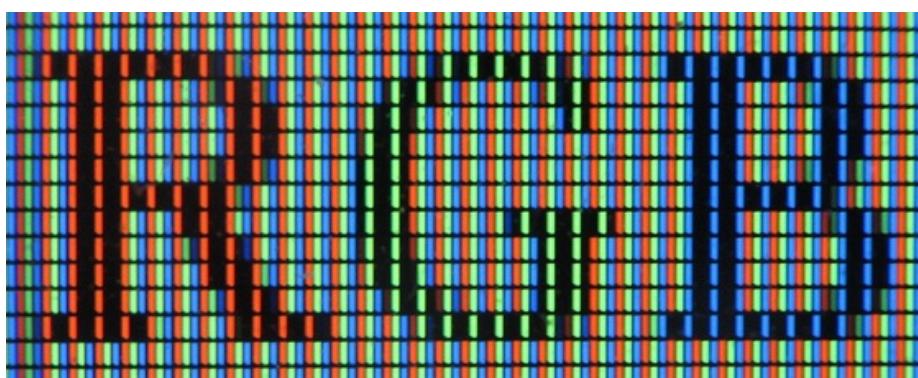


Figure 13: A computer screen seen close up, showing individual pixels. Credit: Luís Flávio Loureiro dos Santos

Likewise there are numbers that say what tone, duration and volume your computer's sound card should produce. These are all big topics in their own right, and it is easy to get lost in the complexity.

Thus we are going to look at the very basics of *how* a computer manipulates numbers.

In the previous chapters we had the mathematician's hat on. We did not care *how* Julia accomplished anything it did. Code was mostly treated like mathematics. Now we are putting on the engineer's hardhat to look at the technical details of how the computer performs tasks. This will help you get a more comprehensive understanding of what programming entails.

## The CPU and the ALU

The brain of a computer is what we may call the *Central Processing Unit* or CPU for short. It reads programs from computer memory and carries out the instructions it is given.

We will focus on one part of the CPU called the *Arithmetic Logic Unit (ALU)*. It is basically the computer's very own calculator.

The ALU cannot actually do very much. A typical ALU can only perform the following operations:

- **Add** two numbers.
- **Subtract** two numbers.
- **Shift** a number. In our ten based number system that means multiplying a number with ten, hundred, thousand etc or divide by those numbers (with remainder).

This may seem *very limiting*. The ALU cannot even multiply, divide, calculate the power of a number or compute the logarithm of a number.

Why can the computer shift but not multiply or divide? If you think about it, those operations are quite simple. They involve adding or removing zeros.

Early computers could not even handle decimal numbers, only integers such as -3, 0, 1, and 42.

But none of these limits are as problematic as they seem. Remember while the ALU can only perform simple arithmetic, the CPU can in fact perform *control-flow* operations such as loops and if-statements.

Let us look at a simple way of implementing integer multiplication and division through looping.

```
function multiply(x, y)
    product = 0
    for i in 1:x
        product += y
    end
    product
end
```

What this code does is performing multiplication through repeated additions. It multiplies 4 by 5, by simply adding up 4, five times.

We can even perform division in this manner. The terminology used in integer division is as follows:

```
divident ÷ divisor = quotient, remainder
11 ÷ 3 = 3, 2
```

To perform this requires slightly more complicated code.

```
function divide(dividend, divisor)
    quotient = 0
    remainder = dividend

    while remainder - divisor >= 0
        remainder -= divisor
        quotient += 1
    end
    quotient, remainder
end
```

This code keeps track of how many times we have subtracted the divisor from the dividend. We do this by assigning the dividend to the remainder. Thus on each iteration we subtract from the remainder. Once we reach a point where subtracting the divisor will lead to a negative remainder we stop.

The quotient simply keeps track of how many times we had to loop through.

Here is an example of usage:

```
julia> divide(8, 2)
(4, 0)

julia> divide(10, 3)
(3, 1)

julia> quotient, remainder = divide(11, 3)
(3, 2)

julia> quotient
3

julia> remainder
2
```

The divide function shows a couple of interesting things:

- That in some cases a **while-loop** makes more sense than a *for-loop*. In this case because we cannot know in advance how many iterations (repetitions of code inside the loop) are needed.
- A function returning **multiple values**. In fact Julia supports returning any number of values from a function. You just need to separate them with a comma.

## Efficiency of Code

The approach we have used here is not very efficient. If I multiply 1000 with 1 using my `multiply` function, it would require 1000 additions. One of the world's first computers the Z1<sup>2</sup> had a clock frequency of 1 Hz. That meant it could do one operation per second. Adding one number would count as one operation. Hence multiplying 1000 with 1 using this approach would take more than 17 minutes.

A problem with understanding how a modern computer works, is that it is so blindingly fast that it is impossible to watch it carry out each step of a task.

That goes for a modern electronic calculator as well. If you multiply two large numbers, it happens so fast that it appears as if it only involved a single operation or step.

For this reason it is interesting to look at old *mechanical* calculators because they had all the same limitations as a basic ALU. They can only perform addition, subtraction and shifting as one operation. Their behavior is easier to observe because they are slow, and involve visible mechanical movements rather than invisible electric current.

## Simulating a Mechanical Calculator

Thus our next challenge is to write Julia code simulating the operations of a famous handheld mechanical calculator called the Curta<sup>3</sup>, which you can see depicted below.

The Curta has some beautiful properties for anybody wanting to understand how a machine performs calculations. The algorithm for performing multiplication or division is entirely transparent, because you have to perform it manually.

### **INFO What is an Algorithm?**

An algorithm is a step-by-step set of instructions for how to perform a task. Repeated additions is an example of a simple algorithm for performing multiplication.

The way you operate a Curta is very similar to how you use an Abacus. In fact we will go through the operations of an Abacus as a way of explaining what the principles behind performing advance math operations on the Curta is.

## Multiplication with an Abacus

An Abacus is made up of several columns with colored beads on which you can move up and down. Each column has 10 beads although this can vary depending

---

<sup>2</sup>Z1 was an electromechanical programmable computer created by Konrad Zuse in 1938 in the midsts of WWII in Germany.

<sup>3</sup>The Curta handheld mechanical calculator was developed by Curt Herzstark while he was held in the Buchenwald concentration camp.



Figure 14: The Curta handheld mechanical calculator. Credit: Thomas Schanz

on he type of Abacus. This is to allow representing digits from 0 to 9 plus carry. The first column starting from the right are the ones. The second column are the tens, the third column the hundreds and so on.

That means if I want to represent the number 4023 on an Abacus, then I arrange the beads as shown in the illustration above. With this arrangement addition and subtraction is easy. If I want to add 12, I just pull down two extra beads in the ones column and one bead in the tens column.

However it is more interesting to look at how multiplication is done. We could do it as we did earlier in our `multiply` function. That would mean performing as many additions as the number you multiply with. That would quickly require moving a lot of beads.

There is however a much quicker method relying on doing the equivalent of shifts. This helps you understand why an ALU can perform multiplication quickly using just addition and shifts.

Our task is to perform the multiplication:

$$234 \times 12 = 2808$$

We start by splitting up our Abacus in two parts. I will use similar terminology as used on the Curta and CPUs.

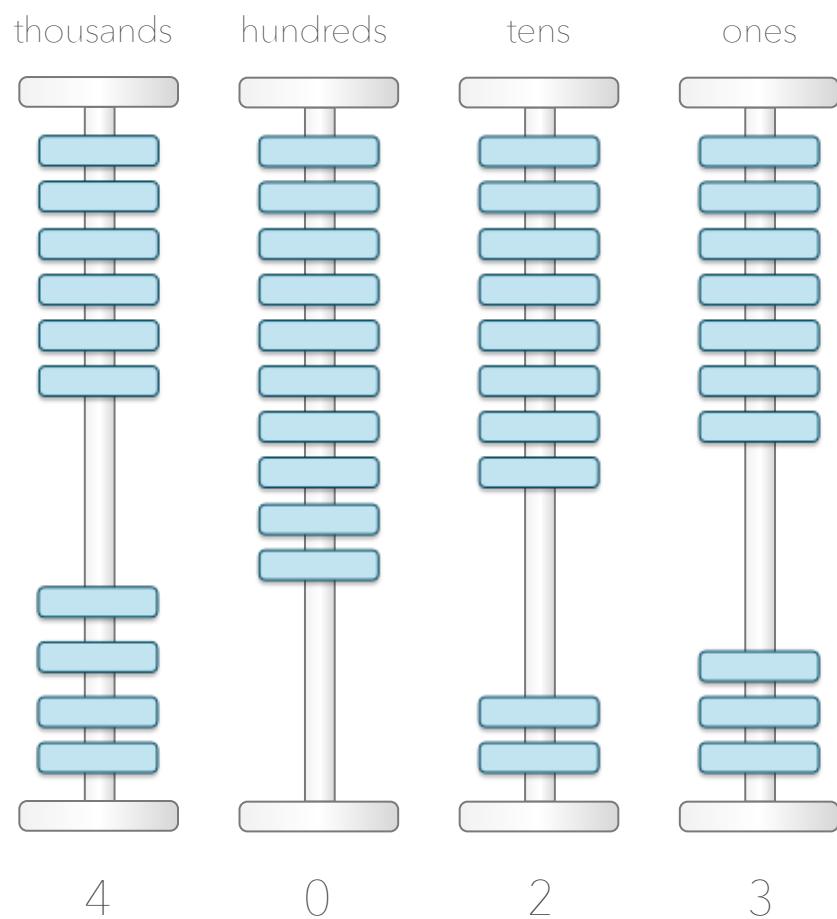


Figure 15: An illustration of a typical abacus. Each column represent a digit in the decimal system.

- On the **left** we have what I would call the **counter**. This is where we keep track of how many additions we have performed.
- On the **right** we have the **accumulator**. This is where we are accumulating the results of multiple additions or subtractions.

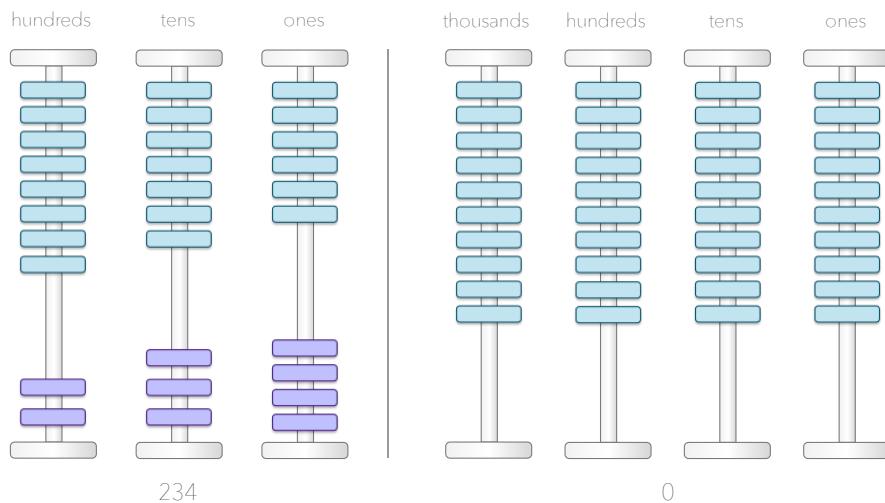


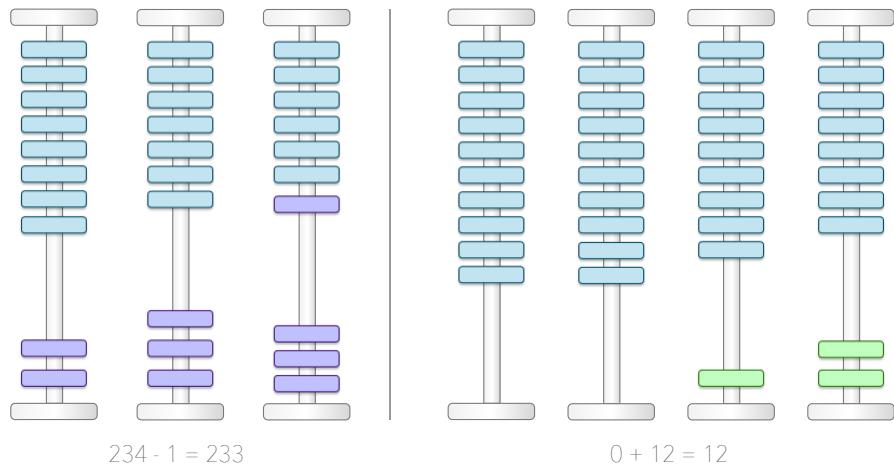
Figure 16: Performing  $234 \times 12$  using an Abacus. Right side is the *Counter* and on the left is the *Accumulator*.

On calculating machinery, we call the parts that hold numbers used in calculations for **registers**. In this case we could call the *counter* and the *accumulator* for registers. This terminology started with mechanical calculators and computers<sup>4</sup> but interestingly carried over to electronic computers as well.

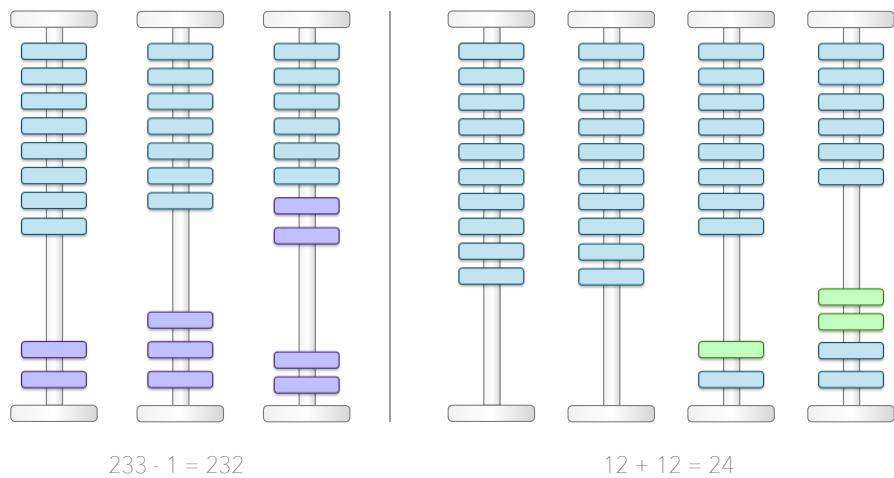
Thus in a modern day CPU, we also call the places inside the CPU holding numbers to perform calculations on for *registers*. Anyway, lets us begin performing the multiplication!

---

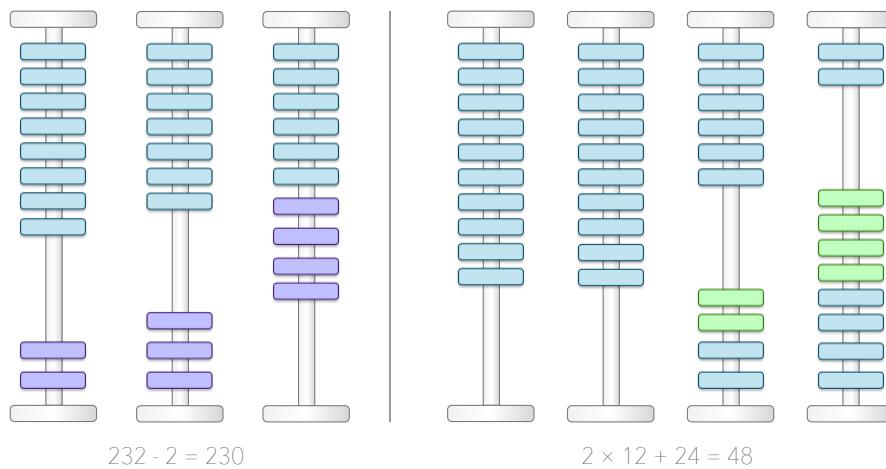
<sup>4</sup>Computer pioneer Charles Babbage, actually tried to create a fully mechanical and steam driven computer called the Analytical Engine in 1837.



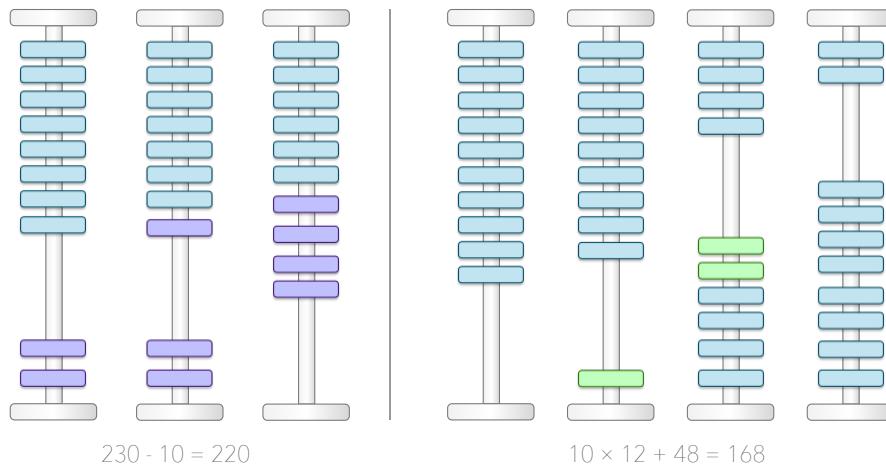
We move a bead up in the *ones* column of *counter* to indicate that we have added 12 to the *accumulator* once. Green indicates beads I have just moved in the current step. Let us make another move!



We add 12 a second time. The *counter* shows that we have added 12 twice to the *accumulator*. The green beads shows 1 ten-bead and 2 one-beads being added, representing 12 in total.

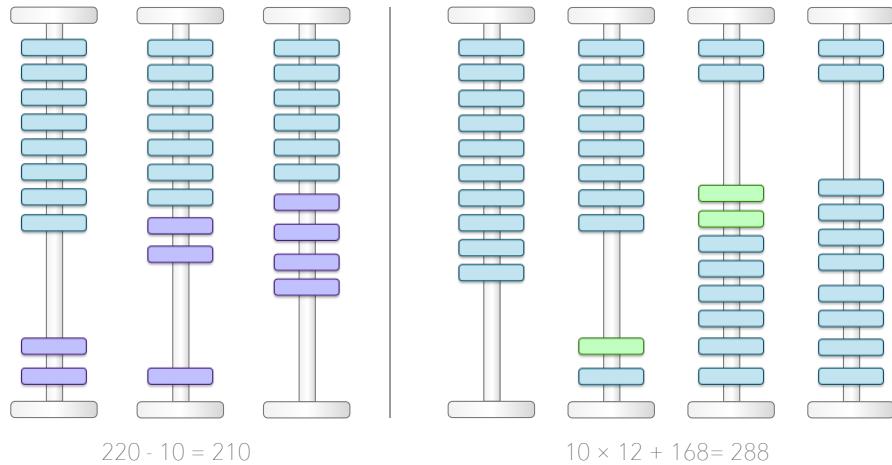


We speed up a little bit, and move two beads in the *counter*. That means we need to add  $2 \times 12 = 24$  to the accumulator.

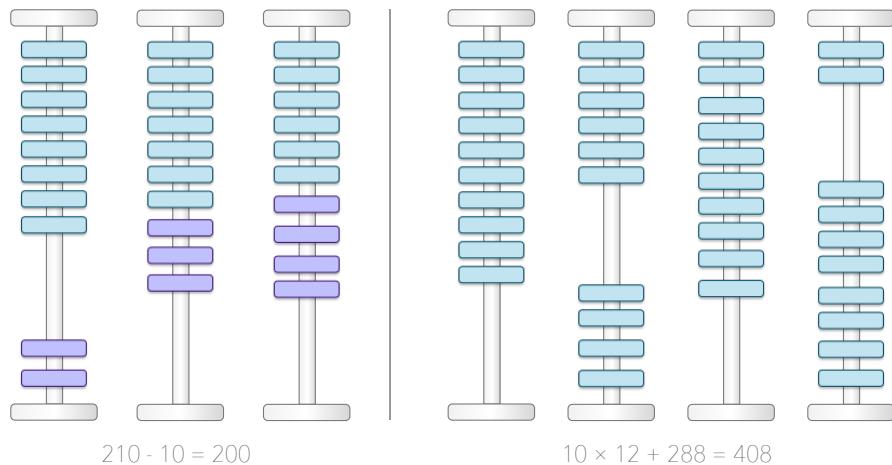


Now we get to the interesting part. All the *ones* in the counter have been spent. So we are on to counting the *tens*. We *shift* the adding of 12, one column to the left. This has the effect of adding  $10 \times 12$  instead of 12.

This allows us to add 12 ten times in a single operation. Much faster than our earlier naive for-loop approach to multiplication.



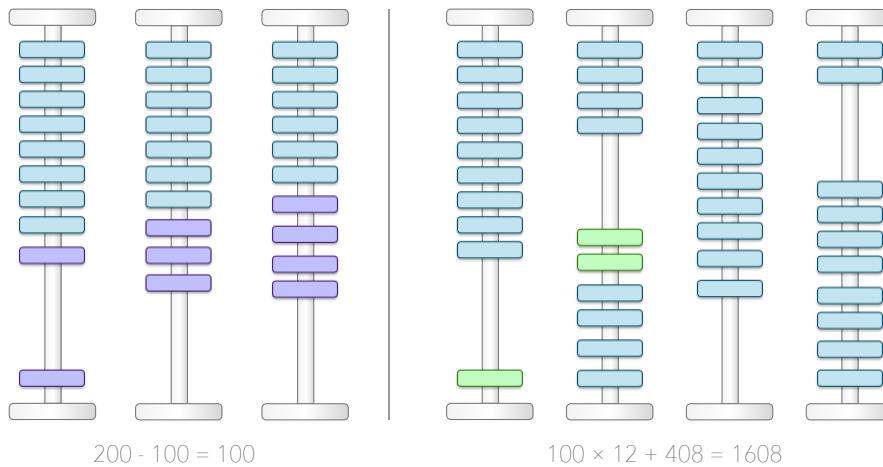
While we increase the accumulator with 120 upon each add, we increase the *counter* by 10 each time. You can read off the counter and see that we have added twelve, 24 times at this point.



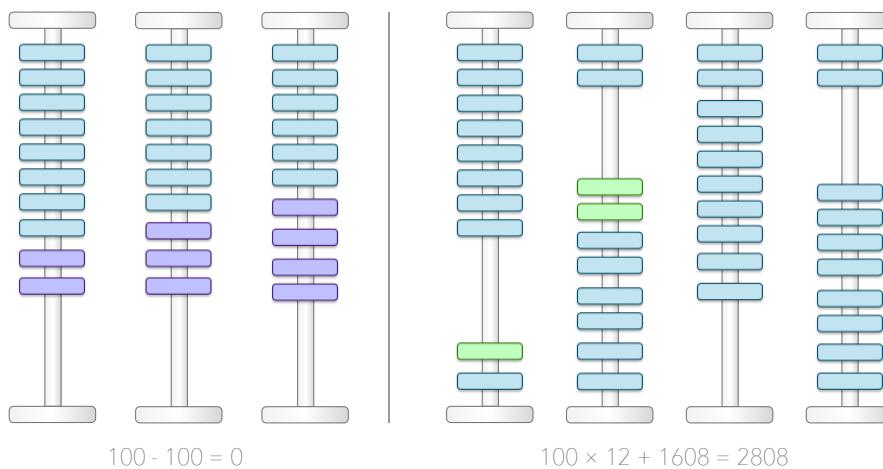
All the *ones* and the *tens* in the counter has been spent. By counting the purple beads which have been moved up, you can see that we have added twelve 34 times.

$$12 \times 34 = 408$$

The accumulator shows 408 as should be expected at this point. Since all the *tens* are spent we shift over the the purple beads in the *counter* representing *hundreds*.



Moving one of these beads up is the same as adding  $100 \times 12 = 1200$  each time. This is accomplished by shifting the twelve we add one more column to the left.



Moving the last bead in the *hundreds* column completes the calculation. The *accumulator* now shows 2808, which is the correct result given that:

$$234 \times 12 = 2808$$

What have we learned from this? That we can in fact perform multiplication with repeated additions a lot fewer times when we utilize the ability to perform shift operations. Multiplying the number you are adding with 10, 100, 1000, or any other multiple of 10 is easy.

### **Curta is an automated Abacus**

The Curta mechanical calculator is a mechanical automation of the exact same process.



Figure 17: Curta II mechanical calculator. Credit: Oldsoft

The Curta consists of the following relevant parts:

- **Input** register, which are the sliding pins in black and red at the bottom. You use these to specify a number you want to add or subtract from the *accumulator*.
- **Accumulator** or result register. These are the digits with a black background.
- **Counter** register. Digits with white background. Shows how many additions or subtractions have been performed.
- Each time the **Crank** is turned the *input* register is added or subtracted from the *accumulator* depending on the direction you turn it. The *counter* register is updated so you can keep track of how many turns you have made.
- **Barrel Shifter** lets you specify what multiple of 10 you want to multiply the input with before adding it to the *accumulator*.
- **Clearing Lever** is the ring sticking out which allows you to clear the *counter* and *accumulator* registers. Whatever register you slide it over will get cleared.

We can attempt to simulate the operations of this calculator in Julia code. To mimic the Curta behavior we can only allow our code to do the following operations:

1. Addition and subtraction.
2. Loops.
3. Shift operations. Multiply or divide by multiples of 10.

If you are interested in this Jan Meyer actually made an interactive Curta simulator you can tryout online.

## Curta Simulator in Action

Here is an example of how a simulation of the Curta in code could behave in the Julia REPL, when multiplying 12 with 234, as we just did with the Abacus.

```
julia> clear!()

julia> setinput!(12)
12

julia> crank!(4)
(48, 4)

julia> leftshift!()
1

julia> crank!(3)
(408, 34)

julia> leftshift!()
2
```

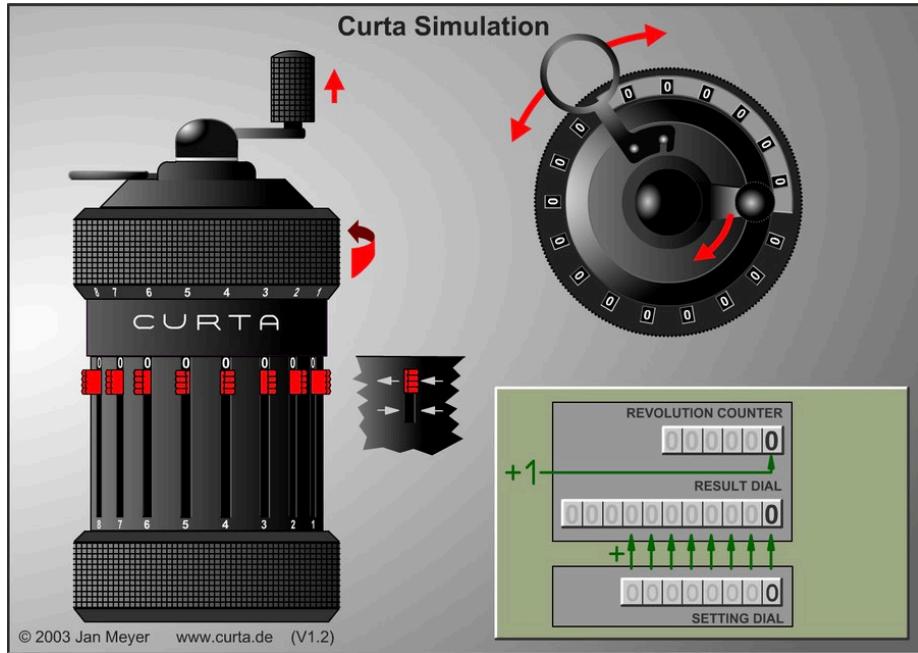


Figure 18: Jan Meyer's online Curta simulator

```
julia> crank!(2)
(2808, 234)
```

**NOTE Exclamation Mark Function Name Suffix**

Notice all the functions above have an exclamation mark (!) at the end of the name. This is not required. The code would have worked fine without. However this is a convention used by Julia programmers to tell everybody else using that function, that this function modifies its input or global data. It does not merely perform a calculation and return a result. For instance `crank!` modifies the accumulator and the counter.

1. We begin by clearing all the register and setting everything to default values with `clear!()`.
2. `setinput!(12)` sets our input register to 12. What we want to multiply with. In the Abacus sense, this is what we want to keep adding on each iteration.
3. `crank!(4)` turns the handle four times around. Meaning we add 12 four times. `crank!` returns the sum of all these additions and the total number of times the input has been added to the accumulator.
4. `leftshift!()`. We shift the adding of 12 one column over in the Abacus sense. That means we are now adding 120, rather than 12 each time we crank.
5. `crank!(3)`. We are working from the least significant digit in 234, to the

- most significant digit. So next digit is 3, which represents the tens. By turning the crank 3 times, we are adding twelve 30 times. The second number in the result, shows we have now added twelve a total of 34 times.
6. `leftshift!()` one more time means we are now adding the *hundreds*. The 2 returned indicates that we are multiplying with 100 because  $10^2 = 100$ .
  7. `crank!(2)`. We add twelve 200 times. That means we have now added 12 a total of 234 times, which gives the final result of 2808.

## Implementation of a Curta Simulator

Because we are writing multiple related functions in this case relying on shared data, it is smart to put this code in a file. I am putting the code in a file named `curta.jl`. However you can name the file whatever you like.

### Defining Registers

In the top of the file we put the following code:

```
input = 0
accumulator = 0
counter = 0
shifter = 0
```

This is to define the registers that the Curta works with. These are simple variables that store the current values in the registers.

The `shifter` contains a number for how many positions the barrel shifter has been moved.

Each time we do a new calculation, we want to reset the Curta by clearing all the registers.

```
function clear!()
    global input = 0
    global accumulator = 0
    global counter = 0
    global shifter = 0
    nothing
end
```

Notice the use of the keyword `global`. If we had written simply `input = 0` then Julia would have interpreted `input` as a local variable only visible inside the `clear()` function.

In most programming languages the same variable or function name can refer to different variables or functions depending on the *scope* it is defined and used. Variables defined outside of any function are part of the the *global scope*. Variables defined inside a function exist in a scope *local* to that function.

**NOTE What is a Scope?**

A simple way to think about **scope** is to use the analogy of a filesystem with folders and files. You can have two different files with say the name `foobar` if they exist in different directories. Files only have to be uniquely named within one directory. Julia is the same: Variable names only need to be unique inside a function.

### Setting and Modifying Registers

Before performing calculations we need to be able to set the `input` register. We could let people set it directly, but it is a good habit to only modify (mutate) a system through function calls.

```
function setinput!(x)
    global input = x
end

function leftshift!()
    global shifter += 1
end

function rightshift!()
    global shifter -= 1
end
```

`leftshift!` and `rightshift!` are used to simulate the twisting of the barrel shifter to affect what multiple of 10 the input will be multiplied by before being added to the accumulator.

### Performing Addition and Subtraction

The actual addition and subtraction with shifting is done with the `crank!` and `uncrank!` functions. `crank!` performs additions, while `uncrank!` is what happens when you pull the Curta crank up a notch and turn it around. This causes a subtraction.

```
function crank!(n)
    for _ in 1:n
        global accumulator += input * 10^shifter
        global counter += 10^shifter
    end

    accumulator, counter
end

function uncrank!(n)
    for _ in 1:n
        global accumulator -= input * 10^shifter
        global counter -= 10^shifter
    end
```

```
    accumulator, counter
end
```

We can see we simulate turning the crank  $n$  times with for-loops iterating over the range  $1:n$ .

$10^{\text{shifter}}$  gives us different multiples of 10. The  $+=$  operator means to add to the existing value of a variable. Thus  $x += 2$  is equal to  $x = x + 2$ .

I bet you have a few questions about scoping when you look at the use of `global` in front of `accumulator` and `counter`. This tells Julia that both these variables come from the global scope. Julia does not need to bother looking for their definition inside the `crank!` function.

But wait a minute! What about `input` and `shifter`? Are they not in the global scope as well? Do we not need to tell Julia about that somehow?

The reason we don't have to tell Julia that `input` and `shifter` is global is because we are only *reading* values from these variables, we are not *writing* to them. Global variables could be used by multiple functions, so writing to them by accident can screw up the behavior of other functions in unpredictable ways.

Julia is trying to protect us from doing that by forcing us to use the `global` keyword when we want to *write* to a variable. That forces you to be explicit about modifying global variables.

## Efficient Multiplication Algorithm

Our Curta simulator basically simulates how you interact with the Curta, but we can extract the principles in how the Curta operates to create a more efficient generic multiplication algorithm than the naive one we used earlier.

```
function multiply(input, counter)
    accumulator = 0
    while counter > 0
        count = rem(counter, 10)
        counter = div(counter, 10)
        for i in 1:count
            accumulator += input
        end
        input *= 10
    end
    accumulator
end
```

The way this works is very similar to our earlier `multiply` function except it uses the Curta algorithm for multiplication. With this approach the time it takes to multiply is proportional to the number of digits in the counter rather than its value. If we consider an average 3-digit number such as 555, then it will require  $5 + 5 + 5 = 15$  additions. Compare that with our naive multiplication method which would have required  $555/15 = 37$  times as many additions.

Keep in mind this is not efficient to run on your computer. We are merely simulating efficient decimal multiplication (base 10).

Your computer does not use a decimal number system, so there is no efficient way of performing decimal shifts on your computer, the way you can efficiently do that on a Curta or Abacus.

Thus instead of shifting left and right, we use multiplication and division of 10, which on a binary digital computer is as slow as dividing and multiplying any other number.

### **Integer Division**

Both in our Abacus example and with the Curta, we are dealing with single digits at a time in the *counter*. To be able to accomplish this here we need to chop off one digit at a time with the `div` and `rem` functions.

These are functions for integer division in Julia. `div` gives the quotient and `rem` gives the remainder. These can also be written as the `÷` and `%` operators. Here are examples of usage:

```
julia> div(234, 10)
23

julia> rem(234, 10)
4

julia> 234 ÷ 10
23

julia> 234 % 10
4

julia> 23 ÷ 10
2
```

Special symbols such as `÷` can be written quickly in the Julia REPL by using backlash (`\`) and the tab key. In the REPL if you write `\div` and hit tab it will complete to `÷`.

### **Lookup Help on Functions**

As you learn more functions in Julia it becomes cumbersome to skip through this book looking for a description of each function we have covered. A more practical solution is to use the built-in help system of the Julia REPL.

For instance say you don't remember the difference between writing:

```
23 ÷ 10
23 / 10
```

You can then write a question mark `?` at the Julia prompt like this.

```
julia> ?
```

This will change the Julia prompt to:

```
help>
```

To indicate you have switched to help mode. You can switch back to Julia mode by hitting *backspace*. Mode switching is caused by the first character written on a prompt. Say a user has written:

```
julia> 23 ÷ 10
```

They can then hit *Ctrl+A* to move the cursor to the beginning of the line (*Ctrl+E* move to end of line) and write ?. Hit enter and you get help describing what ÷ does when used with two integer numbers.

```
julia> 23 ÷ 10
julia> ? 23 ÷ 10
help> 23 ÷ 10
help?> 23 ÷ 10
    div(x, y)
    ÷(x, y)
```

The quotient from Euclidean division. Computes x/y, truncated to an integer.

#### Examples

---

```
julia> 9 ÷ 4
2
```

```
julia> -5 ÷ 3
-1
```

We can lookup how floating-point division works.

```
help?> 23 / 10
/(x, y)
```

Right division operator: multiplication of x by the inverse of y on the right. Gives floating-point results for integer arguments.

#### Examples

---

```
julia> 1/2
0.5
```

```
julia> 4/2
2.0
```

```
julia> 4.5/2
2.25
```

## Binary Number System and Multiplication

While Julia gives us the illusion of working with decimal numbers, under the hood the computer works with a binary number system.

You only need to understand decimal numbers to understand how an Abacus and a Curta works, but to understand how the equivalent part in digital computer, the Arithmetic Logical Unit (ALU) works, you need to have a grasp of binary numbers.

At first it can be hard to wrap your head around other number systems because we are so accustomed to using the decimal system. But there is nothing particular natural about decimal numbers. They are based on ten because we got ten fingers on our hands.

### Counting for Dogs and Octopus

Thus it was practical for humans all through time to count up to ten. Now imagine if you were a dog instead? You may count with paws instead. Thus you would count up to four. The counting may have worked like this:

I paw, 2 paws, 3 paws,  
I dog, I dog and I paw, I dog and 2 paws etc.

Thus the decimal number 7, would be represented by: 1 dog and 3 paws because  $1 \times 4 + 3 = 7$ .

The decimal number 9 would be represented as: 2 dogs and 1 paw because  $2 \times 4 + 1 = 9$ . For a dog only familiar with this number system it would be natural to write these numbers as 13 and 21. Thus 13 in base four equals 9 in base ten.

An octopus in contrast may have base eight number system given that they have eight tentacles. 1 octopus and 5 tentacles would by an octopus be written as 15, but would in our decimal system correspond to  $1 \times 8 + 5 = 13$ .

Our poor digital computer however can only count to two. It has no fingers or paws, but rather relies on distinguishing between high and low electric voltage, although binary numbers can be represented in numerous ways. Thus internally a computer does not use our ten different digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Rather a digital computer only use two measly digits 1 and 0.

Depending on circumstances these may also be referred to as *true* and *false* or *high* and *low*. The benefit of the binary number system is that it can be handled with what is called *boolean logic* and arithmetic with binary numbers is very easy.

E.g. here is the whole multiplication table for binary numbers:

$$\begin{aligned}0 \times 0 &= 0 \\0 \times 1 &= 0 \\1 \times 0 &= 0 \\1 \times 1 &= 1\end{aligned}$$

Likewise adding binary numbers is simple.

$$\begin{aligned}0 + 0 &= 00 \\0 + 1 &= 01 \\1 + 0 &= 01 \\1 + 1 &= 10\end{aligned}$$

We just apply those simple rules over and over again for larger numbers:

$$\begin{array}{r} 1011 \\ + 0001 \\ \hline = 1100 \end{array}$$

While binary numbers are cumbersome for us humans to work with given the large number of digits required to represent modestly large numbers, they have the benefit of radically simplifying the design of the hardware performing addition and subtraction. That is why Konrad Zuse's Z1, which was basically a mechanical computer, would fit inside a regular living room while Charles Babbage's Analytical Engine (decimal based) would have taken up a whole factory hall.

### Hexadecimal and Octal Number System

Because it is very cumbersome to write binary numbers, computer scientists invented two alternative number systems called Hexadecimal and Octal. Hexadecimal is a base 16 number system while Octal is a base 8 system.

The octal system is easy to deal with. You just use digits from 0 to 8. Essentially it is like the Octopus system discussed earlier. Thus nineteen would be written as 23, because  $8 \times 2 + 3 = 19$ .

Hexadecimal is harder, because we need more digits. The solution computer scientists came up with was to use the letters from A to F as digits. Thus fifteen is written as F in hexadecimal while sixteen is written 10. To avoid confusing hexadecimal numbers with decimal numbers a prefix or suffix is usually used. Hence sixteen would be written as 0x10 or 10h. In Julia Hexadecimal numbers are prefixed with 0x while octal numbers are prefixed with 0o. Binary numbers are prefixed with 0b.

```
0010 == 10
0b10 == 2
0o10 == 8
0x10 == 16

0x0B == 11
```

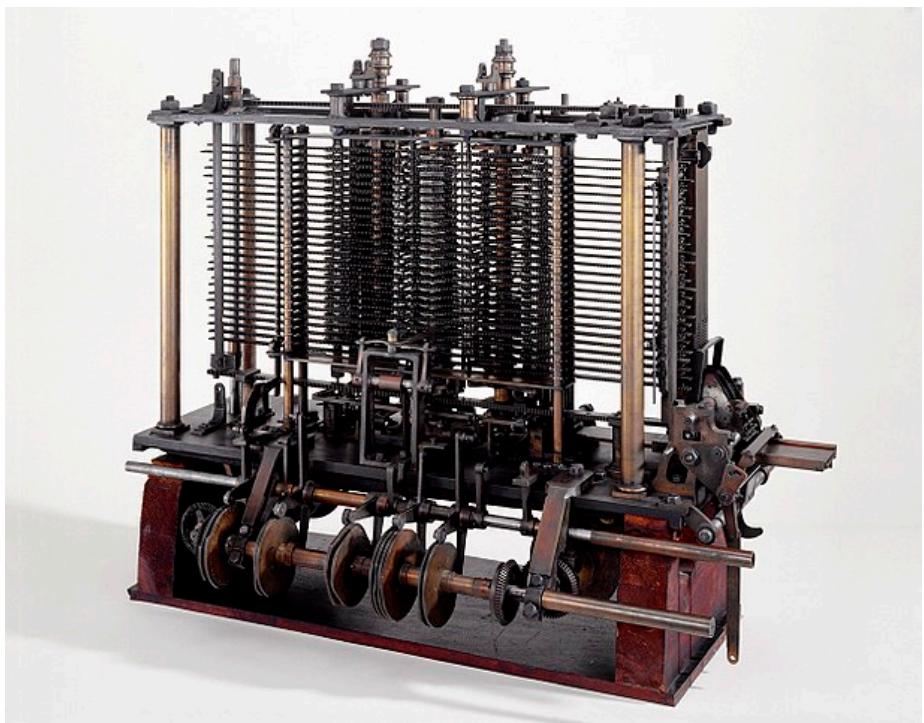


Figure 19: Small section of Charles Babbages' Analytical Engine. Credit: Science Museum London

```
0b11 == 3
0o23 == 19
```

### Integer Types and Bit Sizes

When we worked with the Abacus and the Curta you saw that there was limited space to represent digits. We could not work with numbers of arbitrary size. It is the same with numbers on a computer. For binary numbers we refer to each digit as a bit. An 8-bit number is called a byte. Example of an 8-bit number:

```
0b1011_0001
```

The bit sizes of numbers computers work with are not arbitrary. No modern computer works with 3, 5 or 21 bit numbers e.g.

**NOTE Using \_ in long numbers**

Notice the usage of underscore `_` to make numbers with many digits easier to read. You can use this in any number in Julia not just when using binary notation.

Rather the sizes are all multiples of two: 8, 16, 32, 64 and 128. These numbers take 1, 2, 4, 8 and 16 bytes of storage space respectively.

In Julia an 8-bit integer is called a `Int8` or `UInt8` while e.g. a 64-bit integer is called `Int64` or `UInt8`. The U stands for “Unsigned”, which implies that both `Int8` and `Int64` are signed integers. We can discover the type of an object with the `typeof` function.

Examples of creating integers of different bit-lengths in Julia:

```
julia> x = Int8(42)
42

julia> y = UInt8(200)
0xc8

julia> z = 3451
3451

julia> typeof(z)
Int64

julia> typeof(x)
Int8
```

Signed numbers keep track of whether they are positive or negative, while unsigned numbers simply assume they are always positive. Although it is important to realize that a computer cannot store a number sign in memory. It can only store bits which are either 0 or 1.

In memory a signed byte can look identical to an unsigned one, but entirely different to the user. Why is that? Because the difference is not in how they

are stored but how they are used and presented to the user. For instance the byte `1000 0010` interpreted as a `UInt8` will be seen as the value `130`. While interpreted as a `Int8` it will have the value `-126`.

We can explore the minimum and maximum values for a specific Julia type with the `typemin` and `typemax` functions:

```
julia> Int(typemax(UInt8))
255
```

```
julia> Int(typemin(UInt8))
0
```

```
julia> typemax(Int8)
127
```

```
julia> typemin(Int8)
-128
```

A signed and an unsigned integer with the same number of bits will have a different range of valid values. A signed 8-bit integer e.g. cannot go higher than `127`, because it has to reserve a bit to represent negative numbers. Thus an unsigned 8-bit integer can go all the way up to `255`.

If you don't specify the type of an integer in Julia, it will be a 64-bit signed integer. You can calculate the max values of integers based on the bit size yourself.

The max value of an unsigned integer of  $n$  bits is  $2^n - 1$ , while the the max value of a signed integer is  $2^{n-1} - 1$ . We can verify this in the REPL.

```
julia> typemax(Int8)
127
```

```
julia> 2^(8-1) - 1
127
```

```
julia> typemax(UInt8)
0xff
```

```
julia> 2^8 - 1
255
```

```
julia> typemax(Int64)
9223372036854775807
```

```
julia> 2^(64-1) - 1
9223372036854775807
```

This is completely analogous with the decimal system. The max value of a decimal number with  $n$  digits is  $10^n - 1$ . Let us calculate the max value of a 3-digit decimal number as an example. According to this equation it should be  $10^3 - 1 = 999$ , which is obviously correct.

## Simulating an Arithmetic Logic Unit (ALU)

The Curta prepares us for looking at how the equivalent kind of hardware in a CPU works.

The Curta works with decimal numbers while the ALU of a modern computer works with binary numbers. We must change the multiplication code to reflect how binary numbers are different from decimal numbers.

```
function binary_multiply(input, counter)
    accumulator = 0
    while counter > 0
        if counter & 1 == 1
            accumulator += input
        end
        counter >>= 1
        input <<= 1
    end
    accumulator
end
```

There are a number of new things in this code such as `counter & 1` and `counter >>= 1` which we have not covered earlier. The `&`, `>>` and `<<` operators are specific to binary numbers.

Thus we will have to leave this code and take a detour discussing binary operators, before we return to this code example later.

## Binary Shift Operators

Let us use the Julia REPL to explore how some of these new operators used here work. The `>>` and `<<` operators are binary shift operators.

```
julia> 0b01100 >> 1 == 0b00110
true
```

To make it easier explore how this works, it is practical to use the `bitstring` function which takes a number and produce a string which looks like the binary number representation of that number.

If we don't use that, it will be hard to read the results because Julia signed integers are always shown as decimal numbers while unsigned are shown in hexadecimal notation.

```
julia> 0b01100
0x0c

julia> bitstring(0b01100)
"00001100"
```

So by wrapping every shift operation in `bitstring` we can easily see how it works:

```
julia> bitstring(0b01100 >> 2)
```

```
"00000011"

julia> bitstring(0b01100 << 1)
"00011000"

julia> x = 0b0001;
julia> x <= 3;

julia> bitstring(x)
"00001000"

julia> x >= 2;

julia> bitstring(x)
"00000010"
```

You can see these operators are able to move the bits left or right as many places as you want. So `x >> n` moves the bits in the number `x`, `n` places to the right, while `x << n` would move the bits in `x`, `n` places to the left.

## Bitwise Boolean Operators

While the basic buildings blocks of a Curta are shafts and gears, modern digital electronics used to build the ALU is all composed of logical gates. These gates perform what we call boolean logic which provide the foundation for performing binary arithmetic.

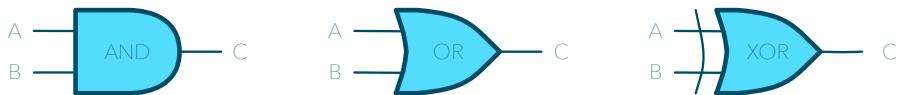


Figure 20: Basic logical gates, used to perform boolean logic.

Above are some of the more common logical gates: **AND**, **OR** and **XOR**. They work by taking two electric signals on inputs marked by A and B and producing an output signal C. The inputs and outputs are binary. They are either a low or high voltage<sup>5</sup>, representing a 0 or 1.

### AND

You can perform the bitwise AND operation in Julia with the `&` operator. AND needs a 1 signal on both A and B to give a 1 on the output C. In all other cases it will output a 0. Alternatively you could think about it as both A and B need to be true for the output to be true. Julia can apply the `&` operator to multiple bits in one go.

---

<sup>5</sup>The voltage levels used for 0 and 1 depends on the integrated circuits used. The 7400-series integrated circuits have 0 - 0.8 Volt to represent 0, while 2.7 - 5.0 Volt represent 1. This is referred to as TTL Logic levels.

```
julia> 1 & 0
0

julia> 0 & 1
0

julia> 1 & 1
1

julia> 0b0101 & 0b1100 == 0b0100
true

julia> 0b0101 & 0b1111 == 0b0101
true
```

**OR**

Bitwise OR is performed with the | operator in Julia. A logical OR gate will give a 1 as output if either input or both are 1. Here is an example of how that works in Julia.

```
julia> 1 | 0
1

julia> 0 | 0
0

julia> 1 | 1
1

julia> 0b1100 | 0b0011 == 0b1111
true
```

**XOR**

Exclusive OR only gives 1 as output if either input is 1. If both are 1, it will output 0.

```
julia> xor(1, 1)
0

julia> xor(1, 0)
1

julia> xor(0, 1)
1
```

Logical gates have a lot of uses in electronics. But in our case they are interesting to know about because many of these gates are used to create an ALU. Remember a big part of what an ALU does is adding.

To wire up an adder in electronics is actually not that hard. You can add two bits A and B with a *Half Adder* as shown below.

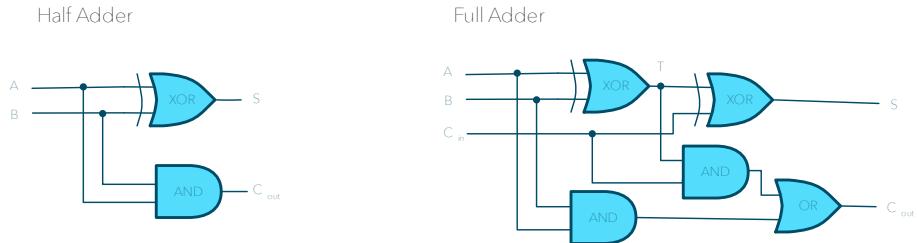


Figure 21: Combining logic gates to allow adding of bits (binary digits).

You get the sum out on S.  $C_{out}$  is the carry, which should be transferred to the next digit. To handle adding up two numbers composed of multiple bits we need a *Full Adder*, shown to the right.

You need one for each bit. The *Full Adder* takes the carry from the addition of the previous bit and include it in its addition. If it produces a carry, that will be transferred to the next *Full Adder* handling the next bit.

We can simulate these adders in code using bitwise boolean operators. Here is a *Half Adder*.

```
function half_adder(A, B)
    S = xor(A, B)
    Cout = A & B
    Cout, S
end
```

But to do more interesting work we need a *Full Adder*.

```
function full_adder(A, B, Cin)
    T = xor(A, B)
    S = xor(Cin, T)
    Cout = (A & B) | (Cin & T)
    Cout, S
end
```

Once we got multiple 1-bit adders we can combine them to create a 4-bit adder. 4-bits are referred to as a nibble<sup>6</sup> among old-school programmers. Here is a nibble adder:

```
function nibble_adder(A4, A3, A2, A1, B4, B3, B2, B1)
    carry, S1 = half_adder(A1, B1)
    carry, S2 = full_adder(A2, B2, carry)
    carry, S3 = full_adder(A3, B3, carry)
    carry, S4 = full_adder(A4, B4, carry)

    S4, S3, S2, S1
end
```

---

<sup>6</sup>A nibble is a word play on “half a byte”, which sound like “half a bite”.

We can try it out in the Julia REPL. Lines starting with # are just comments and ignored by Julia.

```
# 6 + 5 == 11
julia> nibble_adder(0,1,1,0, 0,1,0,1)
(1, 0, 1, 1)

julia> nibble_adder(0,0,0,1, 0,0,0,1)
(0, 0, 1, 0)

# 1 + 3 == 4
julia> nibble_adder(0,0,0,1, 0,0,1,1)
(0, 1, 0, 0)
```

Electronics is much the same as code. We use simple functions to construct more complex functions. These complex functions can be combined to create even more complex functions.

Likewise transistors can be packages together to create logical gates. Logical gates can be packaged together to create half or full adders. These can further be combined to create nibble adders, which can be used to create adders for 8-bit numbers.

These adders are then combined with other logical gates to create things like the Arithmetic Logical Unit (ALU) of a CPU.

## Understanding Binary Multiplication

Now we know more of the detail of the bitwise operations used when performing binary multiplication. Time to revisit our `binary_multiply` function. Let us look closer at the behavior of the while-loop.

```
while counter > 0
    if counter & 1 == 1
        accumulator += input
    end
    counter >>= 1
    input <<= 1
end
```

To check the value of the rightmost bit we use `counter & 1 == 1`. This is also referred to as the *least significant bit (LSB)*, because it contributes the least to the total value of the number. `counter & 1` is the same as `counter & 0b00001` or however many zeros you add, which means the boolean expression only evaluates to 1, if and only if the LSB of `counter` is 1 as well.

Thus together with `counter >>= 1`, we have a method of chopping off one bit at a time from `counter`. Since multiplication is either with 1 or 0, we handle it with an if-statement which either adds the `input` to the `accumulator` or doesn't.

This is analogous to how the Curta simulator worked. Remember the  $234 \times 12$  example? The `counter` was set to `234` and the `input` to `12`. Instead of using binary shift to get digits we would use `rem` and `div`.

We would pull off the 4 and then use that to add twelve to the accumulator four times. Then we would get the next digit 3, from the counter and use it to add 120 three times.

If we look at this at a binary level, then 234 corresponds to `0b1110_1010` in binary, while 12 is `0b0000_1100`. What `binary_multiply` does is basically to carry out the calculation you see below.

$$\begin{array}{r}
 00001100 \\
 \times 11101010 \\
 \hline
 0000 \\
 1100 \\
 0000 \\
 1100 \\
 0000 \\
 1100 \\
 1100 \\
 1100 \\
 \hline
 10101111000
 \end{array}$$

It is performed by multiplying every bit in the bottom number (234) with the number on the top (12). Since bit is either 0 or 1. The number added will be either `0b0000` or `0b1100`. Every time we add this number it has to be shifted one step to the left, as we are multiplying with a more significant bit.

Written in decimal form, the calculation we are carrying out is:

Calculation	Total
$12 \times 0 \times 1$	0
$12 \times 1 \times 2$	24
$12 \times 0 \times 4$	0
$12 \times 1 \times 8$	96
$12 \times 0 \times 16$	0
$12 \times 1 \times 32$	384
$12 \times 1 \times 64$	768
$12 \times 1 \times 128$	1536
Sum	2808

This is quite efficient since instead of the naive method which would require adding 12, 234 times, we are only making 5 additions.

## Summary

You don't need to know how to simulate a Curta or what an ALU does to write Julia programs. The intention in this chapter was *not* to make you memorize lots of technical details, but to develop a sense or intuition of what numbers on a computer actually are.

Julia has a wide variety of operators to deal with integer numbers such as shift operators and boolean operators. It is hard to grasp what these actually do or their purpose without seeing some examples of how binary numbers are used.

Unless you work with microcontrollers or systems programming you will likely never need to use operators such as `>>`, `<<`, `&`, `|` and `xor` again.

However `div`, `rem` and their symbol equivalents `÷` and `%` are very useful to know, in everyday Julia programming.

We covered multiplication in this chapter in detail, but understanding multiplication in detail like this, is not necessary to be a Julia programmer.

The point of delving into the details of multiplication was to convey how even the most basic operations on a computer are made up of even more basic operations. The whole system is built layer by layer, where each layer looks more capable and sophisticated than the layer below it.



# More on Types

- **Number hierarchy.** Numbers in Julia are organized in a hierarchy.
- **Concrete types.** All values are of a concrete type.
- **Abstract types.** To represent several concrete types with shared traits.
- **Multiple dispatch.** Determining what code to run based on the type of all the function arguments.

We have already worked with values of different types such as text strings, dictionaries, ranges and numbers. However we have not talked about what a type is or how they are organized in Julia. Every value in Julia has a type. The type determines what you can do with a value. The kind of things you can do with a text string is different from the kind of things you can do with a number. Where it gets interesting is that types are often subdivided into more specific or concrete types. A number is not just a number. A whole number supports a different set of operations from an irrational number.

Thus types form elaborate hierarchies often expressed as trees, growing downwards. At the root, or top we find more generic types and at the bottom there are more concrete types. To better understand what this means, we will start this chapter by looking more closely at the *number hierarchy* in Julia.

## Number Hierarchy

Above you can see all the different types of numbers in Julia visualized in a hierarchy forming a tree. At the root of the tree you find the `Number` type. At the leaves of the tree you will find number types such as `Float32`, `Int64` and `Irrational`.

As you move towards the tree leaves, the types become more concrete. While the types become more abstract as you move towards root of the tree. You can discover this type hierarchy yourself in the Julia REPL. Let us do some experiments to learn more about the type system.

```
julia> x = 42
```

```
42
```

```
julia> y = 4.2
```

```
4.2
```

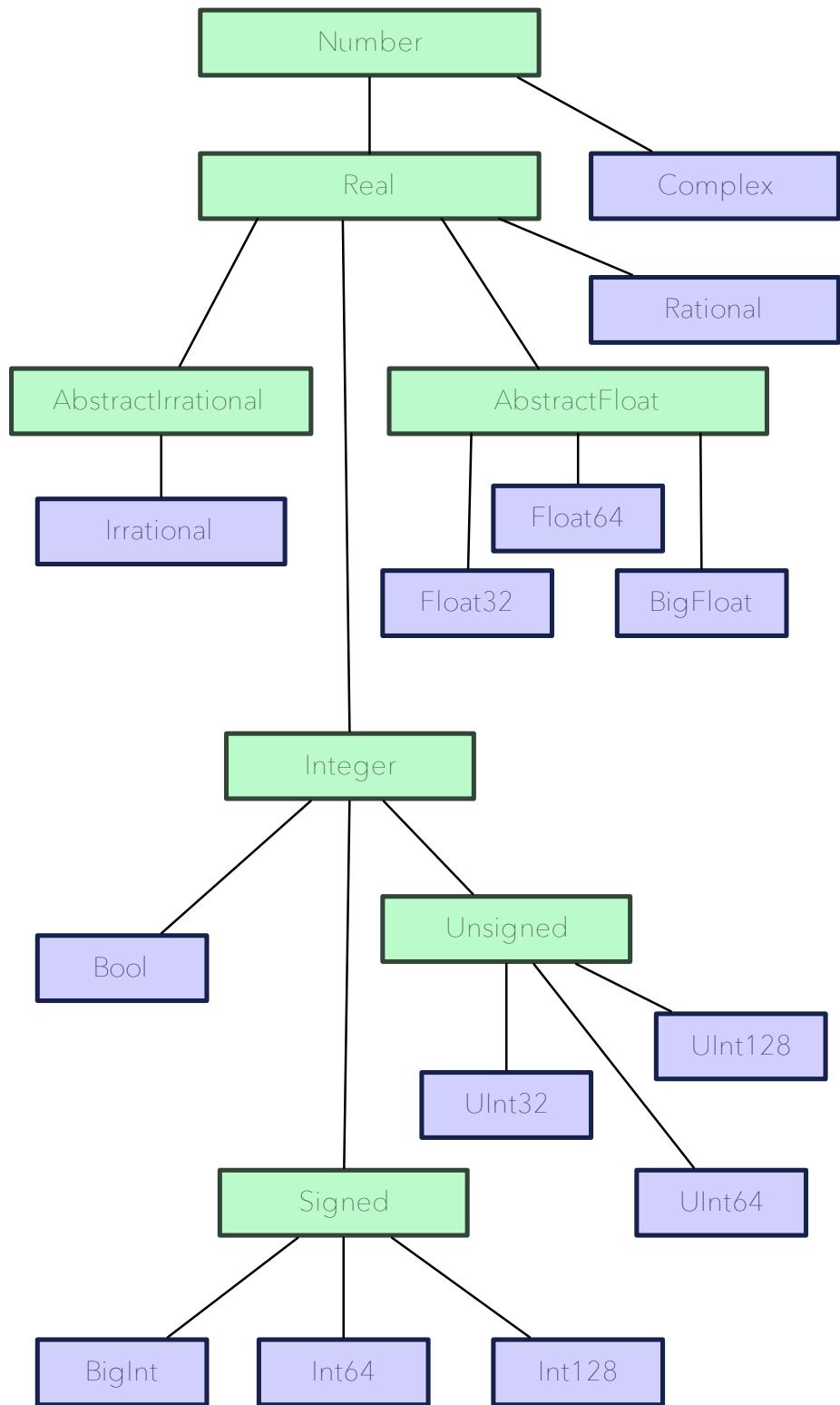


Figure 22: Type hierarchy for numbers

We put two numbers in the variables `x` and `y`. Using the `typeof` function we can find out what their type is:

```
julia> typeof(x)
Int64
```

```
julia> typeof(y)
Float64
```

Of course we don't have to stick the numbers in a variable first to discover its type. You can simply write:

```
julia> typeof(42)
Int64
```

This is a concrete type. We can make values with concrete types. They exists as bits (ones and zeroes) in your computer's memory. However we can move up the type hierarchy using the `supertype` function. It allows us to ask what the super type of another type is.

```
julia> supertype(Int64)
Signed
```

```
julia> supertype(Signed)
Integer
```

```
julia> supertype(Integer)
Real
```

We can continue doing this until we get to `Number`. Every super type is a further abstraction of the type below. We can also go the opposite direction and ask what the subtypes of a number types is:

```
julia> subtypes(Integer)
3-element Array{Any,1}:
 Bool
 Signed
 Unsigned
```

```
julia> subtypes(Signed)
6-element Array{Any,1}:
 BigInt
 Int128
 Int16
 Int32
 Int64
 Int8
```

Subtypes are specializations of an abstract type. Let us look at the most important abstract number types and how they are subdivided into more concrete types.

## Integers

These are the whole numbers  $\mathbb{Z}$ , you typically use for counting, such as:

-2      -1      0      1      2      3

In Julia these are further subdivided into `Signed` and `Unsigned` integers. Unsigned integers are the numbers 0, 1, 2, 3 etc, while signed integers include the negative numbers such as -5, -4, -3 etc. Thus far we have only talked about abstract numbers.

However signed and unsigned integers are further subdivided into integer types of different bit sizes. This is *not* something a mathematician would do. In mathematics it is irrelevant how many bits (binary digits) are used to represent a particular kind of number. However a computer operates with limited resources. Memory is a limited resource and so the bit size matters. `Int16` is a 16 bit signed integer. A byte in a computer is made up of 8 bits. That means a 16 bit integer consumes 2 bytes in computer memory. An `Int8` number is 8 bits, so it only consumes 1 byte. While a `Int64` integer consumes 8 bytes. This sounds very small when modern computers have giga bytes of memory. However once you start using lots of numbers this will start to matter.

What bit size you use, depends on what you are doing. For instance if you want to store number of children in a variable, an 8 bit unsigned integer `UInt8` would work fine. It can store number from 0 to 255. Nobody has more than 255 children, nor do they have negative number of children.

However for most tasks the default Julia integer, `Int64` works fine. Most of your programs will not use so many numbers that you have to consider their bit size.

If you are uncertain of whether a number type supports large enough or small enough numbers you can use `typemax` and `typemin` to find out.

```
julia> typemax(Int8)
```

```
127
```

```
julia> typemin(Int8)
```

```
-128
```

```
julia> typemax(Int64)
```

```
9223372036854775807
```

If you think it is hard to read the number of decimal digits in this number there is a trick you can use.

```
julia> string(typemax(Int64))
```

```
"9223372036854775807"
```

```
julia> length(string(typemax(Int64)))
```

```
19
```

We are converting the number into a text string. Then we can count digits by simply counting the characters in the string.

## Floating Point Numbers

Examples of floating point numbers are:

3.38    0.5    2.3e6    7.2f0    34.67f0

These numbers look a lot like a decimal number such as 4.52, where the whole number part (4) is separate from the fractional part (52).

However a computer cannot accurately represent arbitrary decimal numbers. Thus instead we have floating point numbers in most programming languages. These are inaccurate approximations to decimal numbers. That means when working with floating point numbers you have to assume there are minor rounding errors. The accuracy of the floating point numbers depends on the bit size of the concrete floating point type you use. `AbstractFloat` is just an abstract floating point number type. Actual floating point numbers will be e.g. `Float32` or `Float64`. Which one you choose will depend on how accurate numbers you want and how much memory you want to waste.

## Rational Numbers

Any number which can be written as a fraction, is a rational number.

$$12 \quad \frac{1}{2} \quad -5 \quad \frac{3}{4} \quad \frac{2}{3}$$

However in Julia the definition is a bit more specific. Numbers written like this `1//2`, `3//4`, `32//64` etc are rational numbers.

## Irrational Numbers

These are numbers which you cannot write as a fraction. For instance there is no way to express  $\pi$  or  $e$ , Euler's number as a fraction. We don't have to get into the details here, but a number such as  $\pi$  has infinite number of digits.

## Array and Range Hierarchy

Arrays and ranges are also part of an elaborate hierarchy. Notice below that range is an `AbstractArray`. This explains why they can often be used interchangeably.

You can explore this type hierarchy yourself.

```
julia> typeof(2:4)
UnitRange{Int64}

julia> typeof(3:2:9)
StepRange{Int64, Int64}

julia> supertype(typeof(3:2:9))
```

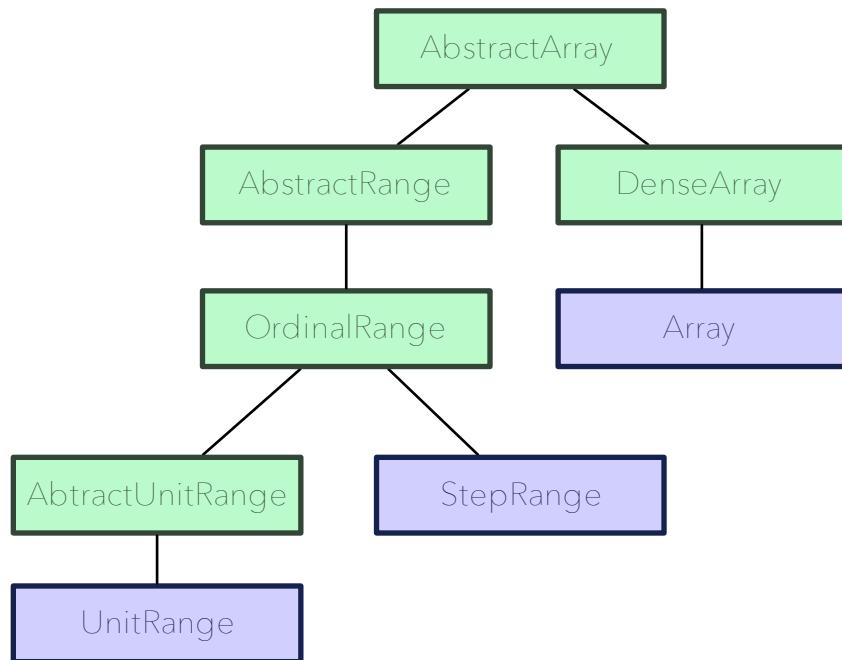


Figure 23: Type hierarchy for arrays and ranges

`OrdinalRange{Int64, Int64}`

Arrays and ranges are *composite types*. Meaning unlike most numbers, they are made up of multiple parts. Let us look at what a unit range and step range are made of.

```
julia> fieldnames(UnitRange)
(:start, :stop)
```

```
julia> fieldnames(StepRange)
(:start, :step, :stop)
```

What this tells us is that objects of type `UnitRange` contains two fields `start` and `stop`, while objects of type `StepRange` contains three fields. It has an extra `step` field.

Let us explore what these fields actually mean by creating some range objects and access their fields.

```
julia> ru = 2:5
2:5
```

```
julia> rs = 3:3:9
3:3:9
```

```
julia> ru.start
2
```

```
julia> rs.start
3

julia> rs.step
3

julia> ru.step
ERROR: type UnitRange has no field step
```

As you can see we get an error when trying to access the `step` field on `ru` which is of type `UnitRange`. A unit range does not have a step size, because it is always one. Instead we use functions to access fields:

```
julia> first(ru)
2

julia> first(rs)
3

julia> last(ru)
5

julia> step(rs)
3

julia> step(ru)
1
```

This is the beauty of using abstractions. You can hide implementation details. Somebody working with a range does not need to distinguish between a step range or unit range. They can be treated the same.

If we don't want to traverse the whole type hierarchy, but just want to find out if an object conforms to some other abstract type we can use the `isa` operator.

```
julia> 2:4 isa OrdinalRange
true

julia> 3:2:5 isa OrdinalRange
true

julia> 3 isa Integer
true

julia> 3 isa Number
true
```

Alternatively if we got a type, rather than an object, we use the `<:` operator.

```
julia> Int64 <: Signed
true

julia> Int64 <: Rational
```

```
false

julia> Int64 <: Real
true

julia> Integer <: Number
true

julia> UnitRange <: AbstractArray
true

julia> UnitRange <: OrdinalRange
true
```

Of course knowing that the range `3:6` is both an `AbstractArray` and a `OrdinalRange` does not matter much if you don't know what that allows you to do. Fortunately Julia offers ways of discovering that.

```
julia> methodswith(OrdinalRange)
[1] +(r1::OrdinalRange, r2::OrdinalRange) in Base at range.jl:1003
[2] -(r)::OrdinalRange) in Base at range.jl:851
[3] -(r1::OrdinalRange, r2::OrdinalRange) in Base at range.jl:1003
[4] ==(r)::OrdinalRange, s)::OrdinalRange) in Base at range.jl:717
[5] first(r)::OrdinalRange{T,S} where S) where T in Base at range.jl:562
[6] iterate(r)::OrdinalRange) in Base at range.jl:590
[7] iterate(r)::OrdinalRange{T,S} where S, i) where T in Base at range.jl:593
[8] last(r)::OrdinalRange{T,S} where S) where T in Base at range.jl:567
[9] reverse(r)::OrdinalRange) in Base at range.jl:946
```

This gives us a list of all functions which take `OrdinalRange` as an argument. Looking at this list we can see e.g. at line 1-3 that ranges can be added and subtracted. Line 5 and 8 tells us that we can get hold of the first and last element in the range.

**NOTE Is Julia a statically typed language?**

If you are familiar with statically typed programming languages such as Java, C++ or C# this list of functions may make you think that Julia is a statically typed language.

`reverse(r)::OrdinalRange)` looks like the definition of a statically defined function which takes an argument `r` of type `OrdinalRange`. However Julia is in fact a dynamically typed language and has more in common with Python, JavaScript and Ruby than with C++ and Java. The distinction will become clearer later in this chapter.

## Multiple Dispatch

We saw earlier that it was very practical having the `step()` function because it allowed us to hide the difference between `UnitRange` and `StepRange`. A `UnitRange` could simply be treated as a `StepRange` with stepsize 1.

A great way of understanding how something works is to try to implement it yourself. So let us make our own step function called `leap()`:

```
leap(r::UnitRange) = 1
leap(r::StepRange) = r.step
```

We can test this function in the Julia REPL. To demonstrate that we are indeed creating different range objects I create them explicitly like this:

```
julia> ru = UnitRange(2, 5)
2:5

julia> rs = StepRange(4, 3, 12)
4:3:10
```

We can see that `leap()` hides the difference between unit ranges and step ranges. The user never has to know that a `UnitRange` does not have a `step` field.

```
julia> leap(ru)
1

julia> leap(rs)
3
```

Previously we would have defined functions like `leap(r) = 1`. That would not work in this case, because we need to distinguish between *which* function should be called depending on the type of the argument `r`. Thus we have added what is called a *type annotation* to the argument `r`. We use the `::` operator to add a type annotation to a function argument.

## Functions have Methods

In Julia terminology we have a **function** called `leap`, but this function has multiple **methods**. One method taking a `UnitRange` and another one taking a `StepRange`. In fact I have only ever shown you syntax for creating methods in Julia. Functions are automatically created once you define a method for which there is no function already defined. However you can chose to only create a function. Start a fresh REPL, and write:

```
julia> function leap end
leap (generic function with 0 methods)

julia> methods(leap)
# 0 methods for generic function "leap":
```

Notice how it says `leap` is a function with zero methods. However as you start

adding methods, you will see `methods()` listing multiple methods for the function `leap`.

```
julia> leap(r::UnitRange) = 1
leap (generic function with 1 method)

julia> methods(leap)
# 1 method for generic function "leap":
[1] leap(r::UnitRange)

julia> leap(r::StepRange) = r.step
leap (generic function with 2 methods)

julia> methods(leap)
# 2 methods for generic function "leap":
[1] leap(r::UnitRange)
[2] leap(r::StepRange)
```

Observe that function `leap` end is different from function `leap()` end. The former creates a function. The latter adds a method to function `leap` if it already exists which takes zero arguments. If the function does not exist, it will also create the function `leap`.

#### **NOTE Methods in object-oriented programming**

If you come from an object-oriented programming background, this terminology will seem strange to you. You are used to methods being functions attached to classes. However in many ways the concepts are not that different. Methods in object-oriented programming also give a function the chance to have many different implementations depending on what class it is attached to.

We could use the same principle to implement our own version of `typeof`, which we will call `kindof`:

```
kindof(x::Int64) = Int64
kindof(x::Float64) = Float64
kindof(x::String) = String
```

Here is an example of using it.

```
julia> kindof(42)
Int64

julia> kindof(3.14)
Float64

julia> kindof("hello")
String
```

This is however a very cumbersome way of implementing `typeof`. We will look

at a more elegant way of doing this when covering Parametric Types.

## Dispatching on Multiple Arguments Example

So far we have only demonstrated what is called single dispatch: selecting a particular method or function implementation, based on the type of one argument. That is not unique to Julia. One can do that in every object-oriented programming language. The syntax would just be different. We would write `r.step()` instead of `step(r)`. However in Julia the type of **every** argument is used to pick a particular method.

Let us create some examples to demonstrate that. We will create a function `inside(item, range)` which checks whether an item or subrange is within another range.

```
function inside(needle::Number, haystack::OrdinalRange)
    haystack.start <= needle <= haystack.stop
end

function inside(needle::Number, haystack::Array)
    sortedstack = sort(haystack)
    sortedstack[1] <= needle <= sortedstack[end]
end

function inside(needle::OrdinalRange, haystack::OrdinalRange)
    haystack.start <= needle.start && needle.stop <= haystack.stop
end
```

So here we have a function `inside` with three different *methods* allowing us to check:

1. Whether a number is within a range. We know that both `UnitRange` and `StepRange` have a `start` and `stop` field, so we can specify that the `haystack` argument should be their supertype, an `OrdinalRange`.
2. If a number is inside the range implied by an array. We need to sort the array to find its minimum and maximum values.
3. Check whether a range is a subrange of another range.

This however does not cover all variations because we have not written a method to handle subranges within an array, or arrays with values within another array. However the important thing is that it demonstrates **multiple dispatch**, picking the correct method to invoke, based on the type of *more* than one argument. Here are some examples of usage:

```
julia> inside(3, 3:5)
true
```

```
julia> inside(2, 3:5)
false
```

```
julia> inside(4, 3:5)
true
```

```
julia> inside(4:5, 2:10)
true

julia> inside(8:12, 2:10)
false

julia> inside(3, [4, 2, 10, 5])
true

julia> inside(12, [4, 2, 10, 5])
false
```

But we can do it much simpler. That is the beauty of abstractions. You can layer it, making new abstractions easy to create. We can utilize the fact that every kind of range and array are all abstracted to the `AbstractArray` type, which tells us that everyone of its subtypes support the operations `minimum` and `maximum`.

```
function inside(needle::Number, haystack::AbstractArray)
    minimum(haystack) <= needle <= maximum(haystack)
end

function inside(needle::AbstractArray, haystack::AbstractArray)
    minimum(haystack) <= minimum(needle) && maximum(needle) <= maximum(haystack)
end
```

## The Power of Abstractions to Simplify

So now we have cut down the number of methods we need to implement to just two. But do you think it is possible to do better? In fact the clever designers of Julia realized that you may want to treat single numbers the same way as ranges and arrays. Look at this:

```
julia> first(4)
4

julia> last(4)
4

julia> minimum(5)
5

julia> maximum(5)
5
```

You see how that beautifully abstracts away the difference between a range and a scale value? That means we can implement `inside` with just one function that handles all cases:

```
function inside(needle, haystack::AbstractArray)
    minimum(haystack) <= minimum(needle) && maximum(needle) <= maximum(haystack)
end
```

**NOTE**

A problem with this implementation, is that we allow `needle` to be **any** type. Thus this method would also get called if `needle` was a string. We could however specify its type as either `Number` or `AbstractArray` by forming the union of the two types. That is written `needle::Union{Number, AbstractArray}`. However union types will be covered more in depth later.

## Single vs Multiple Dispatch

If your previous experience is with an object oriented language, it can be hard to completely grasp the difference between what Julia is doing and what Python, Java or C++ is doing when dispatching a function at runtime.

**IMPORTANT**

This section is not a requirement to read. If your prior programming experience is C, Fortran or Matlab some of the concepts discussed may not be familiar to you. However I do my best to provide footnotes explaining potentially unfamiliar terminology.

To explain better we are going to look at an example, where we assume we have some geometry objects such as `Square`, `Circle` and `Hexagon` defined.

We can imagine defining various functions such as finding intersection and difference between different objects.

```
function intersect(c1::Circle, c2::Circle)
    ...
end

function intersect(c::Circle, s::Square)
    ...
end
```

These operations are finding an area between different geometric shapes as shown below.

## Dynamic Multiple Dispatch Example

Below you can see an explanation of what happens when a function is called in Julia:

1. `intersect` is called with a hexagon object and an object `shape`, of type `Circle`, but this unknown when the program starts running.
2. Julia takes the name of our function and looks through a table over all the functions Julia has loaded into memory. The entry for the `intersect` function contains a pointer to all the methods of the `intersect` function.

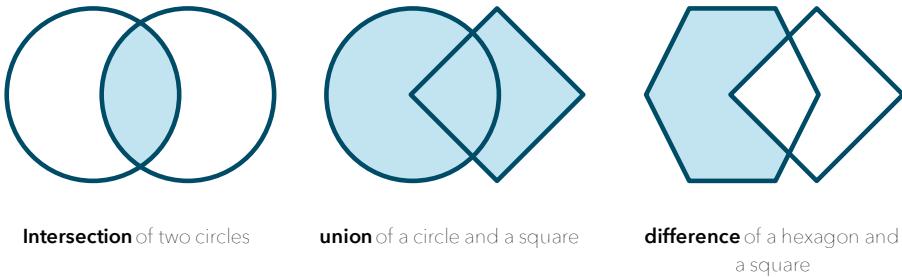


Figure 24: Operations on different geometric shapes

3. Julia essentially calls `typeof(hexagon)` and `typeof(shape)` to find their type. Together Julia creates a tuple<sup>7</sup> (`Hexagon, Circle`) which it uses to search through the method table to find a match.
4. This is where things get interesting. If this method has been called before, Julia jumps straight to point 5. If not we need to touch upon a bit of compiler theory. Before a Julia program can run, it is parsed<sup>8</sup>. This creates an abstract syntax tree (AST). The AST is stored with a method.
5. The Julia *Just in Time Compiler*<sup>9</sup> will take the AST and turns it into machine code. Alternatively the machine code already exists (cached in method table) and is just executed.

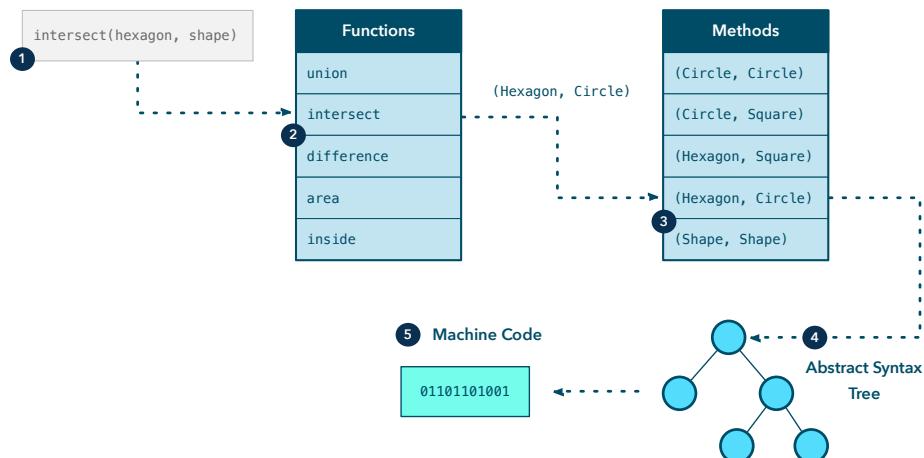


Figure 25: Multiple dispatch in Julia.

<sup>7</sup>A *tuple* is a bit like an array, expect it has fixed size and you cannot change its contents. In other words it is *immutable*. So this is an array [2, 3] and this is a tuple (2, 3).

<sup>8</sup>Parsing is to take a chunk of text and create a data structure called an abstract syntax tree, representing that code. If your language was an interpreted it could execute the code by evaluating this syntax tree.

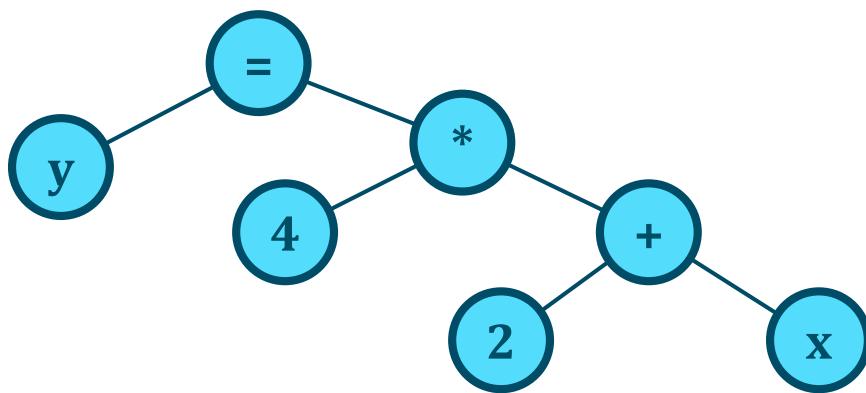
<sup>9</sup>A *Just in Time* (JIT) compiler differs from an *Ahead of Time* (AOT) compiler in that code is compiled on function at a time as they are encountered when the program is running (run time). A traditional compiler (AOT) compiles the whole program and stores it to disk. Every time you run a program, you run this precompiled code.

### ASTs for the Curious

This is not a book about compiler concepts such as abstract syntax trees. But let me give a little bit of information about them to help you understand Julia. Consider an expression such as:

```
y = 4*(2 + x)
```

When a compiler or interpreter reads such code it will usually turn it into a tree structure called an AST, like this:



In Julia every method is turned into such a tree structure. The methods table for each function keeps track of each of these tree structures. The Julia compiler uses these to create actual machine code that the computer understands.

### Dynamic Single Dispatch

It is useful to contrast this with how **single dispatch** in dynamic object-oriented languages such as Python works. This example is inspired by how dispatch is implemented in Objective-C, which is a kind of a hybrid between statically and dynamically typed languages.

1. Although most object-oriented languages would perform a method call like `hexagon.intersect(shape)`, under the hood it is typically more like a regular function `intersect(hexagon, shape)`, where the first argument has special meaning.
2. The first object, `hexagon` in this case has a pointer `isa` to its class `Hexagon` in this case.
3. In a dynamic object oriented language like Objective-C, the method name is a string, which is used to lookup a function implementation in a dictionary. If it cannot find the function in the dictionary, we continue looking by following a pointer, `super` to the super class (super-type).
4. In this example, `Hexagon` is a subclass of `Polygon`. So we try to lookup the method named `intersect` in the dictionary stored on `Polygon`. We find it here.
5. In Objective-C that would mean getting an actual compiled C function. In Python or Ruby it would mean getting hold of some bytecode or an

abstract syntax tree, which is then executed.

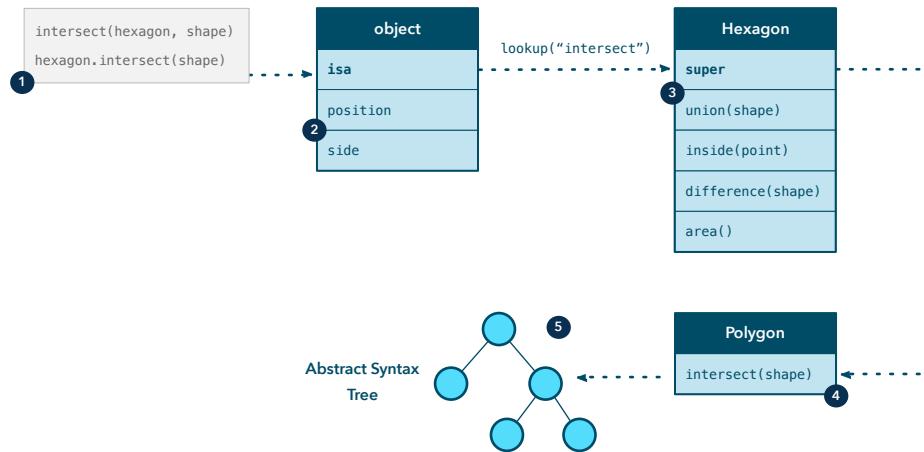


Figure 26: Dynamic single dispatch, as commonly used in object-oriented dynamic languages.

## Static Single Dispatch

Many are not familiar with dynamic programming languages at all. So for those with a C++, Java or C# type of background, here is a contrast with how these languages dispatch a method call at runtime.

- For a statically typed languages there may not be runtime lookup of methods at all. If `intersect` is not a virtual method <sup>10</sup>, the specific method to call will be resolved at compile time (the compiler figures it out when it compiles your code). Here we are showing a call to a method which is dispatched at runtime.
- The `self` or `this` object will contain a hidden member variable, `vtable` which points to an array of functions (function pointers) located at particular indices.
- This `vtable` differs from our dynamic single dispatch, where functions were stored in a dictionary (hash table). At compile time the compiler stores an integer offset into this table to locate the correct function. Secondly there is no chaining to `vtables` of superclasses. Each object has `vtable` which the compiler has figured out what should look like.
- Once the function is located machine code is executed. Alternatively if this was Java or C# a JIT may compile some bytecode.

---

<sup>10</sup>A virtual method is a concept existing in statically typed object-oriented languages. It refers to a method where there is potentially a different implementation in each subclass. When the method is called at runtime, dynamic dispatch will be performed.

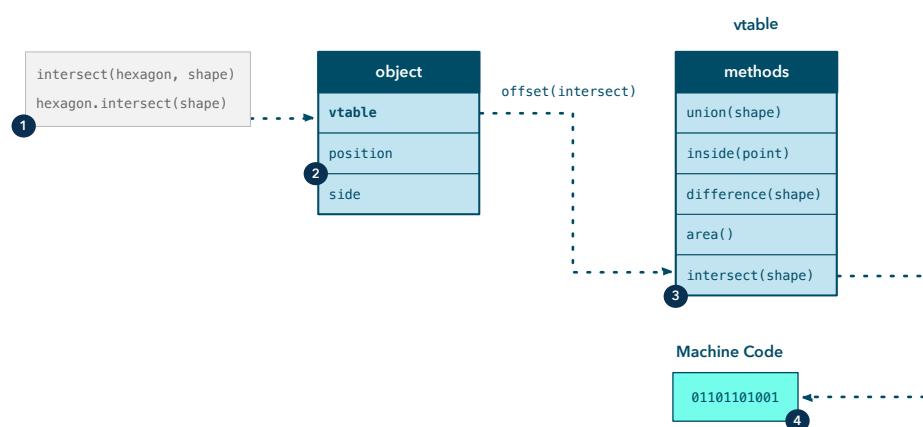


Figure 27: Single dispatch in a statically typed language.



# Defining Your Own Types

- **Composite Types.** Defining your own types made up of more primitive types.
- **Modeling a Space Rocket.** Develop an example using space rockets to explain how types are defined.

Let us look at defining our own types. We will look at creating abstract as well as concrete types which are subtypes of those abstract types. Then we will look at how we can add functions operating on instances of those types utilizing runtime dispatch <sup>11</sup>.

## Simple Rocket

Let us start by modeling a simple space rocket in code. This is a single stage rocket, made up of the following parts from bottom to top:

- A rocket engine.
- propellant tank.
- payload at the top.

The payload is the useful stuff we want to move around in space. It could be a crew module for astronauts or a probe with instruments to explore other planets.

We begin with the simplest part, our propellant tank. Propellant is the matter a rocket engine expels to move forward. In its simplest form it is just a compressed gas being released. In real space rockets however it is a combination of a fuel such as kerosene or hydrogen and an oxidizer such as liquid oxygen (LOX). Internally a propellant tank can contain multiple tanks:

- One for the fuel.
- Oxidizer.
- An inert gas such as Helium to pressurize the fuel tank. To get the fuel pushed out.

However in our simplified model, we don't need to know this. You don't need these details to model the movement of a space rocket. So we will model our propellant tank as something which has:

---

<sup>11</sup>Dispatch is the process or mechanism for selecting a particular function implementation to execute based on types involved. When dispatch happens at runtime we refer to it as dynamic dispatch. This contrasts with static dispatch which happens at compile time.

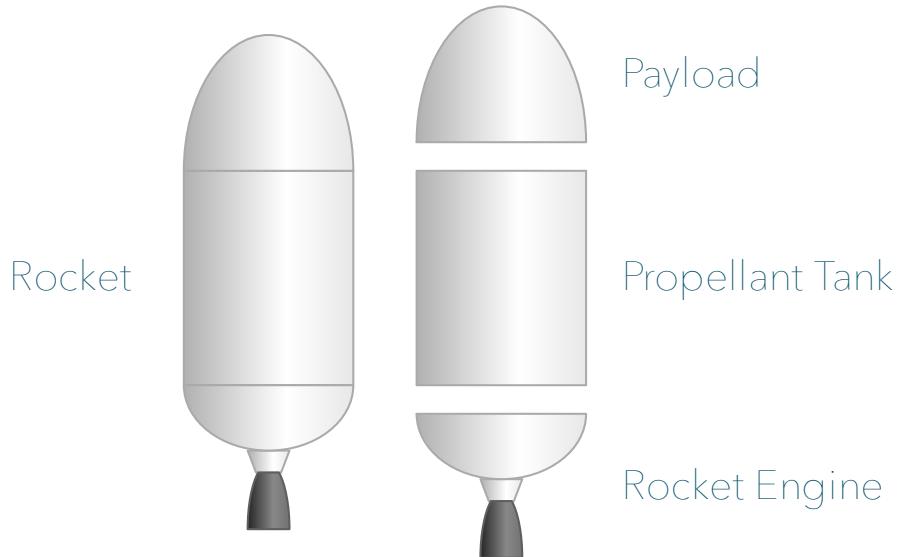


Figure 28: rocket hierarchy

1. A dry mass. The mass of the tank without any propellant.
2. The total mass. What the tank weighs when it is filled up.

In Julia this looks like:

```
struct Tank
    dry_mass::Float64    # excluding propellant
    total_mass::Float64  # including propellant
end
```

This is a composite type, a type made from other types such as two `Float64` numbers in this case. Composite types are made using the keyword `struct`. `end` marks the end of the definition of the type.

You can store this definition in a text file <sup>12</sup>. I stored mine in a file called `rocket-parts.jl` because we will be putting more rocket parts in this file.

#### **IMPORTANT** Code comments

Please note that you don't need to write the parts `# Mass` excluding propellant because these are just comments to the code. Anything beginning with a hash symbol `#` is considered a comment in Julia. This is a common convention in a wide variety of script languages.

Once the file is stored, you can load it into a Julia REPL session:

---

<sup>12</sup>Julia code must be stored as plain text. It is recommended to use UTF8 text format, rather than say ASCII format. However this is the default format for most code editors, so it is not something you need to specify or care about most of the time.

```
julia> include("rocket-parts.jl")
```

Let us make a tank which has a mass of 5 kg empty and 100 kg when filled with propellant. You can see that we can access individual fields once it has been created.

```
julia> t = Tank(5, 100)
Tank(5.0, 100.0)
```

```
julia> t.dry_mass
5.0
```

```
julia> t.total_mass
100.0
```

We can however not change the fields, because a Julia struct is by default immutable:

```
julia> t.dry_mass = 10
ERROR: setfield! immutable struct of type Tank cannot be changed
```

## Mutable Types

That is not a problem at the moment, because we can just make a new tank with the desired properties. However if we want to simulate the rocket flying, we want to be able to keep track of how much propellant is left. That involves having a field for propellant mass which can be changed. In this case we add the `mutable` keyword.

```
mutable struct Tank
    dry_mass::Float64    # excluding propellant
    total_mass::Float64  # including propellant
    propellant::Float64 # propellant mass left
end
```

You can replace your previous version in `rocket-parts.jl` with this definition. However it is important to be aware of that Julia is very strict about redefining previously defined types. Thus if you try to paste this code into the REPL you will get:

```
julia> mutable struct Tank
           dry_mass::Float64
           total_mass::Float64
           propellant::Float64
       end
ERROR: invalid redefinition of constant Tank
```

We will explore ways around this annoyance in the future. But for now, the simplest solution is simply to exit the REPL environment and restart it with the line:

```
$ julia -i rocket-parts.jl
```

This will start the Julia REPL after loading the `rocket-parts.jl` into memory. Now we can see that it is possible to change the fields in the tank.

```
julia> t = Tank(10, 100, 45) # Half full
Tank(10.0, 100.0, 45.0)

julia> t.dry_mass
10.0

julia> t.total_mass
100.0

julia> t.dry_mass = 5
5

julia> t.propellant = 95 # Full tank
95

julia> t
Tank(5.0, 100.0, 95.0)
```

You will notice that to construct a tank you now have to provide the propellant as well. What is perhaps not apparent is that when you define a type, Julia automatically creates a function called a *constructor*, which has the exact same name as the type. This will take as arguments all the fields of the type in the order they were defined.

However that may not be what you want. By allowing the user to specify the propellant mass, you risk setting a value for propellant mass which does not match the maximum allowed. This should not be allowed, but is perfectly possible to do:

```
julia> t = Tank(10, 100, 200)
Tank(10.0, 100.0, 200.0)

julia> t.total_mass < t.propellant
true
```

When working with objects of different types we want to maintain what is called *invariants*. Invariants are things which should always be true such as:

```
0 <= t.propellant + t.dry_mass <= t.total_mass
```

## Outer Constructor

One way which can help achieve that is to create custom constructors which maintain those invariants. You can create any number of constructors as long as the number of or the type of the arguments *differ*, due to multiple dispatch.

Add this function to your `rocket-parts.jl` file:

```
"Create a full tank"
function Tank(dry_mass::Number, total_mass::Number)
```

```
Tank(dry_mass, total_mass, total_mass - dry_mass)
end
```

Now we can create a full tank where the propellant mass does not break the invariant `t.propellant + t.dry_mass <= t.total_mass`. Notice the *text string* above the function. This adds documentation to the function which can be looked up in the Julia help system. Reload `rocket-parts.jl` hit ? to get into help mode and write `Tank`:

```
help?> Tank
search: Tank ReentrantLock tanh tand tan atanh atand atan instances transpose transcode UnitRange
```

Create a full tank

Also notice that I write the arguments to the function as `dry_mass::Number` rather than `dry_mass::Float64`. That is to allow a variety of numbers. We don't care if the user specifies tank mass using integer numbers e.g. If I had defined the constructor like this:

```
function Tank(dry_mass::Float64, total_mass::Float64)
    Tank(dry_mass, total_mass, total_mass - dry_mass)
end
```

Then I could easily have gotten errors when constructing a tank object. As you can see below, Julia is complaining that it cannot find any function called `Tank` which has a method which takes `Int64` arguments.

```
julia> t = Tank(10, 100)
ERROR: MethodError: no method matching Tank(::Int64, ::Int64)
Closest candidates are:
  Tank(::Any, ::Any, ::Any) at rocket-parts.jl:2
```

The default `Tank` constructor Julia makes is flexible on the types it accepts.

## Inner Constructor

One of the problems with defining our constructor the way we have, is that it doesn't nullify the constructor Julia already made for us. We can still, wrongly, write `Tank(10, 100, 200)`. That is because we wrote an **outer constructor**. That is what we call constructors defined outside the type. However if we define an **inner constructor**, as seen below we replace the default constructor provided by Julia.

```
mutable struct Tank
    dry_mass::Float64
    total_mass::Float64
    propellant::Float64

    "Create a full tank"
    function Tank(dry_mass::Number, total_mass::Number)
        new(dry_mass, total_mass, total_mass - dry_mass)
    end
end
```

In this case it is not possible to create the wrong kind of tank.

```
julia> t = Tank(10, 100, 200)
ERROR: MethodError: no method matching Tank(::Int64, ::Int64, ::Int64)
Closest candidates are:
  Tank(::Number, ::Number) at none:8

julia> t = Tank(10, 100)
Tank(10.0, 100.0, 90.0)
```

You will notice we made another small change. We wrote `new(dry_mass, ...)` rather than `Tank(dry_mass, ...)`. That is because we are replacing the built in `Tank` function so it can no longer be called. `new` can only be used in *inner constructors*. It is quite flexible in that you can provide fewer arguments than there are fields. If you don't specify a value for a field with `new`, numbers will be initialized to zero.

## Rocket Engine and Payload

In the simplest case we can model payload as just a type with mass.

```
struct SpaceProbe
    mass::Float64
end

struct Satellite
    mass::Float64
end
```

This may seem simplistic but remember we are creating models. Models only contain the properties we need to answer questions we are interested in. E.g. an initial model of smart phone may just be a block of wood, no buttons, screen or color scheme. Why? Because initially the questions you want answered are:

Is this shape or size comfortable to carry in my pocket? How much space do we have available to create a screen and electronics inside?

The same applies to designing and building a rocket. Initially we are only interested in mass budgets. We want to know things such as:

1. How much propellant do I need?
2. How big payload can I launch into orbit?
3. How far can a given rocket go?

To answer such questions we don't need to include in our model what sort of instruments exist on the space probe or what kind of batteries or solar cells it has. Our rocket engine will have more details however:

```
struct Engine
    thrust::Float64 # Newton
    Isp::Float64     # Specific Impulse
```

```
mass::Float64    # Kg
end
```

`thrust` is the force produced by the rocket engine. If we know the total mass of the rocket this helps us calculate how much the whole rocket accelerates once the rocket engines are fired up. We get this from Newtons second law, which states that force  $F$  is proportional to mass  $m$  times acceleration  $a$ :

$$F = ma \iff a = \frac{F}{m}$$

However to know how much mass we are pushing at any given time, we need to know how much propellant the engine consumes each second. Thrust alone cannot tell us that. For that we need another important property of a rocket engine, called the specific impulse (ISP). It is an analogy to gas mileage for a car. Unlike a car on a road, a rocket in outer space continues moving even without thrust, so you cannot measure fuel efficiency (or propellant efficiency) by how far a kg of propellant gets you. Instead we measure it in terms of how many seconds a unit of propellant can sustain a force of 1 G (the force of gravity on earth).

This allows us to calculate mass flow (consumption of propellant per second).

```
g₀ = 9.80665 # m/s² acceleration of gravity on earth
mass_flow(thrust::Number, Isp::Number) = thrust / (Isp * g₀)
```

We can e.g. use this to calculate the propellant consumed per second in a Falcon 9 rocket. It has 9 Merlin 1D engines, each with a specific impulse of 282 s and thrust of 845 kN.

```
julia> engine_thrust = 845e3
845000.0

julia> Isp = 282
282

julia> thrust = engine_thrust * 9
7.605e6

julia> flow = mass_flow(thrust, Isp) # kg/s
2749.979361594732
```

So we get that a Falcon 9 rocket consumes an estimated 2.7 tons of propellant each second.

## Assemble the Rocket

Now we got all the pieces to assemble our rocket.

```
mutable struct Rocket
    payload::SpaceProbe
    tank::Tank
```

```
    engine::Engine
end
```

Let the rocket building start! We will make a rocket resembling the Falcon 9 rocket. It can put a payload into low Earth orbit weighing 22.8 tons. We will put a space probe with that mass on top. The first stage of the Falcon 9 rocket has a total mass of about 433 tons, of which 22 tons is dry mass.

```
julia> rocket = Rocket(SpaceProbe(22.8e3), Tank(22e3, 433e3), Engine(845e3, 282, 470))
Rocket(SpaceProbe(22800.0), Tank(22000.0, 433000.0, 411000.0), Engine(845000.0, 282.0,
```

However if you have been paying attention, you may realize there are multiple problems with this setup:

1. We can only put **one** kind of payload on this rocket, a space probe. What if we want a crew capsule instead?
2. The Falcon 9 rocket has nine Merlin engines not just one.
3. Space rockets have multiple stages, which separate as the rocket goes higher. Our rocket only has a single stage.

## Creating Abstractions and Encapsulation

What we need, is to do the same as we saw with ranges and numbers earlier. We need to create abstractions, extracting common behavior among different objects. One way to start doing that is by defining and using abstract types. If you remember from earlier, `Number` was an abstract type representing all numbers, while `Integer` was an abstraction of all concrete integers such as `UInt8`, `Int32`, `BigInt` etc. We will unify different types of payloads with the `Payload` abstract type.

```
abstract type Payload end

struct Capsule <: Payload
    mass::Float64
end

struct SpaceProbe <: Payload
    mass::Float64
end

struct Satellite <: Payload
    mass::Float64
end
```

Now we can generalize the kind of payload a rocket can have.

```
mutable struct Rocket
    payload::Payload
    tank::Tank
    engine::Engine
end
```

But wait we are not done. Payloads can be generalized further. A space rocket is a bit like a Matryoshka doll. The rocket has a payload, which is another rocket. This rocket has a payload which is yet another payload. Below is an illustration of popping open the payload of each rocket one stage separation at a time.

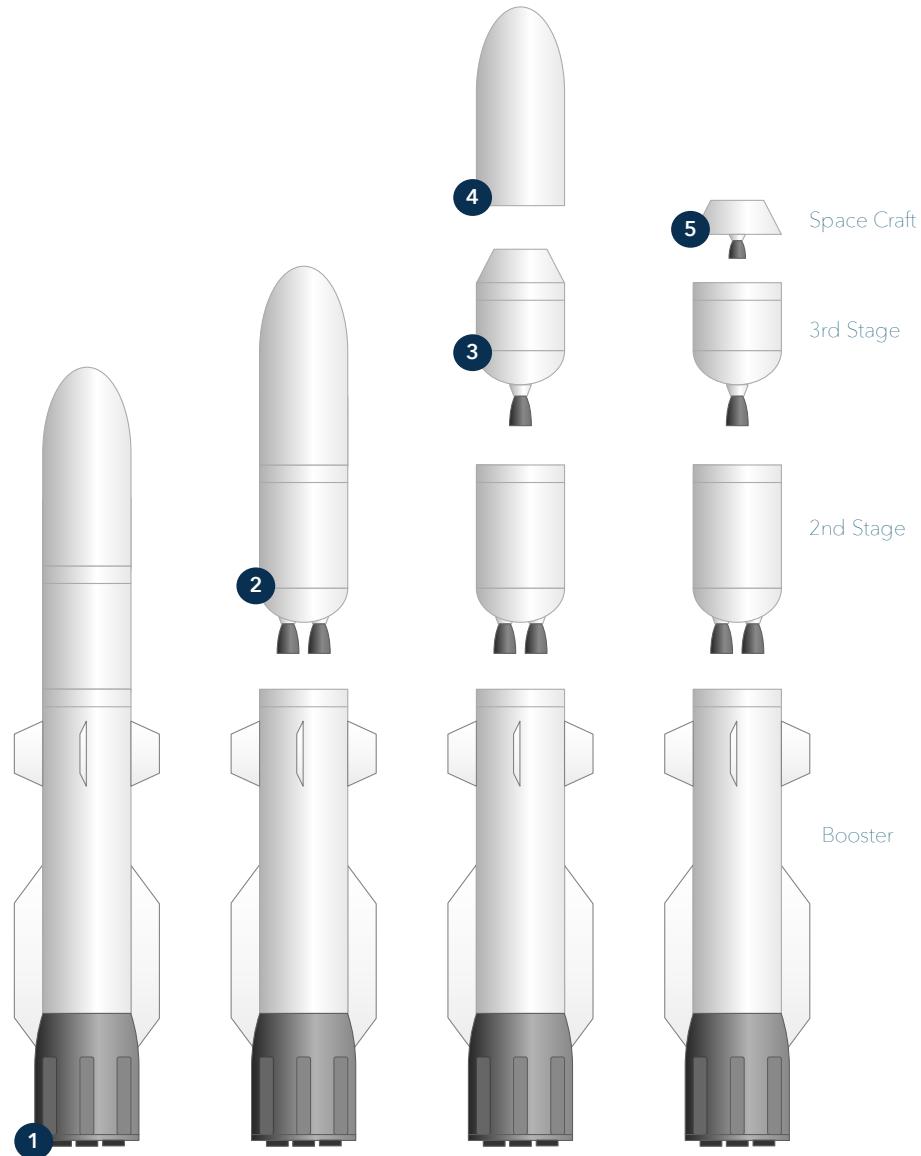


Figure 29: rocket stages

Rocket is a very general term. So we have more specific names for rockets of particular type.

1. We start with a rocket consisting of multiple rockets or a multi-stage rocket. We refer to this as the space vehicle. Basically a rocket powered vehicle taking stuff to space.

2. We pop open the space vehicle, which has a payload which is yet another rocket. This is our 2nd stage. What is left behind is the booster. The booster is the first stage rocket that pushed our 2nd stage rocket up to velocity and altitude.
3. On top of the 2nd stage rocket there is a cap, called the fairing. We pop that off, and below we find yet another rocket. Just like a Russian Matryoshka doll.
4. The fairing gets dumped in the ocean. It has no other purpose than protecting the top payload.
5. At the end we get to the final payload which is also a form of rocket. Except we call this a space craft, because it moves around in space. It could be a space probe, capsule, satellite or whatever.

So now we can go one step further and make the rocket a subtype of `Payload` as well.

```
mutable struct Rocket <: Payload
    payload::Payload
    tank::Tank
    engine::Engine
end
```

But there is one more change needed. As mentioned rockets don't always have single engines. They could have clusters of multiple engines as well. But we want to abstract the difference away between a single engine and an engine cluster. To do that we will turn `Engine` into an abstract type.

```
abstract type Engine end

struct SingleEngine <: Engine
    thrust::Float64
    Isp::Float64
    mass::Float64
end

struct EngineCluster <: Engine
    engine::SingleEngine
    count::Int8
end
```

We don't necessarily want users of our `Engine` type to be exposed to the difference between an `SingleEngine` and just an `Engine`. We can thus make a constructor function for `SingleEngine` with the name `Engine`.

```
function Engine(thrust::Number, Isp::Number, mass::Number)
    SingleEngine(thrust, Isp, mass)
end
```

This is perfectly possible to do, since you cannot create instances of abstract types anyway.

To abstract what an engine is, we have to encapsulate the technical difference between a cluster and an engine. So we need to define functions for all important attributes erasing these differences.

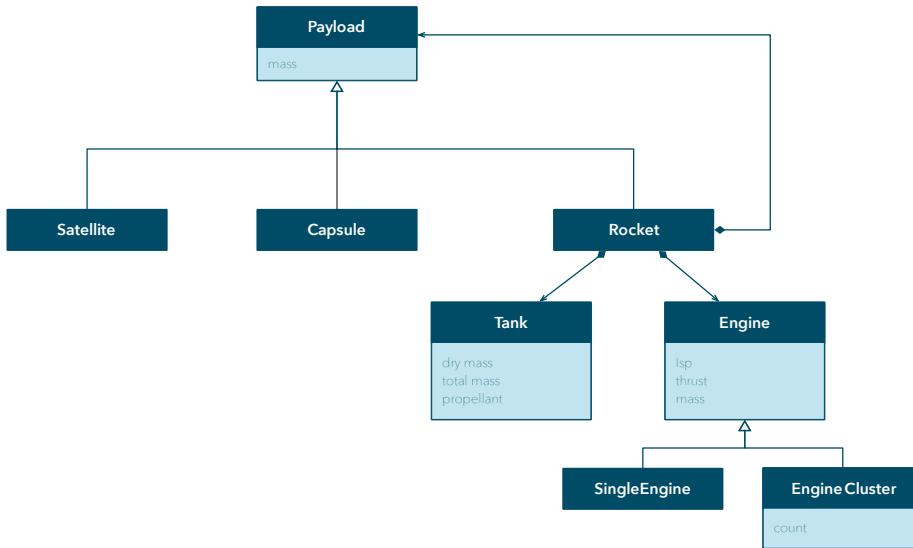


Figure 30: Conceptually how we want our types modeled expressed in the UML modeling language.

```

thrust(engine::SingleEngine) = engine.thrust
thrust(cluster::EngineCluster) = thrust(cluster.engine) * cluster.count

mass(engine::SingleEngine) = engine.mass
mass(cluster::EngineCluster) = mass(cluster.engine) * cluster.count

Isp(engine::SingleEngine) = engine.Isp
Isp(cluster::EngineCluster) = Isp(cluster.engine)
  
```

We can use these abstractions to define a function `update!`, which takes care of depleting propellant as we simulate our rocket flying. We simulate by performing small time steps  $\Delta t$ .

#### **NOTE Simulations and Games**

When simulating something complex such as a rocket flight where the thrust will vary over time, the direction of the rocket will change, air resistance and gravity then a purely analytical solution with a simple equation doesn't work. We have to split up what happens into multiple tiny time steps, and make calculations for each step.

Computer games work much the same way. Every frame drawn on the screen you would typically update the positions of characters, bullets, swords or whatever is in motion based on their current position, velocities and accelerations.

```
function update!(r::Rocket, Δt::Number)
```

```
mflow = mass_flow(thrust(r.engine), Isp(r.engine))
r.tank.propellant -= min(mflow * Δt, r.tank.propellant)
end
```

Notice the exclamation mark (!) in `update!`, it is there to remind us that this function alters (mutates) its input data. The propellant field in the rocket's tank gets changed.

We now have enough functionality to define a Falcon 9 rocket with multiple stages:

```
julia> second_stage = Rocket(SpaceProbe(22.8e6), Tank(4e3, 111.5e3), Engine(845e3, 348
Rocket(SpaceProbe(2.28e7), Tank(4000.0, 111500.0, 107500.0), SingleEngine(845000.0, 348
```

```
julia> first_stage = Rocket(second_stage, Tank(22e3, 433e3), EngineCluster(Engine(845e3, 348
Rocket(Rocket(SpaceProbe(2.28e7), Tank(4000.0, 111500.0, 107500.0), SingleEngine(845000.0, 348
```

Later when performing physics calculations it is useful to be able to abstract away how a property such as mass is determined for a rocket or part of a rocket.

```
mass(tank::Tank) = tank.dry_mass + tank.propellant
mass(probe::SpaceProbe) = probe.mass
mass(capsule::Capsule) = capsule.mass
mass(satellite::Satellite) = satellite.mass

function mass(r::Rocket)
    mass(r.payload) + mass(r.tank) + mass(r.engine)
end
```

You can see the benefits of abstraction of mass in how `mass(r::Rocket)` is defined. Implementing this function we don't have to concern ourselves with details of the payload. The payload could be a space probe or 10 other rockets stacked on top of each other. We don't have to care. We know there is a `mass` function defined for all of them which takes care of those details.

Likewise we don't have to concern ourselves with whether we are getting the mass of a single engine or engine cluster. Imagine we had implemented this function *before* creating the `EngineCluster` type. We would not need to change this implementation, because as long as `EngineCluster` is of type `Engine` and has implemented `mass`, everything works.

# Static vs Dynamic Typing

- **When** do types exist in a statically and dynamically typed language?
- **Type Info Storage** How is type information stored.
- **Runtime**. How are types accessed and used at runtime?

Julia is a dynamically typed language, but for those already familiar with dynamic and static languages this may seem strange given its syntax. The frequent use of type annotations in Julia makes many people think Julia is a statically typed language.

This means there is a need to go into more depth about what the difference between static and dynamic typing is.

## Values Have Types Not Expressions

A common misconception about dynamic typing is that it is the same as making every object the same type. For someone with a Java background it would be the same as saying every object is of type `Object` or for those with C/C++ background it would be like stating everything is of type `void`.

However this is not the difference.

The difference between dynamic and statically typed languages is that in statically typed languages *expressions* have types, while in dynamically typed languages *values* have types.

This needs some unpacking. Consider the Julia expression below where we morph some doodad to get a new object which we then combine with a gadget to get a thing which should be of type Thingy.

```
thing::Thingy = combine(morph(doodad), gadget)
```

Further the Thingy and Doodad types are defined as:

```
struct Thingy
    length::Int64
end

struct Doodad
    capacity::Int64
```

```
    charged::Bool
end
```

If Julia was a statically typed language, then this code would have to be compiled before it could be run.

#### **NOTE Compilation and Compilers**

A compiler reads source code files and turns them into binary<sup>13</sup> code which is stored in a new file, the *executable*. This process is called *compilation*. The executable file can be run directly by the operating system<sup>14</sup> on your computer.

Conceptually types don't exist at runtime (when the program runs) for statically typed languages. This is not entirely true in all cases however, but we will get to the exceptions later.

If Julia was a statically typed language then the compiler would compiled our code example by performing the following steps related to types:

1. Lookup where the `doodad` variable as defined and figure out what type it was. Say it is of type `Doodad`.
2. Lookup definition of `morph` and see what the type of its argument is. It would then have to make sure this argument is of type `Doodad` or a subtype of `Doodad`.
3. Find where `gadget` is defined to determine its type.
4. Make sure that the types of the arguments to `combine` match the type of the output from `morph` and the type of `gadget`.
5. Make sure `combine` returns a type which is the same as `Thingy` or a subtype.

If just one of these type checks fail, then the compiler will produce an error and it *never* spits out the finished program you were eager to run.

To compile the final program, the compiler has to able to hunt down the definition of every variable, type and function you use. Everything must be known.

What happens if everything is not known? Then you risk undefined behavior which is very bad. Imagine `morph` got an input which was not a `Doodad` but a `Thingy`, and `morph` is defined this way:

```
function morph(doodad::Doodad)
    ...
    doodad.charged = true
    ...
end
```

A `Thingy` does not have a `charged` field. A `Thingy` object is 8 bytes large to hold the 64 bit `length` field. This `charged` field would normally be the 9th byte. But if you have an object of type `Thingy`, but assume it is a `Doodad` and try to read the data on the 9th byte, you will get hold of some data potentially belonging to another variable.

You thus risk making random changes to memory, which eventually will cause weird random behavior.

A dynamically typed language in contrast knows at runtime, what its types are. Before performing `doodad.charged = true` it will make sure `doodad` has a `charged` field. If it does not, it will throw an exception.

This is safer because it represents a *controlled crash*. The programmer can be informed about exactly what went wrong and *where*. And user data will not get *accidentally* corrupted.

This is a very important point. Crashing in this case is a *good thing*, because you are catching a potential problem. If the program kept running while corrupting memory you would risk invalid calculations being performed and stored to disk.

Imagine using a text editor which does not crash when the program makes a mistake. Instead it corrupts memory and when you save your file, you save corrupted data. Yet you are never informed of this behavior, because no crash ever happens. Instead a week later when you open the file you will see that it contains gibberish. This would be bad.

Instead it is preferable to have a controlled and announced crash. This allows users to know that something went wrong, and that perhaps some of the data they currently worked on may have gotten lost. This is an issue they can immediately investigate and correct. Silent failure in contrast means you can have corrupt files and problems all over without knowing about it.

#### **WARNING Don't Hide Failure!**

Many programmers get so focused on fixing crashes that they end up writing code that attempts to hide failure rather than producing a crash. Then you are missing the point. You can attempt to catch and correct a failure in your program code. However if you are *unable* to correct the failure, a crash is often preferable. This depends in large part on the type of software. For instance if a web browser fails to render a webpage it will likely be overkill to crash the program. Corrupted graphics for a webpage is usually not a serious problem.

## **Consequences of Compiler Omniscience**

One of the reasons people often find dynamic languages such as Julia, Python and JavaScript easier to use is that less complex scaffolding is needed.

For the reasons described earlier, a compiler for a statically typed language needs to know everything. It needs to be omniscient like a deity. This frequently creates a need for a complex build system and project management system, where the programmer has to define for all source code files, which files depend on each other and in what sequence. E.g. if file `foo` defines `Thingy` and file `bar` uses `Thingy` then `bar` is dependent on `foo`.

This is one of the reasons why users of statically typed languages such as Java,

C++ and C# tend to use large and complex integrated development environments (IDE).

## Evaluation vs Compilation

If you come from a static typing background one thing which is hard to wrap ones head around, is the fact that almost everything in a dynamically typed language are expressions or statements which get evaluated at runtime.

```
function add(x::Int64, y::Int64)
    x + y
end
```

Consider the function definition above. In a statically typed language, when this definition is encountered by a compiler it will turn it into some chunk of code which then potentially gets called at runtime.

That means from the moment your program starts to run, the `add` function already exist.

Contrast this with a dynamic language such as Julia. Code is executed one line at a time. If the the lines defining the function `add` have not yet been reached, the function simply does not exist.

It is only brought into existence at runtime, as soon as Julia executes the lines of code that defines this function.

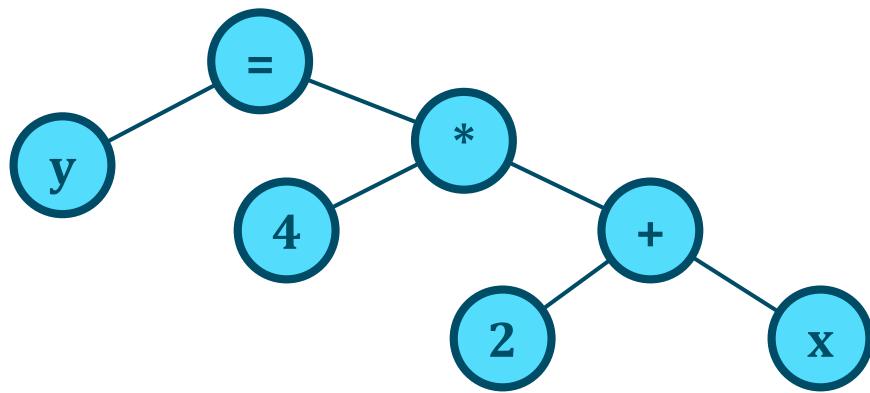


Figure 31: Abstract Syntax Tree

Although technically reality is slightly more complicated. When a Julia source code file is loaded, the whole thing gets parsed, and turned into an *abstract syntax tree* (AST). It is nodes in this tree which get evaluated in turn and not the raw source code lines.

The line beginning with the keyword `function` is an expression like any other in Julia. It is an expression which produces an object, a function object to be specific. You can store this object in a variable if you want to.

```
julia> adder = function add(x::Int64, y::Int64)
           x + y
       end
add (generic function with 1 method)

julia> adder(3, 4)
7
```

## Type Annotations is Not Static Typing

Julia type annotations may look like static typing but this is highly deceptive. You may think that one could easily create a Julia compiler which could compile Julia code much the same way as statically typed code.

However this would be hard, because types in Julia are objects and type annotations are expressions which get evaluated like any other expression.

Here is a simple example showing why a hypothetical Julia compiler would struggle to figure out the types of the arguments to `add`:

```
types = [Int64, Float64, Char]

function make_type(n::Integer)
    if n > 0
        types[n]
    else
        Int8
    end
end

function add(x::make_type(1), y::make_type(2))
    x + y
end
```

This example demonstrates several things which are normally not possible to do in statically typed language.

First we put types into an array like any other object. When you can treat types as regular objects, we call them *first class objects*.

Next we define a function `make_type` which returns a type. Notice this function can be of arbitrary complexity. You have the whole Julia language at your disposal!

Finally we define our `add` function. In this case the type of the arguments `x` and `y` is determined by a function call. A compiler could thus not predict what types this function deals with. It is only at runtime, at the point where the `add` function definition gets evaluated, that we know what types it operates on.

No matter how smart you make the compiler you cannot solve this problem. `make_type` e.g. could in theory read something from a file to determine what type it should return. A compiler cannot predict what data will reside on that file.

## Julia JIT Compilation

What complicates matters when talking about Julia in terms of static and dynamic typing is that Julia does in fact have a compiler. It has what we call a *Just in Time* (JIT) Compiler. This means unlike languages such C/C++, Fortran or Go, compilation does not happen once prior to a Julia program running. Instead compilation happens in principle *every time* a Julia programs runs, because it happens during **runtime**.

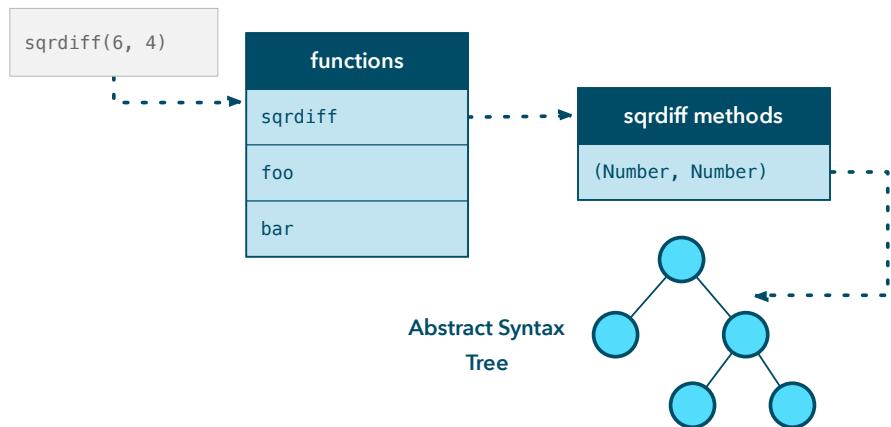
Every time a function is called for the first time, Julia compiles it. In fact as we have seen earlier, every function in Julia can have zero or more method implementations.

Even if you only wrote a single method, Julia can end up compiling numerous methods for a given function, because specialized version of every combination of inputs will be made. Let me clarify this further with an example. Consider a simple function `sqrdiff` working on two abstract number types:

```
function sqrdiff(a::Number, b::Number)
    x = a + b
    y = a - b
    x * y
end
```

I deliberately use abstract types, because there is no way for `sqrdiff` to be called with instances of `Number`. The actual type of `a` and `b` will have to be some concrete number type such as `Int64` or `UInt8` at runtime.

This method definition, will create a single method entry under the `sqrdiff` function, which will point to an abstract syntax tree (AST) of this code. Julia made this AST by parsing the source code, when it was first loaded.



Say a user makes the call `sqrdf(6, 4)`. It means the integer values 6 and 4 of type `Int64` (`Int` for short) is passed to the function for processing.

Julia will begin by looking up the function in the function table. Next it will ask each argument `a` and `b` what their type is. Keep in mind that this is possible in a dynamic language, because values *know* their own type at runtime.

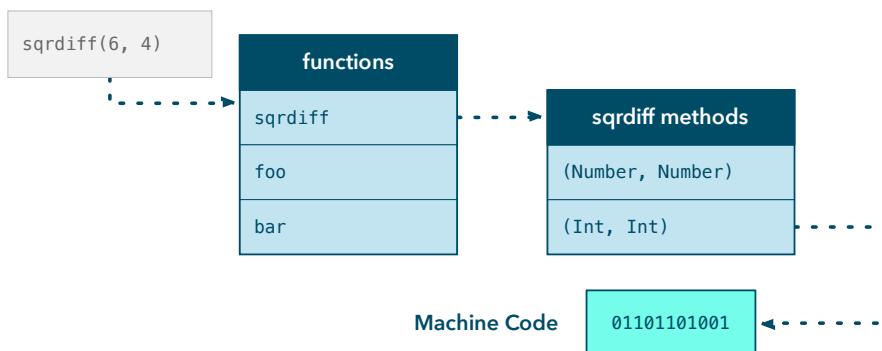
Julia will use this to pick an appropriate method implementation. The only matching entry it can find will be the one for the abstract types (`Number`, `Number`). This will point it to an abstract syntax tree of the code.

You can think of this tree as a sort of template for the code. When Julia parsed the code and made the tree, it could not know the concrete type `a` and `b` at runtime. Consequently it could not know the type of `x` and `y` either.

But the JIT compiler will analyze this AST to figure out the types of each value used. In this case it can. It concludes that if the inputs `a` and `b` are of type `Int` then it logically follows that `x` and `y` are of type `Int` as well. This allows the JIT to compile machine code.

This machine code will be very similar to the code a compiler for a statically typed language makes. Type information is gone. `a`, `b`, `x` and `y` at this point are really just binary blobs<sup>15</sup>, which are added, subtracted and multiplied with machine code instructions.

Julia will store this machine code under a new entry as shown below.



While not explicitly shown here, the AST still exists under the (`Number`, `Number`) entry. Thus if `sqrdf` is called with different concrete types, then the Julia Just in Time Compiler will stamp out new machine code chunks using the AST and add an entry for them.

The benefit, is that next time `sqrdf` is called with two 64-bit integers, that call executes quickly because it can just lookup machine code that was made for the (`Int`, `Int`) entry.

If we take a step back and look at this process, and reflect upon what is going on, you could say that Julia is a chameleon. Conceptually and from the users point of view, it acts as a dynamic language. But if you took a snapshot of Julia

<sup>15</sup>A *binary blob* is a collection of binary data of some unknown size and format. It is a chunk of data without any known encoding, so you just look upon it as a string of ones and zeros.

code that has been running for a while, then what is in memory would look very similar to what would be produced by compilation of a statically typed language.

However this is not a perfect analogy. In our example we have a simple function where the JIT can figure out the concrete types of all values used. That is not always possible.

For instance if Julia could not determine whether `x` is a `Int` or a `Char`, then `x` gets assigned a union type `Union{Int, Char}` by the JIT. This is not a concrete type and thus in this case the `x` would have to store along with its value, what type it is.

In such cases Julia will have the same kind of low performance that other dynamic languages have, since any operation involving `x` will require a type check.

Julia performance derives primarily from the fact that it is possible to guess the concrete type of most variables inside a method in most cases. This highly optimized machine code can be generated by the JIT, not very different from what a C or Fortran compiler would do.

## Runtime Information About Types

What we have described is the conceptual difference between dynamic and static typing. However in reality this picture is a lot more messy.

Many statically typed languages, especially object-oriented ones will often keep runtime information about their values. This is to deal with class inheritance hierarchies.

Nor do dynamically typed languages necessarily always keep information about their types in practice. As we have seen with JIT compilation, there will be chunks of code where there is no type information stored. However to the user that is largely invisible, because if you wrote code to inspect type information at runtime, then the JIT compiler would take that into account.

Nor do statically typed languages necessarily result in machine code. E.g. both Java and C# produce bytecode<sup>16</sup> which is JIT compiled at runtime to machine code.

Both the Nim and Haxe programming languages are statically typed languages which can compile to both C code and JavaScript rather than machine code. Hence a statically compiled language can in principle be compiled to a dynamically typed language.

Thus it is not what happens under the hood which matters but how the language works conceptually. What you as a user of the language is allowed to do. Julia is conceptually a dynamically typed language even though under the hood the JIT compiler will assign types to expressions and compile them to machine code.

---

<sup>16</sup>Bytecode is a sort of pretend machine code. It will look a lot like assembly code, but for a virtual machine. The benefit is that different actual computers can all simulate this same virtual machine, allowing code compiled to bytecode to run on a wide variety of hardware platforms.

# Conversion and Promotion

- **Conversion** of one related type to another. For instance an integer to a floating point number.
- **Promotion** is about finding the least common denominator among related types in an expression.
- **Inspecting** the standard library in Julia and discover how it works using `@edit`.

You may not have paid much thought to the fact that Julia effortlessly handles arithmetic with completely different number types.

```
julia> 3 + 4.2  
7.2
```

```
julia> UInt8(5) + Int128(3e6)  
3000005
```

```
julia> 1//4 + 0.25  
0.5
```

You may not have thought about it because most mainstream programming languages are capable of doing this. Underneath the hood, most programming languages have defined a set of *promotion rules*, which says what should be done if you try to combine numbers of different types. Promotion rules make sure all the numbers are converted to a sensible common number type which can be used in the final calculation.

Usually this mechanism is hidden from the developer. It is documented, but not something you can influence in any way. In this chapter we are going to explore the powerful mechanisms for handling promotion and conversion of numbers in Julia, by covering these topics:

- How Julia treats numbers different from other languages.
- Use examples with built-in numbers.
- Develop an example of using different temperature degrees such as Celsius, Kelvin and Fahrenheit and how you can do promotion and conversion between them.

## Number Promotion

Many modern languages have chosen to not support number promotion because the experience with promotion and automatic conversion has been quite bad. Part of the reason for this, is that the mechanism is buried deep down in the compilers of respective languages. It is not something accessible or introspectable by the users of those languages.

Being designed for numerics, Julia is fundamentally different in this area. Numbers are not built into Julia, rather they are first class objects. By that we mean they are not fundamentally different from any other object in Julia. Numbers are just types defined in the standard library, like many other types. The benefit of this approach is that as a user of Julia you can define your own types, and they will not have any less performance or capability than the number types Julia is delivered with.

Promotion and conversion of numbers is handled by regular Julia functions. It is easy to see what Julia does by using the `@edit` macro.

**NOTE Julia environment variable setup**

For it to work you need to have set the `JULIA_EDITOR` environment variable. This will depend on your operating system. I use the fish shell where I would write the line:

```
set -x JULIA_EDITOR mate
```

In my `$HOME/.config/fish/config.fish` startup configuration file. If you use the bash shell instead you would write:

```
export JULIA_EDITOR=mate
```

In the `$HOME/.profile` configuration file for the bash shell. This will work on macOS and Linux. Windows users also have the option to install a Unix type shell where this would work. Regardless this is not something you actually have to do to read this chapter. It is merely a way of explaining how I obtain the details I am showing you.

Below we are adding an integer and floating point number. By prefixing with the `@edit` macro Julia jumps to the definition of the function being called to handle this expression, allowing you to have a look at the source code.

```
julia> @edit 2 + 3.5
```

**NOTE Everything is a function!**

It is worth being aware of that almost everything in Julia is a function call. When you write `3 + 5`, that is syntactic sugar for calling a function named `+` like this `+(3, 5)`. Every function using a symbol such as `+`, `-`, `*` etc supports being used in infix form.

What you see here is that every arithmetic operation on some `Number` in Julia first calls `promote` before performing the actual arithmetic operation.

```
+{x::Number, y::Number} = +(promote(x,y)...)
*{x::Number, y::Number} = *(promote(x,y)...)
-{x::Number, y::Number} = -(promote(x,y)...)
/{x::Number, y::Number} = /(promote(x,y)...)
```

The `...` is called the ``splat'' operator. You use it to turn to arrays or tuples into function arguments. So `foo([4, 5, 8]...)` is the same as `foo(4, 5, 8)`. We use it because we want to turn the tuple returned by `promote` into arguments to the various arithmetic functions `+`, `-`, `*` etc.

```
julia> promote(2, 3.5)
(2.0, 3.5)

julia> typeof(1//2), typeof(42.5), typeof(false), typeof(0x5)
(Rational{Int64}, Float64, Bool, UInt8)

julia> values = promote(1//2, 42.5, false, 0x5)
(0.5, 42.5, 0.0, 5.0)

julia> map(typeof, values)
(Float64, Float64, Float64, Float64)
```

As you can see from these experiments, `promote` gives us a tuple of numbers converted to the most appropriate common type. So if you are uncertain about how promotion works in Julia, you can just test the different parts of it yourself.

## Number Conversion

The recommended and simplest way of doing conversion in Julia is to use the constructor of the type you want to convert to. So if you have a value `x` you want to convert to some type `T`, then just write `T(x)`. Let me give some examples:

```
julia> x = Int8(32)
32

julia> typeof(x)
Int8

julia> Int8(4.0)
4

julia> Float64(1//2)
0.5

julia> Float32(24)
24.0f0
```

Keep in mind that a conversion is not always possible to perform:

```
julia> Int8(1000)
ERROR: InexactError: trunc(Int8, 1000)

julia> Int64(4.5)
ERROR: InexactError: Int64(4.5)
```

An 8-bit number cannot hold values larger than  $2^8 - 1$ , and integers cannot represent decimals.

In many cases conversions are done implicitly. In these cases we don't use the constructor function but a function called `convert`. While it is typically called implicitly, you can explicitly call it as well, as these examples demonstrate:

```
julia> convert(Int64, 5.0)
5

julia> convert(Float64, 5)
5.0

julia> convert(UInt8, 4.0)
0x04

julia> 1//4 + 1//4
1//2

julia> convert(Float32, 1//4 + 1//4)
0.5f0
```

Notice the first argument in these function calls: `Int64`, `Float64` etc. These are type objects. Types are first class objects in Julia, meaning they can be handled like any other object. You can pass them around, store them and define methods that operate on them. Type objects even have a type. The type of `Int64` would be `Type{Int64}` and for `Float64` it is `Type{Float64}`.

The `convert` function is called implicitly in these cases:

- Assigning to an array element.
- Setting the value of the field of a composite type.
- Assigning to local variables with a type annotation.
- Returning from a function with a type annotation.

Let us look at some examples demonstrating implicit conversion.

```
julia> values = Int8[3, 5]
2-element Array{Int8,1}:
 3
 5

julia> typeof(values[2])
Int8

julia> x = 42
42
```

```
julia> typeof(x)
Int64

julia> values[2] = x # convert(Int8, 42)
42
```

Setting the field of a composite type defined with `struct`:

```
julia> mutable struct Point
           x::Float64
           y::Float64
       end

julia> p = Point(3.5, 6.8)
Point(3.5, 6.8)

julia> p.x = Int8(10)    # convert(Float64, Int8(10))
10
```

Here we add a type annotation to a function to make sure the return value is of a certain type. If it is not, a conversion is attempted with `convert`.

```
julia> foo(x::Int64) :: UInt8 = 2x
foo (generic function with 1 method)

julia> y = foo(42)
0x54

julia> typeof(y)
UInt8
```

Next we will get more into the details of how conversion and promotion is done using a larger code example.

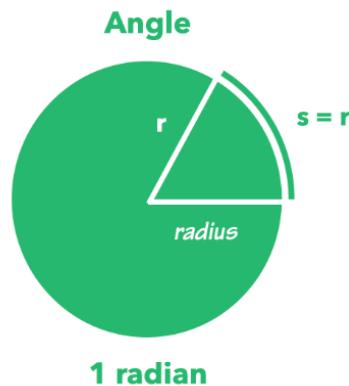
## Defining Custom Units for Angles

A lot of calculations within science can easily go wrong if we mix up units. For instance in the petroleum industry, mixing feet and meters is easy because coordinates of an oil well is usually given in meters while the depth of the well is given in feet.

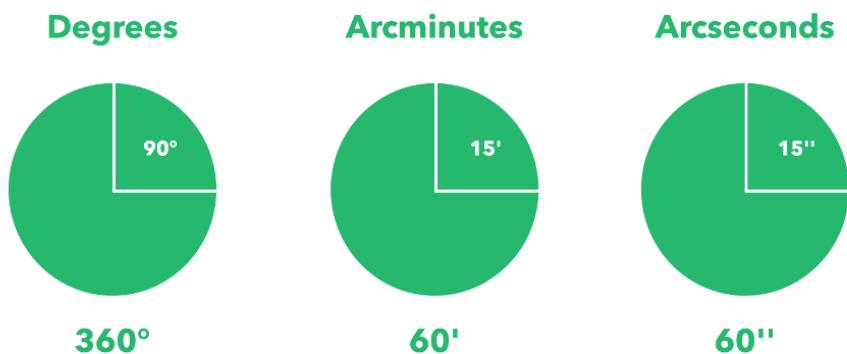
A famous example is the Mars Climate Orbiter, a robotic space probe launched by NASA, which was lost due to NASA and Lockheed using different units of measure. NASA was using metric units and Lockheed used US customary units, such as feet and pound. Thus there is an advantage in designing code where one does not accidentally mixup units.

In this example we will demonstrate working with different units for angles. In mathematics angles are usually given as radians, while people navigating using maps will tend to use degrees. When using degrees we split up the circle in 360 degrees. One degree is thus 1/360th of the circumference of that circle.

With radians in contrast we deal with how many times the radius of a circle is duplicated along the circumference to get that angle. So 1 radian is the angle you get when you mark off a distance along the circumference equal to the radius of the circle.



Degrees in contrast are more strongly tied to navigation, in particular celestial navigation. Each day the earth moves about 1 degree around the sun, since the year is made up of 365 days. An angle is further divided into 60 arcminutes, and an arcminute is divided into 60 arcseconds.



Actually you can work with both metric degrees and degrees-minutes-seconds (DMS), but we are working with DMS here to keep things interesting.

```
abstract type Angle end
```

```
struct Radian <: Angle
    radians::Float64
end
```

```
struct DMS <: Angle
    seconds::Int
end
```

Similar to our rocket example, we have defined an abstract type `Angle`, which

all our concrete angle units are subtypes of. The benefits of this will become clear later.

## Constructors

That degrees-minutes-seconds are stored as seconds should be regarded as an implementation detail and not exposed to the user. Hence users should not use that constructor directly. Instead we will define more natural constructors:

```
Degree(degrees::Integer) = Minute(degrees * 60)
Degree(deg::Integer, min::Integer) = Degree(deg) + Minute(min)
Degree(deg::Integer, min::Integer, secs::Integer) = Degree(deg, min) + Second(secs)

function Minute(minutes::Integer)
    DMS(minutes * 60)
end

function Second(seconds::Integer)
    DMS(seconds)
end
```

## Arithmetic

To be able to actually run these constructors we need to be able to add together DMS numbers. The code snippet `Degree(deg) + Minute(min)` basically does a `DMS(deg, 0, 0) + DMS(0, min, 0)`. However `+` operator has not been defined for DMS types. Nor have we defined them for radians so lets do both:

```
import Base: -, +
+(θ::DMS, α::DMS) = DMS(θ.seconds + α.seconds)
-(θ::DMS, α::DMS) = DMS(θ.seconds - α.seconds)

+(θ::Radian, α::Radian) = Radian(θ.radians + α.radians)
-(θ::Radian, α::Radian) = Radian(θ.radians - α.radians)
```

Let me clarify how this works. As discussed in section Multiple Dispatch, defining a method in Julia will automatically create a function if no corresponding function already exists. E.g. if the `+` function is not imported, Julia will not know that it already exist when you define `+ methods`.

Thus Julia will create an entirely new `+` function and attach your angle specific methods to it. If you then try to evaluate `3 + 4`, Julia will attempt a lookup of matching methods on this newly defined `+` function. But it has no methods dealing with regular numbers, only for angles. Hence you get an error:

```
julia> 3 + 4
ERROR: MethodError: no method matching +(::Int64, ::Int64)
You may have intended to import Base.+
```

Essentially you end up shadowing the existing `+` function and its attached methods. By doing `import`, we are essentially telling Julia that we want to add methods to existing functions, not create new ones.

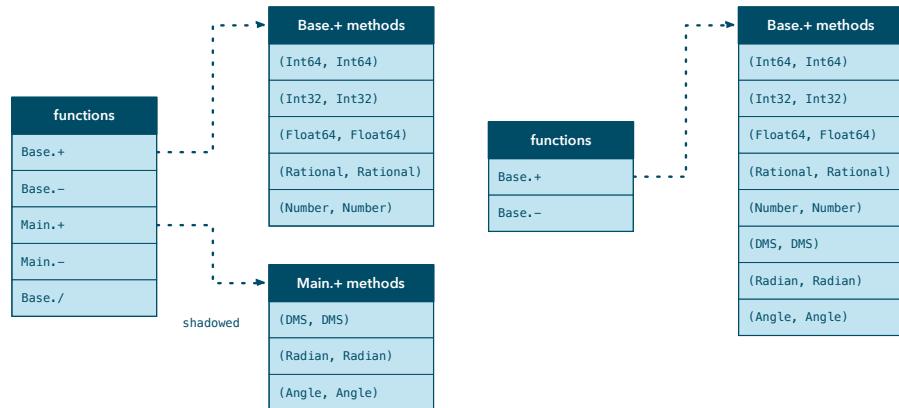


Figure 32: On the left we have not imported the + and - functions. On the right they have been imported.

## Accessors

Given an angle in DMS, we want to be able to know the degree, minute and seconds part.

```
function degrees(dms::DMS)
    minutes = dms.seconds ÷ 60
    minutes ÷ 60
end

function minutes(dms::DMS)
    minutes = dms.seconds ÷ 60
    minutes % 60
end

seconds(dms::DMS) = dms.seconds % 60
```

We can use these functions to provide a custom display of these angles on the Julia REPL. To write that the angle is 90 degrees, 30 arcminutes and 45 arcseconds we would use the notation  $90^\circ 30' 45''$ .

## Displaying Degrees-Minutes-Seconds Angles

If you use the constructors now, the display you get out of the box isn't very good. It exposes the internal representation of DMS degrees as being made up of arcseconds.

```
julia> α = Degree(90, 30, 45)
DMS(325845)

julia> degrees(α)
90
```

```
julia> minutes(α)
30

julia> seconds(α)
45

julia> β = Degree(90, 30) + Degree(90, 30)
DMS(651600)

julia> degrees(β)
181

julia> minutes(β)
0
```

We can define an alternative view by adding a method to the Julia `show` function. The Julie REPL environment uses the `show(io::IO, data)` to display data of some specific type to the user. Remember in Julia you can define methods to work on generic abstract types. However you can add methods dealing with more concrete types. That is what we want to do.

```
import Base: show

function show(io::IO, dms::DMS)
    print(io, degrees(dms), "° ", minutes(dms), "' ", seconds(dms), "''")
end

function show(io::IO, rad::Radian)
    print(io, rad.radians, "rad")
end
```

#### NOTE

The implementation uses the `print` function which we have used earlier. The difference is that now we provide an `io` object as first argument. When you write `print("hello world")` it is the equivalent of writing `print(stdout, "hello world")`. The `io` object could represent a file, a network connection or your command line console.

We will look more closely at `show` and `IO` objects in later chapters when we cover text Strings in detail. But for now, this gives a nice way of looking at DMS angles.

```
julia> α = Degree(90, 30, 45)
90° 30' 45"

julia> β = Degree(90, 30) + Degree(90, 30)
181° 0' 0"
```

## Type Conversions

Now that we got the basics in place, we want to be able to do something useful with these angles. You would want to use them with functions such as `sin` and `cos` but these just take plain numbers which are radians. We need to define conversions so that DMS angles can be turned into radians.

```
import Base: convert

Radian(dms::DMS) = Radian(deg2rad(dms.seconds/3600))
Degree(rad::Radian) = DMS(floor(Int, rad2deg(rad.radians) * 3600))

convert(::Type{Radian}, dms::DMS) = Radian(dms)
convert(::Type{DMS}, rad::Radian) = DMS(rad)
```

This contains a number of new things we need to discuss more in detail to make it clear how it works. Let us do the simplest first. Usually we have defined functions like this `f(x::Int)`, but if you don't actually intend to use the `x` argument for anything, you can simply write `f(::Int)`. What is the point of that? It allows you to define different code which should be run depending on the type of the argument.

With these conversions in place we can implement versions of `sin` and `cos` which take numbers with units as arguments.

```
sin(rad::Radian) = Base.sin(rad.radians)
cos(rad::Radian) = Base.cos(rad.radians)

sin(dms::DMS) = sin(Radian(dms))
cos(dms::DMS) = cos(Radian(dms))
```

In this case we are not importing `sin` and `base` before creating the methods. The reason is because we actually want to shadow the ``real'' `sin` and `base` functions since we don't want people to accidentally call these functions with plain numbers. We want them use radians or degrees explicitly.

```
julia> sin(π/2)
ERROR: MethodError: no method matching sin(::Float64)
You may have intended to import Base.sin

julia> sin(90)
ERROR: MethodError: no method matching sin(::Int64)
You may have intended to import Base.sin

julia> sin(Degree(90))
1.0

julia> sin(Radian(π/2))
1.0
```

Now you cannot accidentally use a number you haven't at some point stated whether should be degrees or radians with trigonometric functions.

## Pretty Literals

This is quite nice, but it would look a lot better if you could write `sin(90°)` instead of `sin(Degree(90))` and `sin(1.5rad)` instead of `sin(Radian(1.5))`.

In fact we can achieve this. Observe that Julia interprets `1.5rad` as `1.5*rad`. Thus by defining multiplication of regular scalars with units of degrees or radians we have magically solved the problem.

```
import Base: *, /

*(coeff::Number, dms::DMS) = DMS(coeff * dms.seconds)
*(dms::DMS, coeff::Number) = coeff * dms
/(dms::DMS, denom::Number) = DMS(dms.seconds/denom)

*(coeff::Number, rad::Radian) = Radian(coeff * rad.radians)
*(rad::Radian, coeff::Number) = coeff * rad
/(rad::Radian, denom::Number) = Radian(rad.radians/denom)

const ° = Degree(1)
const rad = Radian(1)
```

The last two lines show the secret sauce. It means `90°` is read by Julia as `90 * Degree(1)` which when computed will result in `Degree(90)`.

```
julia> sin(90°)
1.0

julia> sin(1.5rad)
0.9974949866040544

julia> cos(30°)
0.8660254037844387

julia> cos(90°/3)
0.8660254037844387

julia> sin(3rad/2)
0.9974949866040544
```

## Type Promotions

The simple but labour intensive way of adding support for doing arithmetic with different angle units would mean defining lots of functions with all possible combinations, like this:

```
+(<math>\alpha</math>::DMS, <math>\beta</math>::Radian) = Radian(<math>\alpha</math>) + <math>\beta</math>
+(<math>\alpha</math>::MetricDegree, <math>\beta</math>::DMS) = <math>\alpha</math> + MetricDegree(<math>\beta</math>)
+(<math>\alpha</math>::Radian, <math>\beta</math>::MetricDegree) = <math>\alpha</math> + Radian(<math>\beta</math>)
```

But as you can imagine this quickly causes an unmanageable explosion in possible permutations. Thus instead we define generic functions for different units like this.

```
+(@::Angle, α::Angle) = +(promote(θ, α)...)  
-(θ::Angle, α::Angle) = -(promote(θ, α)...)
```

The only remaining problem is that we have not told `promote` how to promote angle types. It only knows about `Number` types. A first guess, of how to add the temperature type, would be to add another `promote` method, but that is not how it works. Instead `promote` does its job by calling a function called `promote_rule`. We need to register our types by defining `promote_rule` methods for *our* types. This will look like:

```
import Base: promote_rule  
  
promote_rule(::Type{Radian}, ::Type{DMS}) = Radian
```

These methods are unusual, as all the arguments are type objects. Nor have we given a name to any of the arguments, because the type objects are not used for anything but getting multiple dispatch to select the correct method of the `promote_rule` function.

As demonstrated below, `promote_rule` is a function taking two type objects as arguments and returns another type object:

```
julia> promote_rule(Int16, UInt8)  
Int16  
  
julia> promote_rule(Float64, UInt8)  
Float64  
  
julia> promote_rule(Radian, DMS)  
Radian
```

This defines the promotion rule: Given two different types, what type should they all be promoted to? Now we have put all the pieces in place. We have plugged into the Julia convert and promote machinery, by implementing methods for the `convert` and `promote_rule` functions.

```
julia> sin(90° + 3.14rad/2)  
0.0007963267107331024
```

```
julia> cos(90° + 3.14rad/2)  
-0.9999996829318346
```

```
julia> 45° + 45°  
90° 0' 0''
```

```
julia> Radian(45° + 45°)  
1.5707963267948966rad
```

```
julia> 45° + 3.14rad/4  
1.5703981633974484rad
```

This example gives a hint of the advantages of using a multiple-dispatch language such as Julia. Implementing this behavior using object-oriented programming is harder and gets increasingly hard as you add more types into the

mix. If you defined each angle as a class, you would have to have several methods for every operation. One for each type.

And there are more practical problems with the object-oriented approach. Should you ever need to add another angle unit, this will require:

1. Adding another class with 4 methods for each operator.
2. Modify every other angle class, including the base class `Angle`, by adding a version of each operator handling that particular type.
3. Add another constructor in each class to handle the new angle unit.

This obviously does not scale, and it breaks the open-close principle in object-oriented programming:

Open for extension, closed for modification.

If angle units were provided as a library you could not extend it with other units, without modifying the library itself. That is obviously impractical.

Julia elegantly solves this by *not* making functions a part of the types. Thus you can add new constructor functions to a type without modifying the type definition itself. Adding `convert` and `promote_rule` functions does not require modification of the library providing the types which you seek to define promotion rules and conversion for.



# Different Kinds of Nothing

- **Undefined** values. Values which should exist but which have not yet been defined.
- **Nothing** when a value isn't there at all. For instance when you search for something and don't find it.
- **Missing** values. Values which exist in the real world, but we simply don't know what they are. Their measurements are missing.

An important thing to deal with in any programming language is to have a way of representing the absence of a value. For a long time most mainstream programming languages such as C/C++, Java, C#, Python and Ruby would have a value called `null` or `nil` which is what a variable would contain if it did not have any value. Or more accurately phrased: If the variable was not bound to a concrete object.

When would this be useful? Say you write a function to search a text string for a the substring ``foo.'' If the string does not contain ``foo,'' we must have a way of signaling this. One could raise an error, but that seems inappropriate given that a substring missing, is entirely expected in most cases. Not something which should be dealt with as an error.

So languages like C/C++ would use `NULL` or `nullptr` to indicate this. However it is not without reason its inventor British computer scientist Tony Hoare, called the null pointer his billion dollar mistake.

It makes it difficult to write safe code, because at any given time any variable could be null. This means code in languages supporting null objects have a lot of code which does nothing but check whether a variable is null or not before carrying on.

For this reason modern languages have tended to avoid having null objects or pointers. Julia does not have a generic null object or pointer.

## The Nothing Object

The closest things to null which Julia has is the `nothing` object. It is a simple concrete type defined in Julia as:

```
struct Nothing  
end  
const nothing = Nothing()
```

The `nothing` object is an instance of the type `Nothing`. However every instance of `Nothing` is the same object. You can test that yourself in the REPL:

```
julia> something = Nothing()

julia> nothing == something
true

julia> Nothing() == Nothing()
true
```

However there is nothing magical going on here. Rather **every** type in Julia which has zero fields are singletons. That means their constructor function will always return the same object. We can easily prove this by making our own singleton type.

```
julia> struct Foobar end

julia> foo = Foobar()
Foobar()

julia> bar = Foobar()
Foobar()

julia> bar == foo
true

julia> Foobar() == Foobar()
true
```

This makes it easy to make special purpose objects in Julia which you want to assign special meaning. `nothing` in Julia is by convention used when a function could not find something.

```
julia> findfirst("four", "one two three four")
15:18

julia> findfirst("four", "one two three")

julia> typeof(ans)
Nothing

julia> findfirst("four", "one two three") == nothing
true
```

For data structures such as trees and lists it is also useful to have null objects. For instance we want to be able to terminate a chain of objects in some manner. Consider this modified example of a rocket type:

```
mutable struct Rocket
    nextstage::Rocket
    tank::Tank
    engine::Engine
```

```
end
```

We are creating a data structure to represent a multistage rocket. Each rocket points to the rocket which represent the next stage.

```
julia> merlin = Engine(845e3, 282, 470)
SingleEngine(845000.0, 282.0, 470.0)
```

```
julia> tank = Tank(4e3, 111.5e3)
Tank(4000.0, 111500.0, 107500.0)
```

```
julia> second = Rocket(nothing, tank, merlin)
ERROR: MethodError: Cannot `convert` an object of type Nothing to an object of type Rocket
```

Here you see difference between the usage of `nothing` in Julia and in e.g. Java. In the latter you can always assign `null` to an object of any type except primitives. Julia complains here, and rightly so, that `nothing` is not a `Rocket` type and hence you cannot assign it to the `nextstage` field.

So what do we do if we specifically want `nextstage` to hold either a `Rocket` object or `nothing`? To understand how to do that, we need an introduction to union types.

## Union Types

Union types allow us to combine two or more types into one type. Say you have types named `T1`, `T2`, and `T3`. You can create a union of these types by writing `Union{T1, T2, T3}`. This creates a new type which can be a placeholder for any of those types. This means if you wrote a method with the signature `f(x::Union{T1, T2, T3})`, then this particular method would get called whenever `x` was of type `T1`, `T2` or `T3`.

Let us look at a concrete example:

```
julia> f(x::Union{Int, String}) = x^3
f (generic function with 1 method)

julia> f(3)
27

julia> f(" hello ")
" hello  hello  hello "

julia> f(0.42)
ERROR: MethodError: no method matching f(::Float64)
Closest candidates are:
  f(!Matched)::Union{Int64, String}) at none:1
```

The last example fails because `x` is a floating point number, and we have only define a method for function `f` taking a union of `Int` and `String`. `Float64` is not included.

Every type included in a type union will be counted as a subtype of that union. We use the `<:` to either define a subtype or test if a type is a subtype.

```
julia> String <: Union{Int64, String}
true

julia> Int64 <: Union{Int64, String}
true

julia> Float64 <: Union{Int64, String}
false
```

Perhaps you have guessed already. The solution to our Rocket problem is to make nextstage union type.

```
mutable struct Rocket
    nextstage::Union{Rocket, Nothing}
    tank::Tank
    engine::Engine
end
```

Now we are able to make a multistage rocket which has an end.

```
julia> merlin = Engine(845e3, 282, 470)
SingleEngine(845000.0, 282.0, 470.0)

julia> tank = Tank(4e3, 111.5e3)
Tank(4000.0, 111500.0, 107500.0)

julia> second = Rocket(nothing, tank, merlin)
Rocket(nothing, Tank(4000.0, 111500.0, 107500.0), SingleEngine(845000.0, 282.0, 470.0)

julia> first = Rocket(second, tank, EngineCluster(merlin, 9))
Rocket(Rocket(nothing, Tank(4000.0, 111500.0, 107500.0), SingleEngine(845000.0, 282.0,
```

## Missing Values

Missing values are represented in Julia with the `missing` object which is of type `Missing`. This seems very similar to `nothing`, so why do we need it?

This comes about because Julia aims to be a good language for academics doing scientific computing, statistics, big data etc. In statistics missing data is an important concept. It happens all the time because in almost any data collection for statistics there will be missing data. For instance you may have participants filling out forms and some of them fail to fill out all the fields.

Some participants may leave a study before it is finished, leaving those who conduct the experiment with incomplete data. Missing data can also exist due to errors in data entry. So unlike the concept of `nothing` missing data actually exists out there in the real world. We simply don't know what it is.

Specialized software for statisticians such as R and SAS has long time ago established that missing data should propagate rather than throw exceptions. Julia has chosen to follow this convention as well. Let us look at what that means in practice:

```
julia> missing < 10
missing

julia> nothing < 10
ERROR: MethodError: no method matching isless(::Nothing, ::Int64)

julia> 10 + missing
missing

julia> 10 + nothing
ERROR: MethodError: no method matching +(::Int64, ::Nothing)
```

This pattern is repeated for most arithmetic and boolean operators in Julia. The rational for this is that a lot of serious mistakes have been made in statistical work in the past from not catching that there are missing values. Since `missing` spreads like a virus in Julia, an unhandled `missing` value will quickly get caught.

Missing can be handled explicitly. For instance if we want to calculate the sum or averages of an array which may contain missing values, we can use the `skipmissing` function to avoid attempting to include missing values in the result.

```
julia> sum(skipmissing([2, 4, 8]))
14

julia> mean(skipmissing([2, 4, 8]))
ERROR: UndefVarError: mean not defined

julia> median(skipmissing([2, 4, 8]))
ERROR: UndefVarError: median not defined
```

## Not a Number

Somewhat related to missing values is the floating point number `NaN` (not a number). We get `NaN` as a result when result of an operation is undefined. This typically pops up as an issue when dividing by zero.

```
julia> 0/0
NaN

julia> 1/0
Inf

julia> -1/0
-Inf
```

In this case `Inf` stands for infinity and is what you get when dividing a number different from zero by zero. This makes some sense. As the divisor approaches zero, the result tends to grow larger.

It is tempting to consider `NaN` as similar to `missing` and that they are interchangeable. After all `NaN` also propagates through all calculations:

```
julia> NaN + 10
NaN
```

```
julia> NaN/4
NaN
```

```
julia> NaN < 10
false
```

```
julia> NaN > 10
false
```

Although comparisons return false. Here is why you should **not** use NaN for missing values: If you have a mistake in your algorithm, that causes a  $0/0$  to happen, you will produce NaN. This will be indistinguishable from having missing values as input. You may falsely believe your algorithm is working because it is removing missing values in the calculation, thus masking a defect in your algorithm.

## Undefined Data

Undefined data is something you rarely encounter in Julia but it is worth being aware of it. It happens when a variable or field of a struct has not been set. Usually Julia tries to be smart about this. Thus if you define a struct with number fields, Julia will automatically initialize them to zero, if you don't do anything.

However if you define a struct without telling Julia what the type of its fields are, Julia has no way of guessing what the fields should be initialized to.

```
julia> struct Foobar
        alpha
        beta
        Foobar() = new()
    end

julia> foo = Foobar()
Foobar(#undef, #undef)

julia> foo.alpha
ERROR: UndefRefError: access to undefined reference
```

## Summary

Keeping these different concepts of nothing apart can be a bit daunting, so let me summarize briefly the differences. `nothing` is the *programmers* kind of null. It is what a programmer wants when something does not exist.

`missing` is the *statisticians* type of null. What they want when a value is missing in their input data.

`NaN` is to indicate that somewhere in your code there was an illegal math operation. In other words this has to do with calculations and not with statistical collection of data.

`Undefined` is when you, the programmer, forgot to do your job and initialize all used data. Most likely it points to a bug in your program.

As a final reminder: Julia does *not* have `null` in the common sense because you need to explicitly allow for a *nothing* value using type unions. Otherwise a function argument cannot accidentally pass a *nothing* value.



# Strings

- **Unicode, code points and UTF8.** How strings are stored in memory. Byte indices vs character indices. Different ways of representing unicode.
- Common **string operations**, how to compare strings, convert them to lowercase and uppercase.
- **Raw strings** when do you need them.
- Different types of **string literals**: Regular expressions, MIME types, BigInt literals.

We have looked at working with text string in practice. However to correctly use text strings there are a lot of details worth knowing about. In this chapter we will examine these details more closely. As long as you are working with letters from A-Z, things are simple. However there are a multitude of languages in the world with their own unique set of characters which Julia needs to be able to deal with.

That means a minimal required knowledge to work effectively with Julia strings requires some knowledge of unicode. Unicode is the international standard for mapping numbers (code points) to characters.

Julia has support for special string literals with alternative uses. There are special strings for representing regular expressions, which are a form of text string for representing patterns of text.

## UTF-8 and Unicode

Text strings in Julia are unicode, encoded in UTF-8 format. What does that mean, and should you even care? Let me walk you through a simple example to motivate your need to understand unicode better.

``Æser," is the plural of norse gods in Norwegian. It is a 4 letter word as confirmed with the `length` function:

```
julia> length("Æser")
4
```

But when attempting to access individual characters in the word, you will notice something strange.

```
julia> "Æser"[1]
'Æ': Unicode U+00C6 (category Lu: Letter, uppercase)
```

```
julia> "Æser"[2]
ERROR: StringIndexError("Æser", 2)

julia> "Æser"[3]
's': ASCII/Unicode U+0073 (category Ll: Letter, lowercase)

julia> "Æser"[4]
'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
```

You can see that trying to get the second character at index 2, causes an error (an exception is thrown). The second character happens to be at index 3.

How about another word? ``Þrúðvangr," is the name of the realm of the norse god Thor.

```
julia> length("Þrúðvangr")
9

julia> "Þrúðvangr"[9]
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

It is a 9 character word, but the character at index 9 is the 6th character 'a'. What is going on here? To understand that, you need to understand unicode and how Julia strings support it through the UTF-8 encoding.

```
julia> sizeof("Æser")
5

julia> sizeof("Þrúðvangr")
12
```

In UTF-8 every character is encoded into 1-4 bytes. Normal letters such as A, B and C will take just one byte. While letters such as Æ, Þ and ð which are not used in the english language, will typically require more than 1 byte to encode.

However before we delve further into how Julia deals with this, it is useful to understand some key concepts in unicode, which are not specific to the Julia programming language.

## Code Points and Code Units

Let us take a few steps back in time to understand how we ended up with UTF-8. Originally the ASCII standard was developed in the US back in the 1960s. It gave a number for each character of the latin alphabet used in English. A-Z would be numbered from 65 to 90 and a-z from 97 to 122. With ASCII all of this was encoded into 1 byte for each character.

To deal with different languages, one would operate with different interpretations of these numbers from 1 to 255. However this quickly became impractical. You could not e.g. mix text written using different alphabets on the same page. The solution was unicode, which aimed to give a unique number to every character in the world, not just those in the latin alphabet but also for Cyrillic, Chinese, Thai and all the various Japanese character sets.

The number given to each character is called a code point in unicode terminology. Originally one believed 16 bits would be enough to store every unicode code point. 16 bits gives  $2^{16} - 1 = 65535$  unique numbers. Thus one of the first unicode encodings UCS used 16 bits (2 bytes) to encode every unicode code point (character).

Later it was determined that this would not be enough and we would need 4 bytes (32 bits) to encode every possible unicode character. At this point the UCS approach started to look flawed:

1. UCS was already incompatible with a large array of software written for ASCII assuming every character was just 1 byte.
2. Spending 4 bytes to represent each character would mean every typical english text dominating the internet would require four times the storage and processing.

UTF-8 encoding aimed to solve this problem. Regular ASCII characters were encoded as before with just 1 byte. Hence UTF-8 offers backwards compatibility. Special values in the first byte would signal that a character is encoded in more than 1 byte.



At this point it makes sense to introduce the concept of a unicode **code unit**. Code units are the pieces used to encode a code point. Until UTF-8 arrived, a code point and a code unit was pretty much the same thing, a 16 bit value. In UTF-8 the **code unit** is just one byte long. So in UTF-8 a code point is made up of 1-4 code units.

This is how the various concepts are related: A *glyph* is what you see on the screen. Basically the font for one character. A glyph gives different appearance to the same character (code point). So each code point can have multiple glyphs to visually represent it.

Likewise there are a multitude of ways of encoding a code point. Encoding is how you store the code point. A code point is a plain number. It says nothing about how many bytes (code units) you are going to use to store it.

If this seems too abstract, we can explore these relationships in the Julia REPL environment.

```
julia> codepoint('A')
0x00000041
```

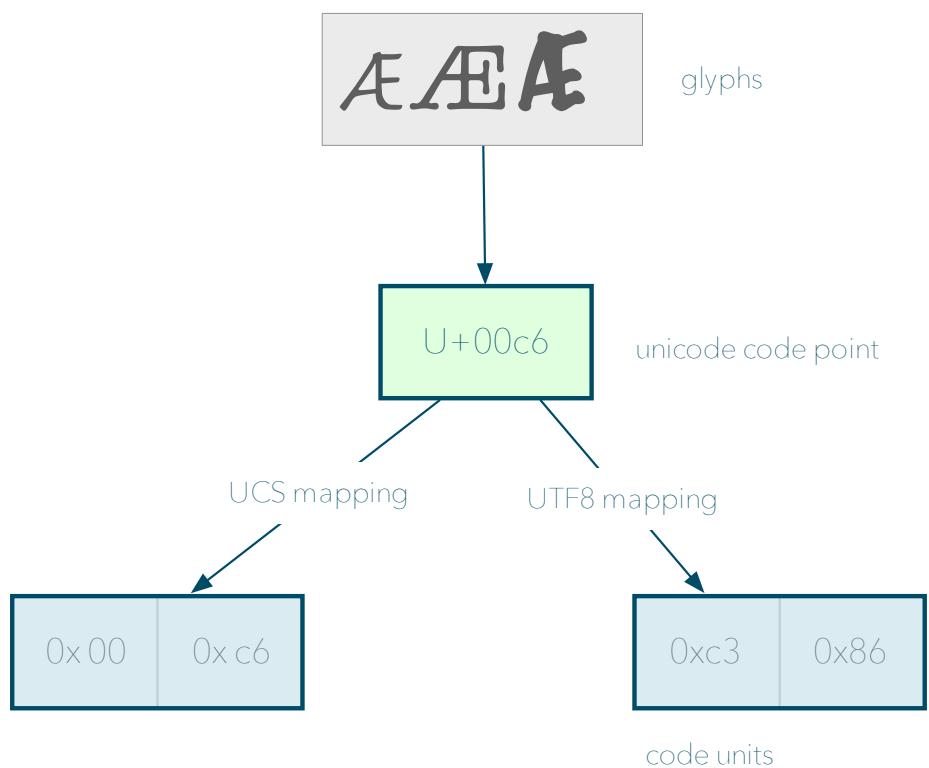


Figure 33: unicode

```
julia> Int(codepoint('A'))
65

julia> ncodeunits('A')
1

julia> isascii('A')
true
```

Let us explore characters which are not part of the original ASCII standard. They should have more than one code unit and not return true when `isascii()` is called.

```
julia> codepoint('Æ')
0x000000c6

julia> ncodeunits('Æ')
2

julia> isascii('Æ')
false

julia> codepoint('☺')
0x0001f60f

julia> ncodeunits('☺')
4

julia> isascii('☺')
false
```

Just typing letters at the REPL will also give you useful information, when the character literal is evaluated.

```
julia> 'A'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> 'Æ'
'Æ': Unicode U+00C6 (category Lu: Letter, uppercase)

julia> '☺'
'☺': Unicode U+1F60F (category So: Symbol, other)
```

Notice how it tells us what the unicode code point number is.

**NOTE How to quickly write unusual characters**

You use backlash \ and tab key to easily write unusual characters not present on your keyboard. For instance to write ☺, you would type \:smirk: in the Julia REPL and hit tab, to get a

completion.

You can even hit tab after writing just \: to get a full list of possible emojis. Norwegian letters such as ÅØÅ that I occasionally use in my examples, can easily be written on a Mac by simply holding down the option key and hitting '0A respectively.

Unicode code points can be written explicitly in Julia in various ways:

```
julia> '\U41'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> Char(0x41)
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> '\U00c6'
'Æ': Unicode U+00C6 (category Lu: Letter, uppercase)

julia> Char(0xc6)
'Æ': Unicode U+00C6 (category Lu: Letter, uppercase)

julia> '\U01f60f'
'⌚': Unicode U+1F60F (category So: Symbol, other)

julia> Char(ox01f60f)
'⌚': Unicode U+1F60F (category So: Symbol, other)
```

You can combine these with `map` to create various ranges. For instance a range does not need to be merely written as numbers. '`'A':'F'`' is a perfectly valid range.

```
julia> map(lowercase, 'A':'F')
6-element Array{Char,1}:
'a'
'b'
'c'
'd'
'e'
'f'

julia> map(codepoint, 'A':'F')
6-element Array{UInt32,1}:
0x00000041
0x00000042
0x00000043
0x00000044
0x00000045
0x00000046
```

And we can of course go the opposite direction:

```
julia> map(Char, 65:70)
6-element Array{Char,1}:
 'A'
 'B'
 'C'
 'D'
 'E'
 'F'
```

## String Operations

Working with text is such a common thing to do, that it pays to be aware of the possibilities that exist in the language. My intention is not to show every single string operation that exists but give an idea of what is possible.

I tend to use Julia a lot as an assistant when working with other programming languages. I would use Julia for transforming code in different ways.

Let me walk you through an example of doing this. In many programming languages it is common to see these sorts of variations in text formatting of identifiers:

- **FooBar**. Pascal case (upper camel case) frequently used style for types or classes. Sometimes used for constants.
- **foo\_bar**. Snake case often used for the name of variables, methods and functions.
- **fooBar**. Camel case (lower camel case) frequently used for methods and variable names.
- **FOO\_BAR**. Upper snake case, often used for constants and enum values.
- **foo-bar**. Kebab case. You will find this in LISP programs and configuration files.

We will look at ways of converting between these styles and how you can turn this into handy utility functions for aiding your programming.

Here is my typical process for developing a simple function to do something. Since you are not certain about how an unfamiliar function works, you try it out. Then you gradually combine it with more function calls to get what you want. Eventually you have enough to implement your function.

```
julia> s = "foo_bar"
"foo_bar"

julia> split(s, '_')
2-element Array{SubString{String},1}:
 "foo"
 "bar"

julia> uppercasefirst("foo")
"Foo"

julia> map(uppercasefirst, split(s, '_'))
```

```

2-element Array{String,1}:
"Foo"
"Bar"

julia> join(["Foo", "Bar"])
"FooBar"

julia> join(map(uppercasefirst, split(s, '_')))
"FooBar"

julia> camel_case(s::AbstractString) = join(map(uppercasefirst, split(s, '_')))
camel_case (generic function with 1 method)

```

Now we got a function that will do the conversion, but often we want to be able to do this quickly. We select some text in our code editor which is in snake case, and which we want to turn into camel case and paste back in.

This is where Julia's `clipboard()` function becomes handy. It can both read from and write to the clipboard. The clipboard is what we call the place where everything you copy-paste reside.

We add a method to our `camel_case` function, which does not take any string arguments, but instead reads the clipboard. Just mark some text and copy it before running `clipboard()`. I marked the first part of this paragraph.

```

julia> s = clipboard()
"We add a method to our"

julia> function camel_case()
    s = camel_case(clipboard())
    clipboard(s)
    s
end

```

`clipboard()` will get the contents of the clipboard while `clipboard(s)` stores the content of `s` on the clipboard. Whenever you are coding and want to change a snake case text to camel case, you can follow these steps:

1. Copy the text.
2. Switch to your open Julia REPL.
3. Start typing `came....` and hit up-arrow ↑, this should complete to `camel_case()` if you called it before. Alternatively hit tab.
4. Go back to editor and paste the result.

**NOTE Keys to make you more productive**

To work quickly with Julia it is important to become accustomed to all the hotkeys. ↑ key is used to quickly search through your history. If you start writing a few letter first it will filter that history to only match history beginning with those first letters.

The tab key is used to complete a word matching a function Ju-

lia knows about. That could be a built in one or one you have defined yourself.

`Ctrl+A` and `Ctrl+E` is used to jump to the beginning and end of a line in the Julia REPL. Say you just wrote:

```
map(uppercasefirst, split(s, '_'))
```

And you want to alter this to:

```
join(map(uppercasefirst, split(s, '_')))
```

Hit `↑`, to get back the line you just wrote. Hit `Ctrl+A`, to jump to beginning of line. Write `join()`. Finally hit `Ctrl+E` to jump to the end and write a `)`

## Camel Case to Snake Case

Let us look at code for going the other direction. In this case using the `split` function will not work. Let us look at why. In our case we cannot split on a specific character, however `split` can take functions instead of characters to decide where to split. To split on whitespace we would use `split(s, isspace)`. So we could try to use the `isupper` function. It checks whether a character is uppercase or not. That is useful since we split where characters are uppercase.

```
julia> isupper('a')
ERROR: UndefVarError: isupper not defined
```

```
julia> isupper('A')
ERROR: UndefVarError: isupper not defined
```

```
julia> s = "oneTwoThreeFour"
"oneTwoThreeFour"
```

```
julia> split(s, isuppercase)
4-element Array{SubString{String},1}:
 "one"
 "wo"
 "hree"
 "our"
```

As you can see this does not work, because `split` strips away the character we use for splitting. Instead what we will use is one of Julia's many find functions. If you write `find` in the REPL and hit tab, you will see a number of possible choices.

```
julia> find
findall    findfirst   findlast
findmax    findmax!    findmin
findmin!   findnext   findprev
```

`findfirst` finds the first occurrence of a match, while `findall` finds all of them.

Let's look at an example to clarify.

```
julia> findfirst(isspace, s)

julia> indices = findall(isspace, s)
0-element Array{Int64,1}

julia> s[indices]
""
```

We can loop over all the indices of the uppercase letters and capture the substrings using ranges.

```
function snake_case(s::AbstractString)
    i = 1
    for j in findall(isuppercase, s)
        println(s[i:j-1])
        i = j
    end
    println(s[i:end])
end
```

This is just a demonstration of how we gradually develop the function. In this case we are using `println` to make sure we are getting the correct output. You can see we are using ranges `i:j-1` to extract a substring.

```
julia> snake_case("oneTwoThreeFour")
one
Two
Three
Four
```

Here is a complete example. We have removed the `println` and added an array of strings called `words` to store each individual word that is capitalized.

```
function snake_case(s::AbstractString)
    words = String[]
    i = 1
    for j in findall(isuppercase, s)
        push!(words, lowercase(s[i:j-1]))
        i = j
    end
    push!(words, lowercase(s[i:end]))
    join(words, '_')
end
```

Once we have collected the words in the array we join them into one string using `join(words, '_')`. The second argument '`_`' causes each word to be joined with `_` as a separator.

## Converting between numbers and strings

In the Working with Text chapter we got input from the user. Whether input comes from the keyboard or a file, it usually comes in the form of text strings.

However you may need the input numbers.

We looked at the `parse` function to deal with this. Let us look at it more in detail. You provide a type-object as first argument, to specify what sort of number type you want to parse to. This could be anything from different types of integers to floating point numbers.

```
julia> parse(UInt8, "42")
0x2a

julia> parse(Int16, "42")
42

julia> parse(Float64, "0.42")
0.42
```

We can even specify the base. Julia assumes base 10 as default when parsing numbers. By that we means digits running from 0 to 9. However we could parse the numbers as if they were binary, base 2, if we wanted. That assumes we only have the digits 0 and 1 to form numbers.

```
julia> parse(Int, "101")
101

julia> parse(Int, "101", base=2)
5
```

Or how about hexadecimal?

```
julia> parse(Int, "101", base=16)
257
```

These conversions can also be done in reverse. You can take a number and decide what base you want to use when converting to a text string.

```
string(5, base=2)
string(17, base=16)
string(17, base=10)
```

From the previous string chapter you may remember the named argument `color=:green`. Here we are using a named argument `base=2` again. This is also a typical case, because you are specifying something which only occasionally needs to be specified.

## String interpolation and concatenation

Strings can be combined in a myriad of ways in Julia. We will compare some different ways of doing it. Often we have objects such as numbers we want to turn into text strings. So we define some variables of different types to use in our string examples.

```
julia> engine = "RD-180"
"RD-180"
```



Figure 34: RD-180 rocket engine

```
julia> company = "Energomash"
"Energomash"

julia> thrust = 3830
3830

julia> string("The ", engine,
   " rocket engine, produced by ",
   company,
   " produces ", thrust,
   " kN of thrust")

"The RD-180 rocket engine, produced
by Energomash produces 3830 kN of thrust"
```

Here we used the `string()` function to perform concatenation of strings and converting non-strings to strings. We could also use the string concatenation operator `*`. If you come from other languages you may be more familiar with `+` operator being used for string concatenation. Part of the rational for the `*` operator is that it makes more sense from a mathematical sense, and remember Julia was designed for scientific programming.

```
julia> "The " * engine *
   " rocket engine, produced by " *
   company *
   " produces " *
   string(thrust) *
   " kN of thrust"

"The RD-180 rocket engine, produced by Energomash
produces 3830 kN of thrust"
```

When dealing with lots of variables it is usually better to use string interpolation. String interpolation is done with the `$` sign.

```
julia> "The $engine rocket engine, produced by $company produces $thrust kN of thrust"
"The RD-180 rocket engine, produced by Energomash produces 3830 kN of thrust"
```

Observe that you often need to use `$(variable)` instead of `$variable`, when there is no whitespace that can clearly distinguish the variable name from the surrounding text. The same applies if you are trying to interpolate an expression rather than a variable. For instance consider the case where we want to write `3830kN` without the space.

```
julia> "produces $thrustkN of thrust" # Don't work
ERROR: UndefVarError: thrustkN not defined

julia> "produces $(thrust)kN of thrust"
"produces 3830kN of thrust"

julia> "two engines produces $(2 * thrust) kN of thrust"
"two engines produces 7660 kN of thrust"
```

## sprintf formatting

If you are familiar with C programming you may be familiar with the `printf` and `sprintf` functions. Julia has macros called `@printf` and `@sprintf` which mimic these functions. Unlike string interpolation, these macros allow you to specify in more detail how a variable should be displayed.

For instance you can specify the number of digits which should be used when printing a decimal number. `@printf` outputs the result to the console. `@sprintf` and `@printf` are not in the Julia base module which is always loaded. Thus to use these macros we need to include the `Printf` module, which explains the first line.

```
julia> using Printf

julia> @printf("π = %0.1f", pi)
π = 3.1
julia> @printf("π = %0.2f", pi)
π = 3.14
julia> @printf("π = %0.5f", pi)
π = 3.14159
```

Here is a short overview of some common formatting options:

- `%d` integer numbers.
- `%f` floating point numbers.
- `%x` integers shown in hexadecimal notation.
- `%s` show argument as string

With each of these formatting options you can specify things like number of digits, decimals or padding. First let us do some examples of the base formatting options.

```
julia> @sprintf("|%d|", 29)
"|-29|"

julia> @sprintf("|%f|", 29)
"|-29.000000|"

julia> @sprintf("|%x|", 29)
"|-1d|"
```

I put the bars `|` in front of and behind the numbers so that when looking at these next examples of how to do padding, it is easier to see.

```
julia> @sprintf("|%2d|", 42)
"|-42|"

julia> @sprintf("|%4d|", 42)
"|- 42|"

julia> @sprintf("|%-2d|", 42)
"|-42|"
```

```
julia> @sprintf("|%-4d|", 42)
"|-42 |"
```

Notice padding can be applied to either the right side or left side. Right padding is achieved by adding a `-`. Padding is useful if you want to display columns of numbers which you want aligned. You can add padding as zeros instead of space by prefixing the padding number with `o`.

```
julia> @sprintf("|%02d|", 42)
"|-42|"
```

```
julia> @sprintf("|%04d|", 42)
"|-0042|"
```

The padding doesn't say how many spaces or zeros to add, but rather how many characters the numbers should fill in total. If the padding is two and the number has two digits, then nothing will happen. However if the padding is four, we get two spaces added resulting in a total of four characters.

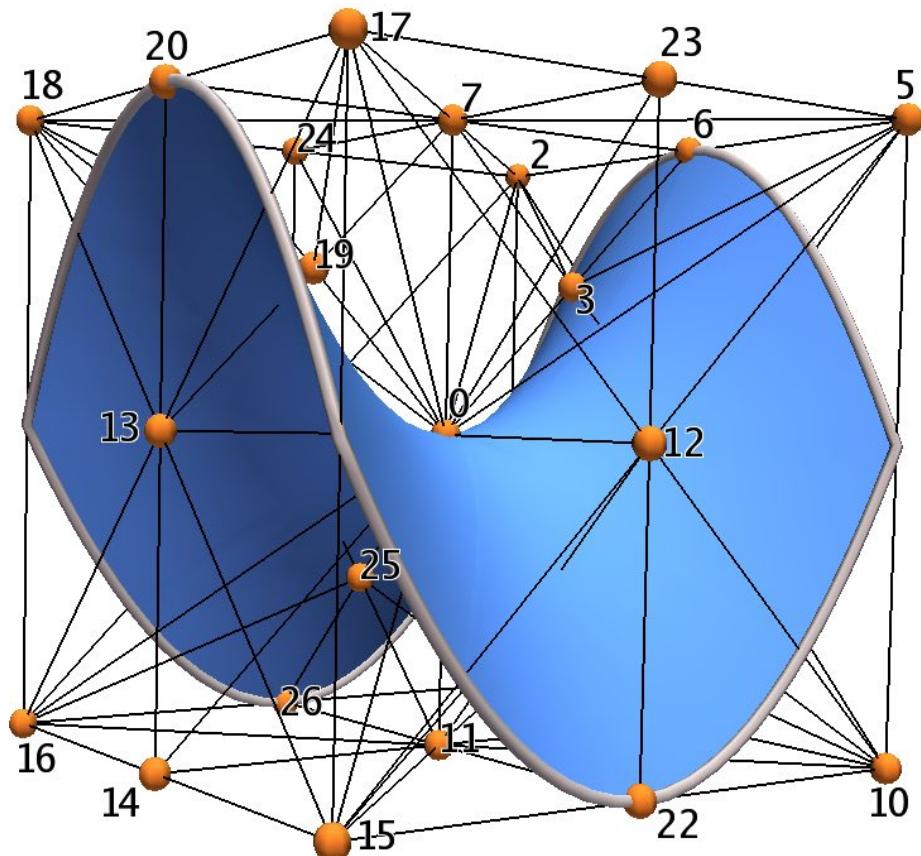


Figure 35: From the Visualization Toolkit VTK

## Using string interpolation to generate code

We can create small utility functions with what we just learned.

In this example we will generate C++ code. Julia may not be your primary work language. Instead you could be using a more verbose language such as C++ or Java at work. But Julia can be used as a companion to make your job easier.

Let me take you through an example of how a C++ developer could simplify their work, by taking advantage of the Julia programming language.

The Visualization Toolkit (VTK) is an amazing C++ library for visualization of scientific data.

Unfortunately writing VTK C++ code is tedious, due to all the typical boilerplate needed in C++.

Below is an example of some of the C++ code used in VTK to define a geometric line. It is not important that you understand what the code below does. I have edit it to remove unnecessary details for our example code.

```
/**
 * @class    vtkLine
 * @brief    cell represents a 1D line
 *
 * vtkLine is a concrete implementation of vtkCell to represent a 1D line.
 */

#ifndef vtkLine_h
#define vtkLine_h

#include "vtkCommonDataModelModule.h" // For export macro
#include "vtkCell.h"
class vtkIncrementalPointLocator;

class VTKCOMMONDATAMODEL_EXPORT vtkLine : public vtkCell
{
public:
    static vtkLine *New();
    vtkTypeMacro(vtkLine,vtkCell);
    void PrintSelf(ostream& os, vtkIndent indent) override;

    int GetCellType() override {return VTK_LINE);}

protected:
    vtkLine();
    ~vtkLine() override {}

private:
    vtkLine(const vtkLine&) = delete;
    void operator=(const vtkLine&) = delete;
};
```

Compare this to the next block of code for defining a polygon. You will notice a lot of repetition. This goes for all VTK code written to define geometric primitives.

```
/***
 * @class    vtkPolygon
 * @brief    a cell that represents an n-sided polygon
 *
 * vtkPolygon is a concrete implementation of vtkCell to represent a 2D
 * n-sided polygon.
 */

#ifndef vtkPolygon_h
#define vtkPolygon_h

#include "vtkCommonDataModelModule.h" // For export macro
#include "vtkCell.h"

class VTKCOMMONDATAMODEL_EXPORT vtkPolygon : public vtkCell
{
public:
    static vtkPolygon *New();
    vtkTypeMacro(vtkPolygon,vtkCell);
    void PrintSelf(ostream& os, vtkIndent indent) override;

    int GetCellType() override {return VTK_POLYGON);}

protected:
    vtkPolygon();
    ~vtkPolygon() override;

private:
    vtkPolygon(const vtkPolygon&) = delete;
    void operator=(const vtkPolygon&) = delete;
};

#endif
```

Imagine that you frequently write new C++ classes (types) like this for different geometric types. It is tedious to repeat all this boilerplate. Fortunately we can make small Julia utility functions to help us.

When generating text consisting of multiple lines it is practical to use triple quotation marks """". It allows you to write strings across multiple lines.

```
function create_class(class::AbstractString)
    s = """
        /**
         * @class    vtk$class
         * @brief
         *
         * vtk$class is a concrete implementation of
```

```

*/
#ifndef vtk$(class)_h
#define vtk$(class)_h

#include "vtkCommonDataModelModule.h" // For export macro
#include "vtkCell.h"

class VTKCOMMONDATAMODEL_EXPORT vtk$class : public vtkCell
{
public:
    static vtk$class *New();
    vtkTypeMacro(vtk$class,vtkCell);
    void PrintSelf(ostream& os, vtkIndent indent) override;

    int GetCellType() override {return VTK_$(uppercase(class));};

protected:
    vtk$class();
    ~vtk$class() override;

private:
    vtk$class(const vtk$class&) = delete;
    void operator=(const vtk$class&) = delete;
};

#endif
"""

clipboard(s) # So we can easily paste class into code editor
println(s)
end

```

Here is an example of using this function to create a Hexagon class. Notice in the last two lines above, that we are also storing the generated code on the clipboard.

```

julia> create_class("Hexagon")
/***
 * @class    vtkHexagon
 * @brief
 *
 * vtkHexagon is a concrete implementation of
 */

#ifndef vtkHexagon_h
#define vtkHexagon_h

#include "vtkCommonDataModelModule.h" // For export macro
#include "vtkCell.h"

class VTKCOMMONDATAMODEL_EXPORT vtkHexagon : public vtkCell

```

```

{
public:
    static vtkHexagon *New();
    vtkTypeMacro(vtkHexagon,vtkCell);
    void PrintSelf(ostream& os, vtkIndent indent) override;

    int GetCellType() override {return VTK_HEXAGON;};

protected:
    vtkHexagon();
    ~vtkHexagon() override;

private:
    vtkHexagon(const vtkHexagon&) = delete;
    void operator=(const vtkHexagon&) = delete;
};

#endif

```

## Non-Standard String Literals

Julia gives you the option of prefixing strings with a word such as `raw"foobar"`, `r"foobar"` and `b"foobar"` to change the meaning of the string ("foobar" in this case). It affects how Julia interprets a string literal.

In fact you can add your own prefixes, but that is a more advance topic.

One interesting fact about non-standard string literals is that they are not created at runtime. By that I mean it is not like we specify a regular string which is then converted to something else when the string literal is encountered at runtime. Instead the value is created at parse time.

## Raw Strings

One issue with regular Julia strings is that characters such as `$` and `\n` have special meaning. For particular kinds of text this can be cumbersome. We can solve it by using the escape character `\``, thus `$` would be written as `\$` and `\n` as `\`n`. However if we don't want to do that, and don't need string interpolation, we can use raw strings.

```

julia> thrust = 3830
3830

julia> raw"produces $thrust kN of thrust" # Don't work
"produces \`${thrust} kN of thrust"

```

## Regular Expressions

Regular expressions is a kind of mini-language which you can use to specify text to match. Regular expressions are widely used in Unix text processing tools and in many coding editors. You can use regular expressions e.g. to search for a particular text string in your code.

In this example we have some Julia source code stored in the variable `s`. We have decided we want to change the name of the `Rocket` type to `SpaceCraft`. We can use the function `replace` to locate some text to replace.

```
julia> s = """
        struct RocketEngine
            thrust::Float64
            Isp::Float64
        end

        mutable struct Rocket
            tank::Tank
            engine::RocketEngine
        end
"""

"struct RocketEngine\n    thrust::Float64\n    Isp::Float64\nend\n\nmutable struct Rocket\n    tank::Tank\n    engine::RocketEngine\nend"

julia> result = replace(s, "Rocket"=>"SpaceCraft");

julia> println(result)
struct SpaceCraftEngine
    thrust::Float64
    Isp::Float64
end

mutable struct SpaceCraft
    tank::Tank
    engine::SpaceCraftEngine
end
```

As you remember from Storing Data in Dictionaries, we use the `=>` operator to create a pair. This was used to create key-value pairs to store in the dictionary. In this case the pair represents text to find and substitution text. So "`Rocket"=>"SpaceCraft"`" means locate "`Rocket`" and replace it with "`SpaceCraft`".

However as we see from the example, this does not do exactly as you would have expected. "`RocketEngine`" also gets replaced with "`SpaceCraftEngine`". However we only want the `Rocket` type to be changed. With regular expressions is easier to be more specific about what we are looking for.

The intention here is not to teach you regular expressions, but just to give a flavor so you can see the benefit of using this in Julia. In regular expressions ". "

means any character. [A–D] means any character from `A' to `D'. While if you write [^A–D] it means any character **not** in the range `A' to `D'. So "Rocket[^A–Za–z]" would mean finding the word ``Rocket'', and where the first succeeding character is **not** a letter.

```
julia> result = replace(s, r"Rocket[^A-Za-z]"=>"SpaceCraft");

julia> println(result)
struct RocketEngine
    thrust::Float64
    Isp::Float64
end

mutable struct SpaceCraft      tank::Tank
    engine::RocketEngine
end
```

In this example we turn the string we are searching for into a regular expression by prefixing it with a `r`. That means it will not be a string object. We can demonstrate this in the REPL.

```
julia> regex = r"Rocket[^A-Za-z]"
r"Rocket[^A-Za-z]"
```

```
julia> typeof(regex)
Regex
```

This regular expression object is created during parsing, not at runtime. That means there is no need to store this regular expression in a variable once and reuse it multiple times. There is no problem writing a regular expression literal inside a loop executed a million times. The regular expression is only compiled once.

By compiling of a regular expression we mean parsing the regular expression string and constructing a regular expression object.

## BigInt

A literal syntax exists for most number types, as shown with these examples:

```
julia> typeof(42)
Int64

julia> typeof(0x42)
UInt8

julia> typeof(0x42000000)
UInt32

julia> typeof(0.42)
Float64
```

```
julia> typeof(0.42f0)
Float32
```

```
julia> typeof(3//4)
Rational{Int64}
```

In the cases where it does not exist, you can do a conversion like this `Int8(42)`, which takes a 64 bit signed integer and turns it into a 8 bit signed integer. When writing integers of arbitrary precision (any number of digits) you can do this as well by writing `BigInt(42)`, however this causes potentially some inefficiency. Everywhere this is encountered an integer has to be converted to a big int. Instead if you write `big"42"` the big integer is created when the program is parsed and not each time it is run.

## MIME types

Various operating systems have different systems for keeping track of the type of its files. Windows e.g. famously use a 3-letter filename extension to indicate the type of a file. The original macOS stored the file type in special attributes.

However to send files of different types between computers on the internet one needs a common standard, to identify what the filetypes are. This is what MIME types are. They are typically described as a type and subtype separated by a slash. HTML pages are denoted as `text/html` while JPEG images are denoted as `image/jpeg`. A PNG file type would be written as `image/png` and so on.

You can create a MIME type object in Julia with:

```
julia> MIME("text/html")
MIME type text/html
```

```
julia> typeof(ans)
MIME{Symbol("text/html")}
```

So a MIME type object `MIME("foo/bar")` would have the type `MIME{Symbol{"foo/bar"}}`. This will look somewhat cryptic until we cover Parametric Types in the last chapter. `MIME{Symbol{"foo/bar"}}` is long and cumbersome to write, which is why Julia offers the shortcut `MIME"foo/bar"`.

This is easy to mixup. `MIME("foo/bar")` and `MIME"foo/bar"` are **not** the same thing, the first case is an object. The latter is the type of this object. Here is a simple example of how you could use this to create methods giving different outputs for different MIME types.

```
say_hello(::MIME"text/plain") = "hello world"
say_hello(::MIME"text/html") = "<h1>hello world</h1>"
```

This is useful because it allows us to define functions in Julia which can give different formatted textual outputs for different contexts.

```
julia> say_hello(MIME("text/plain"))
"hello world"
```

```
julia> say_hello(MIME("text/html"))
"<h1>hello world</h1>"
```

Julia code executing in a graphical notebook style environment such as Jupyter would get passed an HTML MIME type, so graphs and tables can be rendered as HTML.



# Object Collections

- **Collection Categories.** Collections can be organized into different categories depending on the type of operations they support such as iteration, random access, stack operations etc.
- **Iteration.** How to make collections iterable, so they can be used in for loops.
- **Working with Multi-dimensional Arrays.** Operations for handling arrays with more than one dimension, such as dot product, cross product, slicing and concatenation.
- **Sets.** Using operations such as intersection and union on sets.

We have already looked at collections such as arrays and dictionaries, but what about other collection types?

And what exactly makes something a collection? What are the differences and similarities between different collections and how would you make your own collection type?

We are going to explore these questions by expanding on our multi-stage rocket example. Because the rocket is made up of many different parts, it is possible to turn it into something Julia will recognize as a collection.

Finally we will create from scratch a collection type implementing the dictionary interface.

## Interfaces

If your programming background is with statically typed object-oriented languages such as Java or C#, you may already have a very particular idea of what an interface is. It is an abstract type with a set of associated methods (in the object-oriented sense), which you have to implement for your type to adhere to that interface.

To some degree you have similar semantics in Julia. There are abstract types such as `AbstractArray` which your custom collection type can subtype.

However `AbstractArray` as defined in code does not specify any methods you need to implement. Rather it defines an informal interface.

There is a set of functions informally associated with `AbstractArray`, which you are expected to attach methods to, operating on your concrete subtype. Imple-

menting these methods will give you a lot of `AbstractArray` functionality for free.

How do you know which functions you need to attach methods to? Usually this is handled by the documentation of an abstract type. If it is not fully documented, you can determine it by testing your collection in different settings and check if Julia complains about any missing methods.

Let me give an example of what I mean. Say we have an abstract collection called `AbstractBag` and `GrabBag` was a concrete implementation of this collection. Further, let us pretend `methodswith(AbstractBag)` lists `pickitem` as a method defined for `AbstractBag`. Next you want to know if your `GrabBag` type can handle this function. So we try it out in the REPL.

```
julia> bag = GrabBag()
julia> pickitem(bag)
ERROR: MethodError: no method matching shake(::GrabBag)
```

This `MethodError` error message is what tells you that `shake` is a function defined for `AbstractBag` which you need to add a method to for handling `GrabBag` types.

Keep in mind, this is your last way out. For dynamic languages like Julia it is vital that people properly document their APIs.

However not all interfaces in Julia are connected to a specific abstract type like this. For instance there is a iteration interface. If you implement this interface, you will be able to iterate over your collection using a for-loop. It will also make it possible to use it with functions such as `map`, `reduce` and `filter` which operate on iterable collections.

The iteration interface is not represented by any particular abstract type you need to implement. Rather it is informally described. You are at a minimum expected to extend the `iterate` function for your collection type.

In fact here is a full list of required and optional functions to implement the iterable interface, taken from the official Julia documentation.

## Required methods

- `iterate(iter)` Returns either a tuple of the first item and initial state or `nothing` if empty
- `iterate(iter, state)` Returns either a tuple of the next item and next state or `nothing` if no items remain

## Optional methods

These methods tend to have default implementations which work fine in a lot of cases.

- `IteratorSize(IterType)` One of `HasLength()`, `HasShape{N}()`, `IsInfinite()`, or `SizeUnknown()` as appropriate.

- `IteratorEltype(IterType)` Either `EltypeUnknown()` or `HasEltype()` as appropriate.
- `eltype(IterType)` The type of the first entry of the tuple returned by `iterate()`.
- `length(iter)` The number of items, if known.
- `size(iter, [dim])` The number of items in each dimension, if known.

We are going to implement methods for some of these functions, `iterate`, `eltype` and `IteratorSize`.

A collection is a type of object containing other objects referred to as elements. In our case the elements are of type `Payload`, which means they can either be a `Rocket`, `SpaceProbe`, `Capsule` or `Satellite`. But what structure should represent the collection itself?

To answer that we will take a little detour and give some more details on space rocket terminology.

## Rocket Terminology

Initially when building our example code, we used very loose terminology. E.g. when you say rocket, that can mean any number of things. Thus instead let us define the terminology more accurately:

- **Space Vehicle** is a more accurate word for what I have loosely called a *space rocket* previously. It is the whole thing sitting on the launchpad, with multiple rocket stages and a payload on top.
- **Launch Vehicle** is what you get when you remove the payload from the *space vehicle*.
- **Space Craft** is a vehicle that flies around in space. Typically in maneuvers between orbits. It is not made for landing and taking off from planetary or lunar surfaces. A number of vehicles fit the description of *space craft*. Autonomous probes exploring the solar system are a type of space craft. The Apollo command and service module was also a spacecraft. The Russian Soyuz is a space craft used for transporting crew to and from the international space station. It gets put into orbit by a launch vehicle dubbed the ``Soyuz rocket''.
- **Booster** is the first stage rocket. Its job is to get the rest of the rocket up to speed and at high enough altitude that the rocket can start turning sideways to get into orbit.
- **2nd and 3rd Stage** are both rockets. In principle there can be any number of stages.

## Space Vehicle Collection

Thus we are ready to make a new collection type called a `SpaceVehicle`. The elements of the collection are `Rocket` stages. Except the last element which will be something other than a `Rocket`, otherwise we are not terminating the collection. We could achieve this by defining the next stage as a union type

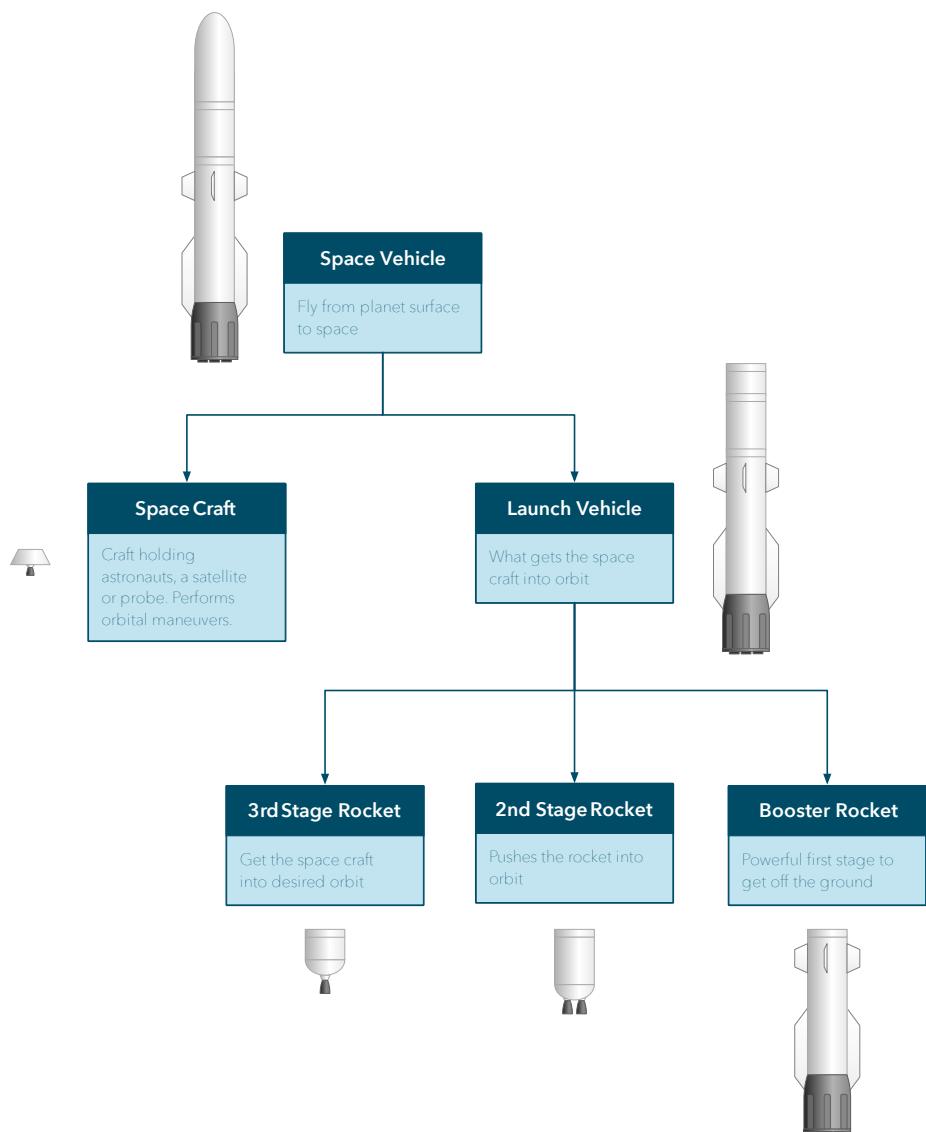


Figure 36: Rocket hierarchy

`Union{Rocket, Nothing}`, however we prefer the flexibility of having different types of payloads at the end.

To achieve this we will define a new type of payload to signal that there is *no* payload, and give the option of creating Rocket stages with no payload as default.

```
struct NoPayload <: Payload
end

mass(payload::NoPayload) = 0.0

Rocket(tank::Tank, engine::Engine) = Rocket(NoPayload(), tank, engine)
```

This allows us to create a space vehicle collection which is empty.

```
mutable struct SpaceVehicle
    active_stage::Payload
end

SpaceVehicle() = SpaceVehicle(NoPayload())

function stage_separate!(ship::SpaceVehicle)
    stage = ship.active_stage
    if stage isa Rocket
        ship.active_stage = stage.payload
        stage.payload = NoPayload()
        stage
    else
        nothing
    end
end
```

The `stage_separate!` function allows us to detach the bottom stages of the space vehicle, as the fuel in the bottom active stage is consumed. The function returns the stage being separated or `nothing` if there are no more stages to separate.

We can construct a space vehicle following the procedure below:

```
julia> merlin = Engine(845e3, 282, 470)
SingleEngine(845000.0, 282.0, 470.0)

julia> tank = Tank(4e3, 111.5e3)
Tank(4000.0, 111500.0, 107500.0)

julia> r2 = Rocket(SpaceProbe(22e3), tank, merlin)
Rocket(
    SpaceProbe(22000.0),
    Tank(4000.0, 111500.0, 107500.0),
    SingleEngine(845000.0, 282.0, 470.0))

julia> r1 = Rocket(r2, tank, EngineCluster(merlin, 9))
```

```
Rocket(
    Rocket(
        SpaceProbe(22000.0),
        Tank(4000.0, 111500.0, 107500.0),
        SingleEngine(845000.0, 282.0, 470.0)),
        Tank(4000.0, 111500.0, 107500.0),
        EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9))

julia> ship = SpaceVehicle(r1)
SpaceVehicle(
    Rocket(
        Rocket(
            SpaceProbe(22000.0),
            Tank(4000.0, 111500.0, 107500.0),
            SingleEngine(845000.0, 282.0, 470.0)),
            Tank(4000.0, 111500.0, 107500.0),
            EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9)))
```

However this is cumbersome and the code does not make it as clear as it ought to be, what the sequence of the different stages are. We want a constructor for the ``collection'' which has a similar looking constructor to that of other collection types.

But first let us create some helper functions to simplify the task of creating such a constructor.

## Adding and Removing Elements

Many Julia collections support operations such as `push!`, `pop!`, `pushfirst!` and `popfirst!`. The difference between these operations is whether elements are being added or removed in the front or back. The specifics are best explained with a example:

```
julia> xs = Int[7]
1-element Array{Int64,1}:
 7
```

Add two numbers to the back.

```
julia> push!(xs, 9, 11)
3-element Array{Int64,1}:
 7
 9
11
```

Add two number to the front.

```
julia> pushfirst!(xs, 3, 5)
5-element Array{Int64,1}:
 3
 5
 7
 9
```

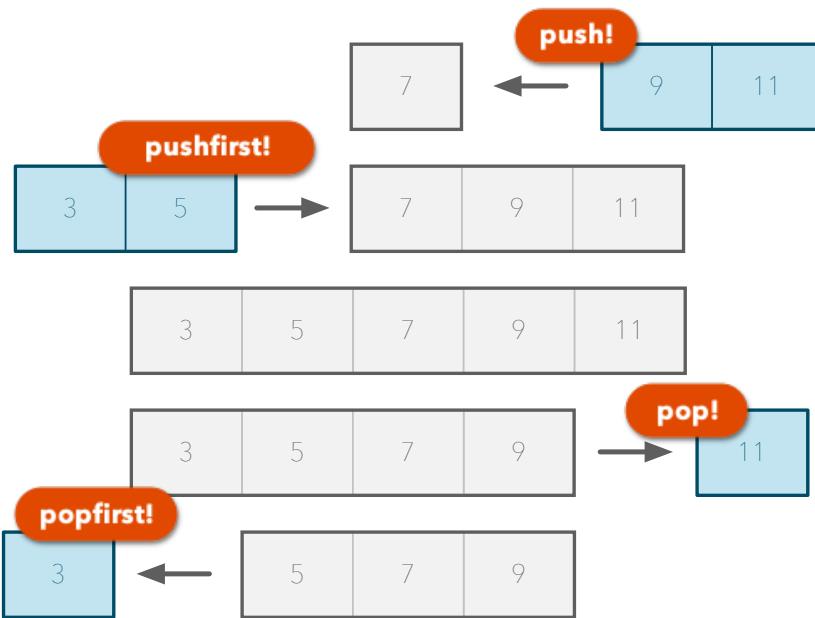
11

Remove number from the back.

```
julia> pop!(xs)
11
```

Remove number from the front.

```
julia> popfirst!(xs)
3
```



In our case it would make most sense to implement the operations working at the front since those don't require a linear time traversal to the end. Instead adding or removing an element can be done in constant time.

#### NOTE

**linear time** and **constant time** are concepts from computer science used to describe how the running time of an algorithm is affected by increasing the number of elements the algorithm has to process. If an operation is linear then it is assumed that the time it takes to process twice the number of elements is doubled. If the time is said to be constant, it does not change regardless of how many or few elements you process.

`popfirst!` should be trivial to implement since it is basically what `stage` separate already does.

```

import Base: popfirst!, pushfirst!

popfirst!(ship::SpaceVehicle) = stage_separate!(ship)

function pushfirst!(ship::SpaceVehicle, rockets...)
    for r in reverse(rockets)
        r.payload = ship.active_stage
        ship.active_stage = r
    end
    ship
end

function SpaceVehicle(rockets::Array{Rocket})
    ship = SpaceVehicle()
    pushfirst!(ship, rockets...)
    ship
end

```

This allows us to construct a space vehicle with cleaner code:

```

julia> r1 = Rocket(tank, EngineCluster(merlin, 9))
Rocket(NoPayload(),
       Tank(4000.0, 111500.0, 107500.0),
       EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9))

julia> r2 = Rocket(SpaceProbe(22e3), tank, merlin)
Rocket(SpaceProbe(22000.0),
       Tank(4000.0, 111500.0, 107500.0),
       SingleEngine(845000.0, 282.0, 470.0))

julia> ship = SpaceVehicle([r1, r2])
SpaceVehicle(
    Rocket(Rocket(NoPayload(),
                  Tank(4000.0, 111500.0, 107500.0),
                  SingleEngine(845000.0, 282.0, 470.0)),
            Tank(4000.0, 111500.0, 107500.0),
            EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9)))

```

Or we could make an empty space vehicle and use pushfirst!:

```

julia> ship = SpaceVehicle()
SpaceVehicle(NoPayload())

julia> pushfirst!(ship, r1, r2)
SpaceVehicle(
    Rocket(Rocket(
        NoPayload(),
        Tank(4000.0, 111500.0, 107500.0),
        SingleEngine(845000.0, 282.0, 470.0)),
    Tank(4000.0, 111500.0, 107500.0),
    EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9)))

```

## Supporting Iteration

If we try to iterate over the rockets in the space vehicle, you can see that is currently not possible, because the `iterate` methods have not been implemented yet.

```
julia> for r in ship
           println(mass(r))
       end
ERROR: MethodError: no method matching iterate(::SpaceVehicle)
```

A for-loop in Julia involving collections which can be iterated on is basically syntactic sugar for a more low level while-loop. Consider this for-loop over a collection of values:

```
xs = [3, 5, 7]
```

```
for x in xs
    println(x)
end
```

This would get simplified in Julia to the code below:

```
xs = [3, 5, 7]

next = iterate(xs)
while next != nothing
    (x, i) = next
    println(x)
    next = iterate(xs, i)
end
```

Notice that `iterate` always gives a tuple or `nothing` as return value. The tuple contains an element in the collection, `x` and some state `i`, which is used to keep track of *where* the next element in the collection is located. In this example `i` is simply an index to the next element.

But in our case it is more practical for `i` to be the next rocket object, because rockets ``know'' the next rocket, through the `payload` property.

To get this working, it is practical to define iteration on rocket objects, because we need to be able to distinguish between cases where the payload is a rocket and when it isn't.

We start by specifying what happens in the general case. We give the second element of the tuple the value `nothing` as a way of marking that the iteration should stop. This is not a standard, but rather something that makes sense in this case.

```
import Base: iterate

iterate(payload::Payload) = payload, nothing
```

Now we can specify that a specific case, like when we have an actual rocket object, we know it is possible to continue the iteration, because it has a payload.

```
iterate(r::Rocket) = r, r.payload
```

In the two-argument case, we got a collection we are iterating over and some state that indicate where we are.

```
iterate(:: Rocket, state) = state == nothing || state isa NoPayload ? nothing : iterate(s
```

If the state is nothing, then it is a signal to stop iteration, so we should also return nothing. The for-loop is designed to stop once iterate return nothing. Otherwise what you do is not different from when you get a single argument to iterate.

We can try out every step of the iteration explicitly to make sure that it works.

```
julia> r0 = ship.active_stage
Rocket(Rocket(NoPayload(),
    Tank(4000.0, 111500.0, 107500.0),
    SingleEngine(845000.0, 282.0, 470.0)),
    Tank(4000.0, 111500.0, 107500.0),
    EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9))

julia> r, rn = iterate(r0)
(Rocket(Rocket(NoPayload(),
    Tank(4000.0, 111500.0, 107500.0),
    SingleEngine(845000.0, 282.0, 470.0)),
    Tank(4000.0, 111500.0, 107500.0),
    EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9)),
Rocket(NoPayload(),
    Tank(4000.0, 111500.0, 107500.0),
    SingleEngine(845000.0, 282.0, 470.0)))

julia> r == r1
true

julia> rn == r2
true

julia> r, rn = iterate(r0, rn)
(Rocket(NoPayload(),
    Tank(4000.0, 111500.0, 107500.0),
    SingleEngine(845000.0, 282.0, 470.0)),
NoPayload())

julia> r == r2
true

julia> next = iterate(r0, rn)
```

```
julia> next == nothing
true
```

The final task is to modify this code to be able to handle iteration over a SpaceVehicle object.

```
iterate(ship::SpaceVehicle) = iterate(ship.active_stage)
```

This handles the start of the iteration. But as you might see, it trips up on the following iteration steps when `iterate` expects two arguments.

```
julia> for r in ship
           println(mass(r))
       end
ERROR: MethodError: no method matching iterate(::SpaceVehicle, ::Rocket)
```

The reason for this is that our two-argument `iterate` method expects a `Rocket`. We can solve this in multiple ways. One way would be to duplicate the existing `iterate` method, to create a version taking a `SpaceVehicle` instead. However in Julia you can often avoid code duplication by using a type union, to let one method handle two different types.

```
function iterate(::Union{Rocket, SpaceVehicle}, state)
    if state == nothing || state isa NoPayload
        nothing
    else
        iterate(state)
    end
end
```

Now the `iterate` function works:

```
julia> for r in ship
           println(mass(r))
       end
227700.0
111970.0
```

## Benefits of Implementing IteratorSize

However, important functionality is still missing. We cannot use `map`, `filter` and `collect`, which are functions frequently used on collections. `collect(ship)` returns an array of rockets contained within the ship. That is basically what `collect` is for: Take something you can iterate over and turn it into an `Array`.

```
julia> map(mass, ship)
ERROR: MethodError: no method matching length(::SpaceVehicle)

julia> collect(ship)
ERROR: MethodError: no method matching length(::SpaceVehicle)
```

You can see Julia complains about there being no implementation of the `length` function for `SpaceVehicle` types. Julia expects to find this function to be able to pre-allocate an array of a particular length and then fill it up with the result.

It does this using an abstract type called `IteratorSize`. How? When you call e.g. `collect(ship)` this translates into a private function call:

```
_collect(ship, IteratorEltype(ship), IteratorSize(ship)).
```

Looking at this line, you may ask ``Didn't you just tell me `IteratorSize` is an abstract type?'' Indeed I did. And as I talked about before, you cannot create instances of abstract types. Thus the 3rd argument `IteratorSize(ship)` looks illegal.

But there is a simple explanation. Julia lets you create functions with any name, including the name of an abstract type.

The `IteratorSize` function is meant as a facade to create one of the concrete subtypes shown below.

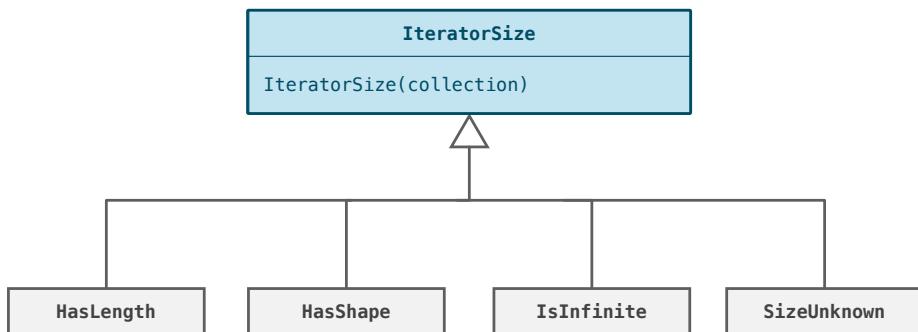


Figure 37: `IteratorSize`

The type of the instance created depends on the type of the collection provided.

If you are familiar with object-oriented design patterns such as the factory method, you may recognize some similarity.

If you remember the way a factory method is implemented, you will be pleasantly surprised by the sheer elegance of the Julia solution to the same problem. All thanks to the marvels of multiple-dispatch.

```

import Base: IteratorSize

IteratorSize(::SpaceVehicle) = Base.SizeUnknown()
IteratorSize(::Payload) = Base.SizeUnknown()
  
```

Adding support for an extra collection type, simply requires adding a single function. Unlike the factory method pattern, there is no central function which needs to be manually edited to add support for another collection type.

With this change we are allowed to use functions such as `map`, `filter` and `collect`

```

julia> map(mass, ship)
2-element Array{Float64,1}:
 227700.0
 111970.0

julia> collect(ship)
2-element Array{Any,1}:
 Rocket()
  
```

```
Rocket(NoPayload(),
       Tank(4000.0, 111500.0, 107500.0),
       SingleEngine(845000.0, 282.0, 470.0)),
Tank(4000.0, 111500.0, 107500.0),
EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9))

Rocket(NoPayload(),
       Tank(4000.0, 111500.0, 107500.0),
       SingleEngine(845000.0, 282.0, 470.0))
```

There is just one tiny missing piece. When using `collect` Julia says the output is a 2-element `Array{Any,1}`. But this is not quite what we want. We want an array of `Payload` items, not of `Any` items. With the `eltype` function we can tell Julia what type each element in a collection is.

Below is an example of how we implement this:

```
eltype(::Type{SpaceVehicle}) = Payload
eltype(::Type{Rocket}) = Payload
```

It shows that whether we are treating rockets or space vehicles as the collection type, the elements are of type `Payload`.

```
julia> collect(ship)
2-element Array{Payload,1}:
 Rocket(
   Rocket(NoPayload(),
         Tank(4000.0, 111500.0, 107500.0),
         SingleEngine(845000.0, 282.0, 470.0)),
   Tank(4000.0, 111500.0, 107500.0),
   EngineCluster(SingleEngine(845000.0, 282.0, 470.0), 9))

Rocket(NoPayload(),
       Tank(4000.0, 111500.0, 107500.0),
       SingleEngine(845000.0, 282.0, 470.0))
```

## Implementing an XML Attribute List

So we have looked at creating a collection interface to an existing data structure, so that it can be used as a collection. In this example we will build a data structure for holding attributes of XML tags.

This is the kind of problem that can arise if you write code for parsing or generating an XML document. Consider the XML fragment below:

```
<item row="1" column="2">
    <widget class="QSpinBox" name="red_spinner"></widget>
</item>
```

We want a structure to be able to hold attributes such as `class="QSpinBox"` `name="red_spinner"`. A simple approach would be to use a dictionary.

```
julia> attrs = Dict("class"=>"QSpinBox", "name"=>"red_spinner")
```

```
Dict{String, String} with 2 entries:
  "name"  => "red_spinner"
  "class" => "QSpinBox"

attrs["class"]
```

However as you can see, this approach does not preserve the order of attributes. If you are reading an XML file, modify it and then write it out again, you don't want to change the order of attributes. However we like to keep the dictionary-like interface.

Thus we are going to build a collection with a dictionary interface which preserves the order in which items were added to it.

Let us call this `OrdDict`, which could be short for ordered or ordinal dictionary. We will make it a dictionary where the keys are symbols and the values are strings. Why symbols? The benefits of this will become apparent later. But one reason, is that XML attributes typically don't have white space. Strings without whitespace are easy to express as symbols.

First we import all the collection type specific functions we want to add methods to. We will cover in turn what their purpose are.

```
import Base: getindex, setindex!, get, delete!,
            length, iterate,
            getproperty, setproperty!,
            haskey, isempty,
            union, push!,
            copy
```

## Type Definition and Constructors

Next we define our collection structure. We add construction functions to make it easier to make an empty collection or one from multiple key-value pairs passed as arguments or within an array.

```
mutable struct OrdDict
    items::Vector{Pair{Symbol, String}}
end

OrdDict() = OrdDict(Pair{Symbol, String}[])
OrdDict(items...) = OrdDict(collect(items))
```

There are a bunch of things here which we never discussed in detail before. For instance we are saying that a `OrdDict` contains items of type `Vector{Pair{Symbol, String}}`. This is what we call a parameterized type. `Vector` and `Pair` are parameterized types and so we got to give them a type as a parameter to create a concrete `Vector` or `Pair` type. Just like a function takes values as parameters, types can take other types as parameters, but only when they are defined as parameterized types.

**NOTE**

We are not going to cover how to make your own parameterized types here, but it is useful to know of their existence and how you use them, because parameterized types are ubiquitous when dealing with collections.

Union is also a parametrizable type.

The type of a type is also a type parameterization. To get the type of the Int type object we, parameterize Type with Int like this Type{Int}.

`Vector{Pair{Symbol, String}}` denotes a vector type. This vector is a 1D array, containing pairs, where each of the pairs are made up of a `Symbol` and a `String` object.

But why do we store dictionary data in an array? Would that not be very slow to lookup? The lookup time in fact linear with this solution. That means it grows with the number of elements. Julia's builtin dictionary has constant time lookup, meaning that the time it takes to find an element does not change with the number of elements contained in the collection.

However searching an array is very fast for *small* arrays. An XML tag is unlikely to contain more than a dozen elements. This is such a small number of elements that an array will be faster than almost any other solution. Key comparison is fast because when comparing symbols one does not look at every character (which you need to do with strings).

Let us try out our current functionality:

```
julia> item = OrdDict(:row=> "1", :column=> "2")
OrdDict([:row => "1", :column => "2"])

julia> widget = OrdDict([:class=> "QSpinBox", :name => "red_spinner"])
OrdDict([:class => "QSpinBox", :name => "red_spinner"])

julia> for (k,v) in item
           println("key: ", k, " value: ", v)
       end
ERROR: MethodError: no method matching iterate(::OrdDict)
```

## Iteration

Naturally iteration does not work, because we have not implemented the iterator interface. But we already have a good idea of how to do that. And fortunately this time we can simply forward to existing built in iteration functionality. We already know that `Vector` can be iterated.

```
length(a::OrdDict)      = length(a.items)
isempty(a::OrdDict)     = isempty(a.items)
iterate(a::OrdDict)     = iterate(a.items)
```

```
iterate(a::OrdDict, i) = iterate(a.items, i)
```

Voilà iteration works as should be expected:

```
julia> for (k,v) in item
           println("key: ", k, " value: ", v)
       end
key: row value: 1
key: column value: 2

julia> collect(item)
2-element Array{Any,1}:
 :row => "1"
 :column => "2"

julia> map(first, item)
2-element Array{Symbol,1}:
 :row
 :column

julia> map(last, item)
2-element Array{String,1}:
 "1"
 "2"
```

## Lookup Values by Key and Exceptions

We can store and lookup values by key in a dictionary. Let me show you how this works in a regular dictionary, because we want to mimic the behavior of a regular dictionary.

```
julia> d = Dict("two" => 2, "four" => 4)
Dict{String,Int64} with 2 entries:
  "two"  => 2
  "four" => 4

julia> d["two"]
2

julia> d["five"]
ERROR: KeyError: key "five" not found
```

Notice that Julia produces an error, when you try to get a key that does not exist in the dictionary. We need to reproduce this behavior. We could use the `error()` function to report an error. However to make it easier to deal with specific types of errors, Julia uses what we call *exceptions*.

Exceptions will not be covered in detail yet. What you need to know is that exceptions affect control flow. Depending on the language, using an exception is referred to as *raising* or *throwing* an exception. In Julia we say to ``throw an exception.''

It is easier to see how exceptions work with some examples:

```
julia> for num in ["one", "two", "three"]
       println(num)
       error("something went wrong")
   end
one
ERROR: something went wrong

julia> for num in ["one", "two", "three"]
       println(num)
       throw(ErrorException(("something went wrong")))
   end
one
ERROR: something went wrong
```

What we are demonstrating here is that calling `error` is just the same as throwing an `ErrorException`. It also shows how the for-loop is exited by the `throw` statement. Thus anywhere you determine something has gone wrong you can throw an exception to abort.

How did I know that `error` just throws an `ErrorException`? Because I used the `@less` macro:

```
julia> @less error("something went wrong")
```

Alternatively, you could use the `@edit` macro. Either way it gives you a chance to look at how a function is implemented. We can use the same approach to discover how key-lookup is done with a dictionary:

```
julia> d = Dict("two" => 2, "four" => 4)
Dict{String,Int64} with 2 entries:
  "two"  => 2
  "four" => 4
```

```
julia> @less d["five"]
```

This will cause the following code to be displayed:

```
function getindex(h::Dict{K,V}, key) where V where K
    index = ht_keyindex(h, key)
    @inbounds return (index < 0) ? throw(KeyError(key)) : h.vals[index]::V
end
```

Notice the types are parameterized, but don't focus on that detail now. The key takeaway here is:

1. Implementing key-based access requires implementing a `getindex` method.
2. We deal with missing keys by throwing a `KeyError` exception.

We implement our version by iterating over all the key-value pairs and looking for the pair where we get a match on the keys. If we don't find such a pair, we throw a `KeyError` exception.

```
function getindex(a::OrdDict, key::Symbol)
    for item in a.items
```

```

        if first(item) == key
            return last(item)
        end
    end
    throw(KeyError(key))
end

```

Let me explain in more detail how this works. An expression such as `dict[key]` will get translated by Julia into `getindex(dict, key)`. But why is the word ``index'' used in the function name? Would it not make more sense to call it `get_value_by_key` or something similar?

The beauty of the Julia approach is that it abstracts away the difference between accessing an element by *key* and by *index*. We use exactly the same function to implement index based access in an array.

Thus if we wanted to, we could add integer index access as well, like this:

```
getindex(a::OrdDict, index::Integer) = a.items[index]
```

This allows us to pick a key-value pair at a particular index. It also demonstrates the power of Julia multiple-dispatch. We have great flexibility in what we use for keys. We can even alter the behavior completely depending on the type of the key.

You could for instance register a method to handle cases where the index is a collection of integers, to return a collection of values. In fact regular Julia collections allow you to do this. That is how ranges (slices) are implemented.

But to get a minimal functioning dictionary interface, we also need a way of adding elements by key. To do this you use the `setindex!` function.

#### NOTE

You can determine which method should be implemented by either looking up collection interfaces in the official Julia documentation, or by using the `@less` or `@edit` macro to discover how builtin collections do it.

However you should always verify with documentation, since you should not base your design decision on how the standard Julia library is implemented. Implementation details can change.

```

function setindex!(a::OrdDict, value::AbstractString, key::Symbol)
    for (i, item) in enumerate(a.items)
        if first(item) == key
            a.items[i] = key => value
            return
        end
    end
    push!(a.items, key => value)
end

```

Because this function mutates our collection, we suffix it with an exclamation mark (!). This is a convention and has no syntactic meaning. It is useful for anybody writing code to see where objects may get modified.

The function is pretty straightforward. Use `enumerate` as covered in the Storing Data in Dictionaries chapter, to get both the value and index of each item in our collection. In this case each value is a key-value pair.

If we don't find the specified key among any of the pairs, we instead add a new pair to the end of the collection with `push!()`. This means items will get the same order as they were added.

In fact it is useful to provide a more array like interface to the collection to avoid having to do a complete search each time an item is added. So we can implement a `push!()` method for the `OrdDict` collection:

```
push!(a::OrdDict, x::Pair{Symbol, String}) = push!(a.items, x)
```

## Checking for Items and Removing Them

There are also other functions useful to give all the functionality expected from a dictionary. We want to be able to check if an item is in the dictionary or not without risking throwing an exception. And we want to be able to remove items.

```
function haskey(a::OrdDict, key::Symbol)
    for item in a.items
        if first(item) == key
            return true
        end
    end
    false
end

function delete!(a::OrdDict, key::Symbol)
    for (i, item) in enumerate(a.items)
        if first(item) == key
            deleteat!(a.items, i)
        end
    end
    a
end
```

Notice at the end of `delete!` we are returning the collection. If you come from a background of typical imperative object-oriented languages this may look strange to you. However this is a common habit in functional languages or languages geared towards REPL based development. To make it easier on the programmer working in a REPL environment, you want to make it quick to see how the state of an object was modified in response to a function call.

When writing Julia code you will get into the habit of deciding what to return based on what aids you when doing interactive coding in the REPL. Incidentally this often encourages better API design.

## Pretending to be a Struct

The next functionality we are going to look at is not *needed* for a collection. However sometimes it is useful to make a collection look as if it is just a regular struct with multiple items. This gives Julia the ability to pretend to be more like traditional script languages such as Python, Ruby and JavaScript, where items can be added and removed to an object at runtime.

A Julia struct is more like a C/C++ struct. It does not allow modification after creation. However that does not mean we cannot fake it.

Whenever you use the dot, to access a field such as `rocket.tank`, you are not actually directly accessing the field. In fact when Julia parses this statement it will replace it with:

```
getproperty(rocket, :tank)
```

Now it may dawn on you why I insisted on using symbols as keys rather than plain strings. `getproperty` is defined in the Julia standard library as:

```
getproperty(x, f::Symbol) = getfield(x, f)
```

So in fact if you want to perform raw access to a member, you don't write `rocket.tank`, but you write `getfield(rocket, :tank)`. However this allows us to override the behavior of `getproperty` for our collection type, by adding our own `getproperty` method.

```
function getproperty(a::OrdDict, key::Symbol)
    if key == :items
        getfield(a, :items)
    else
        a[key]
    end
end

function setproperty!(a::OrdDict, key::Symbol, value::AbstractString)
    if key == :items
        setfield!(a, :item, value)
    else
        a[key] = value
    end
end
```

We have implemented these methods for getting and setting data fields, such that we always check first if the field we are accessing is named `items`, because we want normal access for the `items` field. Otherwise we do a dictionary style lookup.

Here are some examples of using these methods, to easily get and set values:

```
julia> d = OrdDict(:one => "1", :two => "2")
OrdDict([:one => "1", :two => "2"])
```

```
julia> d[:one]
"1"
```

```
julia> d.one
"1"

julia> d[:two] = "five"
"five"

julia> d.two = "eight"
"eight"

julia> d
OrdDict([:one => "1", :two => "eight"])

julia> # Adding items

      d.foo = "bar"
"bar"

julia> d.alpha = "beta"
"beta"

julia> d
OrdDict([:one => "1", :two => "eight", :foo => "bar", :alpha => "beta"])
```

## Make it a Dictionary

We have defined all the functionality we need to make our collection act as a dictionary, but does that mean it *is* a dictionary?

From Julia's point of view it will not always be that. A function expecting some kind of dictionary will indicate that the type must be an `AbstractDict`. This is similar to how string arguments are usually specified as `AbstractString` rather than just `String` to keep open the possibility for other string representations to be used. Our dictionary is not a subtype of `AbstractDict` and thus Julia will not assume it is a dictionary either.

However this is simple to change:

```
mutable struct OrdDict <: AbstractDict{Symbol, String}
    items::Vector{Pair{Symbol, String}}
end
```

### NOTE

`AbstractDict{Symbol, String}` is a parameterized type representing an abstract dictionary with symbol keys and string values. By making `OrdDict` a subtype of this, we get a lot of dictionary related functionality for free.

If you are in a REPL environment and you write this change, you will be mostly

out of luck. Julia will typically allow you to redefine functions, but not types. The best approach is to put all the code we have gone through in a separate textfile, say `orddict.jl` and load the Julia REPL environment with it using `julia -i orddict.jl` on the command line.

## **When to Write Your Own Collections?**

A question that pops up when discussing implementation of your own collections is whether this has any practical application? Truth to be told you will not do this very often.

The primary reason for covering how this is done is educational:

1. It demonstrates important aspects of how the Julia language works.
2. It helps you understand better how the builtin Julia collections work and how you can best utilize them.

The most typical case of implementing collection interfaces is with the first example. We often have data structures such as our multi-stage rockets which where never intended to be generic collections but which nonetheless benefit from being able support some collection interfaces.

Being able to iterate easily over all the individual parts of a larger data structure tends to be very useful.

# Working with Sets

- **What** is a set?
- **Creating** a set.
- **Set vs Arrays**, how are sets different from arrays?
- **Operations**. Common set operations such as:
  - Union
  - Intersection
  - Difference

Explaining what a set is and showing what operations you can do with them does not take much time. What does take time is developing an understanding or intuition about what sort of problems you can solve with sets.

A lot of problems related to organizing and locating data can be solved beautifully by utilizing sets and set operations, but that is not always apparent.

In this chapter we are going to go through both what sets are and what you can do with them as well as looking at various realistic examples showing the power of storing data in sets.

## Motivation

A lot of software requires organizing large amounts of data. Some examples are:

- Photo albums.
- Email clients.
- Bug tracking systems.
- Online shopping.
- Software development projects.
- Specialist software like modeling software for geologists.

For a long time the most popular way of organizing data was tree structures. To find an item you would drill down into subcategories until you found what you were looking for. The problem with this approach is that many items can potentially exist underneath multiple subcategories not just one.

A Web-shop such as McMaster-Carr which sells a huge number of mechanical components is a great example of this problem.

On the side you can see various categories for screws:

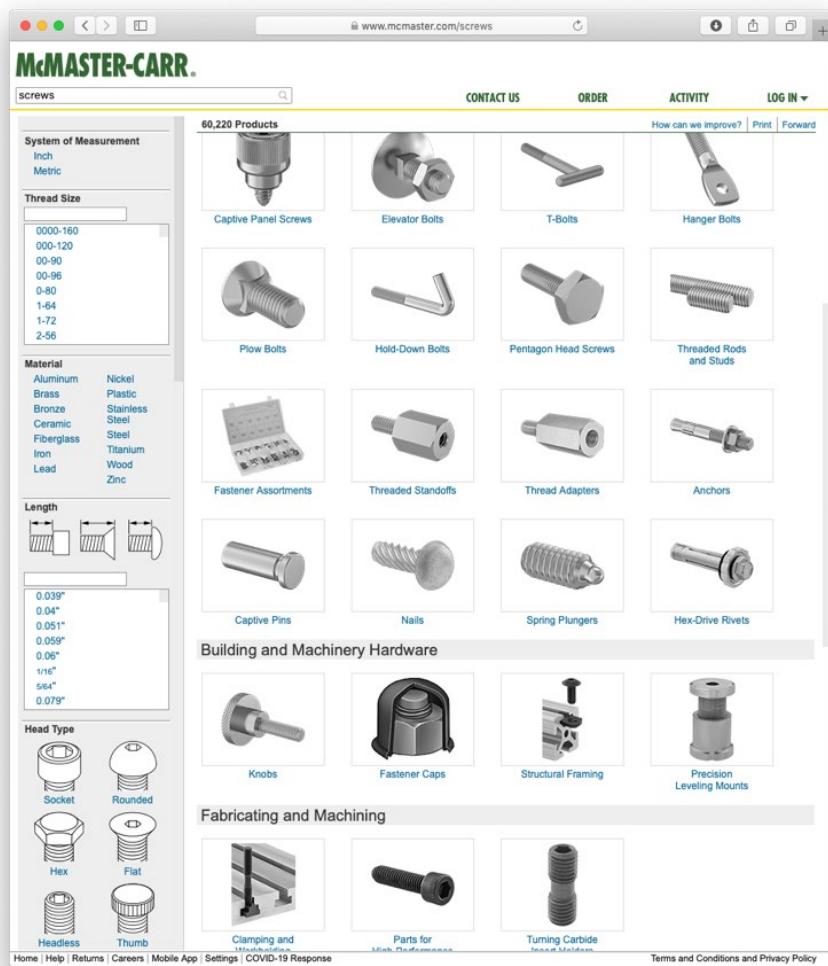


Figure 38: McMaster-Carr Web-shop showing selection of screws

- System of measurement. Is the screw dimensions given in inches or metric?
- Thread size.
- Material. Is the screw made of steel, plastic or wood?
- Length.
- Head type. Is it flat, rounded or hex shaped?

There are far more categories than I have shown here. The point however is that you cannot turn this into a tree hierarchy. Both a plastic screw and a steel screw can have a rounded head e.g.

Another case is photo albums. How do you organize them? You could organize pictures by every family member, so that your wife, and each child get their own album. Or maybe albums based on events such as visiting Barcelona or Hawaii makes more sense? Or maybe one wants more activity based organization such as albums for particular type of attractions such as technical museums or zoos.

Organization is difficult, so let us look at how sets and set operations can help us achieve this task.

## What is a Set?

A set is a collection type just like arrays or dictionaries. Here is an example of creating a set:

```
julia> fruits = Set(["apple", "banana"])
Set(["banana", "apple"])

julia> fruits = Set(["apple", "banana", "pear", "orange"])
Set(["pear", "orange", "banana", "apple"])

julia> fruits = Set([:apple, :banana, :pear, :orange])
Set(Symbol[:pear, :apple, :banana, :orange])

julia> odds = Set([1, 3, 5, 7, 9])
Set([7, 9, 3, 5, 1])
```

In the first case we create a set of fruits, where each fruit is represented by a string. The second case is similar but we use symbols instead. It is a useful example since symbols are often used in Julia to represent keys.

Sets are like the keys of a dictionary, no element occurs twice. Like a dictionary, the elements don't exist in any particular order. When you iterate over a set, elements will appear in a specific order. However you have no control over this order. If you add or remove elements the order can change.

This is different from say an array where you have full control over how the adding and removal of elements affect the order of the array. If you add an element to an array using `push!` then every element stays in the same position as before. Every element can be accessed with the exact same index as previously.

## Sets vs Arrays

We can get a better sense of what a set is by comparing its properties with that of an array.

Property	Sets	Arrays
Duplicates allowed	No	Yes
Elements ordered	No	Yes
Random access	No	Yes
Quick membership test	Yes	No
Iterable	Yes	No

The desirable properties offered by sets, are that you are guaranteed to have no duplicates and it is quick to check whether it contains a specific object.

The reason for this is because determining whether an array contains an element or not requires looking at every element in the array. So looking for a particular element in an array of 2 million elements will on average take twice as long as looking for it in an array of 1 million elements.

We call this a *linear* relationship. However for a set the number of operations required to locate an element does not grow with the size of the set. There is no linear relationship.

Although sets can be implemented in different ways, thus in some variants it requires  $\log(n)$  checks on average to lookup an element in set of  $n$  elements.

To help you better understand the benefits of using sets, let us make some comparisons of sets with arrays for different types of operations which sets are optimized for.

## Sorted Search in Arrays

In a sorted array you can perform a *binary search*. Here is a quick idea of how that works. Consider the sorted array of numbers below:

```
A = [2, 7, 9, 10, 11, 12, 15, 18, 19]
```

It has 9 numbers. Say you are looking for the number 18. Normally that would require 8 comparisons to locate, but with binary search you begin in the middle  $A[5] == 11$  and ask if  $18 > 11$  or if  $18 < 11$ .

Because it is sorted, we can conclude 18 is somewhere in the range 6:9 (index rage). This is repeated by checking the middle of this range. Since we don't have a middle here we could round down the index to  $A[7] == 15$ . We find that 18 is above this value. Hence in 3 comparisons rather than 8 we locate the answer.

Julia has several functions for doing this:

```
julia> searchsorted(A, 18)
8:8
```

```
julia> searchsortedfirst(A, 18)
```

8

```
julia> searchsortedlast(A, 18)
8
```

The downside of using sorted arrays is that the programmer has to make sure the array is sorted at all times. It makes insertions slow, as you must re-sort the array each time. Sets have the benefit that they allow not only fast checks on membership (is this element in the set?), but also fast insertion and removal.

### Membership Test

We can turn the array A into a set S. Both support membership test with `in` or its greek letter equivalent `∈`. You can also use `⊆` or `issubset` to check if multiple elements are members.

```
julia> 18 in S
true
```

```
julia> 18 ∈ A
true
```

```
julia> [11, 15] ⊆ A
true
```

```
julia> issubset([11, 15], S)
true
```

Arrays look similar in behavior, but these operations will happen faster on a set. The exception is for small collections. With few elements, no collection is as fast as an array.

However it is still advisable to use sets for small collections if maintaining a unique set of elements is important and order isn't. It helps *communicate* to the reader of your code, how it is supposed to work.

Using more sophisticated collection types such as `Dictionary` or `Set` really starts to pay off once you have large number of elements.

### No Duplicates

What happens when you attempt to create a `Set` with duplicates.

```
julia> apples = ["apples", "apples", "apples"]
3-element Array{String,1}:
 "apples"
 "apples"
 "apples"
```

```
julia> appleset = Set(apples)
Set(["apples"])
```

```
julia> length(appleset)
```

```

1
julia> numbers = [1, 1, 1, 2, 2, 2, 3, 3];
julia> length(numbers)
8

julia> S = Set(numbers)
Set([2, 3, 1])

julia> length(S)
3

```

Duplicates are allowed in arrays but not in sets.

### **Random Access and Ordering**

We will create a set and an array, with the same elements to demonstrate how random access and ordering is entirely different.

```

A = [3, 5, 7]
S = Set(A)

```

If we use `collect` or `foreach` they will iterate over the collections. You can see the order is different.

```

julia> collect(S)
3-element Array{Int64,1}:
 7
 3
 5

julia> collect(A)
3-element Array{Int64,1}:
 3
 5
 7

julia> foreach(print, S)
735

julia> foreach(print, A)
357

```

I am able to use brackets to access array elements by index.

```

julia> A[2]
2

```

But this is not possible to do with a set.

```

julia> S[2]
ERROR: MethodError: no method matching getindex(::Set{Int64}, ::Int64)

```

With an array `push!` add each elements to a predictable location.

```
julia> push!(A, 9)
4-element Array{Int64,1}:
 3
 5
 7
 9
```

However for sets, the element can end up anywhere.

```
julia> push!(S, 9)
Set([7, 9, 3, 5])
```

With an array, `pop!` will remove the last element added.

```
julia> pop!(A)
9
```

However with a Set this operation is best avoided, as you have no control over what element you actually end up removing.

```
julia> pop!(S)
7
```

In this case it may have been more appropriate to throw an exception, rather than letting the user perform `pop!`.

## Set Operations

Set operations are used to combine sets to create new sets.

However set operations are not actually limited to sets. You can perform set operations between arrays as well. The difference is that sets were designed to support it while arrays are not. Arrays only perform set operations efficiently for small collections of elements.

Set operations allow you to answer questions such as:

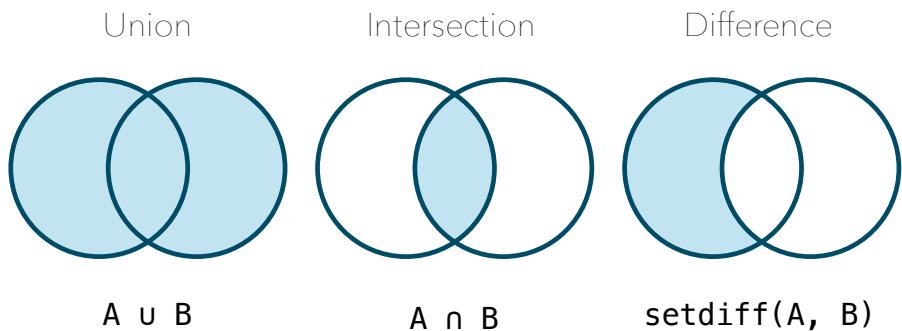
Give me the pictures of Bob when he visited Spain and Greece

If Bob represents all images in your Photos application of your uncle Bob and Spain is a set of all your Spain pictures and Greece is a set of all your pictures from Greece, then such a question can be answered with either of these two equivalent expressions:

`S = Bob ∩ (Spain ∪ Greece)`  
`S = intersect(Bob, union(Spain, Greece))`

This demonstrates the usage of the `union` and `intersect` operations. These can also be written using the  $\cup$  and  $\cap$  symbols.

The best way to visualize the behavior of the different set operations is to use Venn diagrams<sup>17</sup>.



The two circles in each example represents the sets A and B. These are overlapping sets, meaning some of the elements in A also exists in B.

The area colored in blue shows which elements are included in the set resulting from the set operation. For instance with a *set union*, all the elements in A and B are included in the result. For *set intersection* however only elements shared between A and B are part of the result.

With *set difference* the order is important. `setdiff(A, B)` returns the elements in A which remain when you *remove* elements in A, which also exist in B.

Let us look at a practical example of how this is used. We imagine having some sets of photo captions.

```
bob = Set(["Bob in Spain", "Bob in Greece", "Joe and Bob in Florida"])
joe = Set(["Joe in Texas", "Joe and Eve in Scotland", "Joe and Bob in Florida"])
eve = Set(["Eve in Wales", "Joe and Eve in Scotland", "Eve in Spain"])
```

So we have three people Bob, Joe and Eve who have been on various vacations abroad where they have taken pictures. In this case we are representing those pictures as their caption text.

We want to use set operations to find pictures where they have been together. bob is a set of all pictures Bob has been in, while joe is a set of all pictures Joe has been etc.

This finds pictures where Bob and Joe were together on vacation.

```
julia> bob ∩ joe
Set(["Joe and Bob in Florida"])
```

Perhaps Eve broke up with Joe, and don't want pictures with Joe in them. Eve can then use `setdiff` to exclude Joe pictures.

```
julia> setdiff(eve, joe)
Set(["Eve in Wales", "Eve in Spain"])
```

Perhaps Joe wants to find all vacations he spent together with somebody else?

---

<sup>17</sup>Venn diagrams are usually used to illustrate the logical relationships between two or more sets of items.

```
julia> (bob ∪ eve) ∩ joe
Set(["Joe and Bob in Florida", "Joe and Eve in Scotland"])
```

Sets can of course contain any kind of object. Let us do some slightly less exciting set operations with numbers. A is a set of mostly even numbers, while B contains mostly odd numbers.

```
A = Set([1, 2, 4, 6])
B = Set([1, 3, 5, 6])
```

We can get the set intersection in two different ways:

```
julia> A ∩ B
Set([6, 1])
```

```
julia> intersect(A, B)
Set([6, 1])
```

And how about the set union:

```
julia> A ∪ B
Set([4, 2, 3, 5, 6, 1])
```

```
julia> union(A, B)
Set([4, 2, 3, 5, 6, 1])
```

And finally we can get the set difference of A and B.

```
julia> setdiff(A, B)
Set([4, 2])
```

```
julia> setdiff(B, A)
Set([3, 5])
```

As you can see order matters with set difference.

## How to Use Sets in Your Code

The basic operations of sets are not hard to learn. What takes more time to get a sense of is when you can use sets in your code.

Many times I have been surprised by how sets can often provide powerful and elegant solutions to difficult problems. It is very easy to forget that set are lying in your toolbox.

We will look at some problems where you can use sets as well as alternative solutions not utilizing sets.

## Searching for Products

When dealing with products in say a Web-shop, we would typically use an SQL<sup>18</sup> database. However conceptually what is being done is very similar to set operations, which is why we will expand on the example of buying screws from an online hardware shop.

A screw can have different head types:

```
head_type = [rounded, flat, headless, tslot]
```

You may want a flat one if you want the screw flush with the surface. Headless screws are used for things like set screws for axel-collars. Although for our purpose it is not important to know what all these different head types are for.

The screw can have a drive style, which indicates what kind of tip you need to have on your screwdriver to turn the screw around.

```
drive_style = [hex, phillips, slotted, torx]
```

Material should be obvious:

```
material = [aluminium, brass, steel, plastic, wood]
```

Now these are list of categories. Each item in the list is actually a Set. The set contains a product number uniquely identifying that screw. For practical reasons we are just going to invent some 3-digit product numbers.

We will use ranges to quickly create a large number of product numbers.

```
rounded = Set(100:4:130)
flat    = Set(101:4:130)
headless = Set(102:4:130)
tslot   = Set(103:4:130)

hex      = Set(100:108)
phillips = Set(109:115)
slotted  = Set(116:121)
torx     = Set(122:129)

aluminium = Set(100:3:120)
brass    = Set(101:3:120)
steel    = Set(102:3:120)
plastic  = Set(121:2:130)
wood     = Set(122:2:130)
```

If you look carefully at the numbers, you will see that they are overlapping. E.g. some of the aluminum product numbers are the same as the hex product numbers.

With these sets defined I can ask various useful questions such as:

---

<sup>18</sup>Structured Query Language (SQL), is a specialized language for formulating database queries. A query is a request for data in a database matching one or more criteria.

Give me the screws in your product catalog which have a rounded head, made of wood and which I can fasten with a torx screwdriver.

Answering this requires just a simple set operation:

```
julia> intersect(rounded, torx, wood)
Set([128, 124])
```

Or how about getting all steel screws which can be fastened with a Phillips screwdriver?

```
julia> intersect(phillips, steel)
Set([114, 111])
```

Or maybe you just want to know whether t-slot screws which are not made of plastic exists.

```
julia> setdiff(tslot, plastic)
Set([119, 103, 107, 111, 115])
```

This is one way of using sets, but you can accomplish the same with entirely different designs not utilizing sets at all. Instead we could define a screw as a richer data type with properties for each attribute.

```
struct Screw
    prodnum::Int
    headtype::HeadType
    drivestyle::DriveStyle
    material::Material
end
```

Instead of dealing with screws as just a number, we have a data type with properties, which we can potentially attempt to match some given search criteria.

You can see we have made the various properties to be represented by custom types `HeadType`, `DriveStyle` and `Material`. We could have made these strings or symbols, but instead we made them particular types to catch cases where you assign an illegal category to any of the attributes.

## Enumerations

To do this we use enumerations, or `enum` for short. These exist in a number of different languages. In Julia they are a bit peculiar because they are defined using macros.

```
@enum HeadType rounded flat headless tslot
@enum DriveStyle hex phillips slotted torx
@enum Material aluminium brass steel plastic wood
```

The giveaway is the `@` prefix. You can think of `hex`, `slotted` and `torx` as instances of the `DriveStyle` type. In fact you can use the `DriveStyle` constructor to create them.

```
julia> DriveStyle(2)
slotted::DriveStyle = 2

julia> DriveStyle(3)
torx::DriveStyle = 3

julia> DriveStyle(4)
ERROR: ArgumentError: invalid value for Enum DriveStyle: 4
```

However you can see the added type-safety in the last example. It is not possible to create other values for `DriveStyle` than the ones specified when the *enumeration* was defined.

### Creating Test Data

To demonstrate how we can use this type to locate screws with different properties we need to create some test data to operate on.

```
function make_screw(prodnum)
    headtype = rand(instances(HeadType))
    drivestyle = rand(instances(DriveStyle))
    material = rand(instances(Material))

    Screw(prodnum, headtype, drivestyle, material)
end

screws = map(make_screw, 100:150)
```

This code create an array of screws with product numbers in the range 100 to 150. We pick values for each property at random. The `instances` function returns an array of every possible value for an enumeration.

```
julia> instances(DriveStyle)
(hex, phillips, slotted, torx)

julia> instances(Material)
(aluminium, brass, steel, plastic, wood)
```

### Searching for Screws

Instead of using set operations we search through every screw and look for screws matching a set of criteria. Let us define a search criteria for screws with rounded heads made of wood:

```
function isroundwood(screw)
    screw.headtype == rounded &&
    screw.material == wood
end
```

We can then use this predicate (function returning a boolean value) to filter screws:

```
julia> roundedwood = filter(isroundwood, screws)
```

```
3-element Array{Screw,1}:
Screw(100, rounded, torx, wood)
Screw(113, rounded, slotted, wood)
Screw(129, rounded, torx, wood)
```

How about finding what non-plastic t-slot screws are offered in the store?

```
julia> function isnonplastic(screw)
           screw.headtype == tslot &&
           screw.material != plastic
       end

julia> nonplastic = filter(isnonplastic, screws)
15-element Array{Screw,1}:
Screw(105, tslot, hex, wood)
Screw(106, tslot, hex, wood)
Screw(107, tslot, hex, brass)
Screw(108, tslot, phillips, steel)
Screw(117, tslot, phillips, wood)
Screw(118, tslot, hex, wood)
Screw(125, tslot, phillips, wood)
Screw(128, tslot, phillips, wood)
Screw(130, tslot, phillips, wood)
Screw(131, tslot, torx, brass)
Screw(133, tslot, hex, wood)
Screw(134, tslot, slotted, wood)
Screw(138, tslot, hex, steel)
Screw(141, tslot, phillips, steel)
Screw(146, tslot, torx, brass)
```

The best solution for your case is not always easy to determine. But it is worth knowing about different approaches. Sometimes it makes sense to combine solutions. You can put these screw objects into sets as well.

### **Putting Screw Objects into Sets**

We can use the `filter` function to produce sets which can be reused later.

```
julia> issteel(screw) = screw.material == steel;
julia> steel_screws = Set(filter(issteel, screws));

julia> ishex(screw) = screw.drivestyle == hex
julia> hex_screws = Set(filter(ishex, screws))
```

We can then use these sets in set operations.

```
julia> steel_screws ∩ hex_screws
Set(Screw[
    Screw(126, headless, hex, steel),
    Screw(115, headless, hex, steel),
    Screw(121, flat, hex, steel),
    Screw(107, headless, hex, steel),
    Screw(108, flat, hex, steel)] )
```

])

### Using Dictionaries

However this solution can be further improved upon. Frequently buyers know the product number and want to lookup directly on the product number. This has to be the quickest lookup. It should not require a search.

```
julia> screwdict = Dict(screw.proignum => screw for screw in screws)
Dict{Int64,Screw} with 51 entries:
  148 => Screw(148, rounded, hex, brass)
  124 => Screw(124, rounded, hex, aluminium)
  134 => Screw(134, tslot, slotted, wood)
  136 => Screw(136, rounded, torx, aluminium)
  131 => Screw(131, tslot, torx, brass)
  144 => Screw(144, rounded, slotted, steel)
  142 => Screw(142, flat, slotted, steel)
  150 => Screw(150, rounded, hex, steel)
  ...
julia> screwdict[137]
Screw(137, headless, phillips, aluminium)

julia> screwdict[115]
Screw(115, flat, phillips, aluminium)
```

This allows us to get back to the original solution where we utilize product numbers in our sets. Let us first make some new sets based on product numbers.

```
prodnums = keys(screwdict)

function isbrass(prodnum)
    screw = screwdict[prodnum]
    screw.material == brass
end
brass_screws = Set(filter(isbrass, prodnums))

function istorx(prodnum)
    screw = screwdict[prodnum]
    screw.drivestyle == torx
end
torx_screws = Set(filter(istorx, prodnums))
```

Now we are back to the elegance of using set operations to pick desired products, based on product keys in sets.

```
julia> brass_screws ∩ torx_screws
Set([100, 122, 144])

julia> [screwdict[pn] for pn in brass_screws ∩ torx_screws]
3-element Array{Screw,1}:
 Screw(100, rounded, torx, brass)
```

Screw(122, tslot, torx, brass)  
 Screw(144, flat, torx, brass)

## Bug Tracker

When developing larger pieces of software, particularly in a team, companies will usually utilize some form of Bug tracking tool. Commonly these are web applications which allow testers to submit descriptions of bugs.

Managers or product specialists may then review these bugs and assign priorities and severity, before the bugs finally gets assigned to software developers.

Some common attributes recorded with a bug might be:

- **Project.** What software project is it part of.
- **Priority.** How important is this bug to fix?
- **Severity.** Is it a minor annoyance or a crash in critical functionality?
- **Component.** Is this in a user interface, client, server etc?
- **Assignee.** Who is assigned to deal with the bug currently?

Just like with products, bugs will usually be uniquely identified by a bug number. Thus much the same kind approach as described before can be used. You can have bugs in dictionaries where the keys are the bug numbers.

We can define sets composed of different bug numbers. Here are some questions we can imagine being solved using sets.

What are the most critical bugs assigned to Bob in the Lunar Lander project?

`bob ∩ critical ∩ lunar_lander`

It may not be practical to have names for each set like this, and we want sets organized more according to the fields in the bug tracker. We could utilize dictionaries to group related sets.

`assignees ["Bob"] ∩ severity[:critical] ∩ projects["Lunar Lander"]`

When doing the set operation on multiple objects, it may be more practical to not use the operator symbols. This is equivalent:

`intersect(assignees["Bob"], severity[:critical], projects["Lunar Lander"])`

A manager may ask:

What top priority bugs are being handled by Bob and Eve?

`assignees ["Bob"] ∪ assignees["Bob"] ∩ priorities[:top]`

We could have looked at many more examples. But hopefully this gives you a good idea of how you could utilize sets to simplify problems in your applications.



# Your Own Spreadsheet

- Working with numbers in **Tables**, and performing calculations.
- **Slicing and Dicing**. Extracting subsections of an array of different rank (number of dimensions).
- **Concatenation**. Combining arrays in different ways to create new and larger ones.
- What is **Linear Algebra** and how does it relate to **Matrix Multiplication**?

In the Working with Numbers chapter we showed how Julia could replace your calculator. Following that analogy this chapter is about replacing your spreadsheet application such as MS Excel or Numbers with Julia.

Spreadsheets are about working with numbers in tables, that is dealing with values in rows and columns.

A mathematician however will not talk about tables but about matrices. A table column is called a *column vector* and a row is called a *row vector*.

#### **NOTE Vectors and Matrices in Mathematics**

A matrix or a vector is not just a dumb container of numbers. For instance in mathematics both sets, tuples and vectors may look like a list of numbers and hence seem similar. But what you can do with them is different.

So this is very similar to types in programming. An Array, Tuple or Set object in Julia may contain exactly the same numbers, but what you can do with them differs.

A mathematician would call them *mathematical objects* because they are not just data but also have invariants that must be true as well as operations which must be defined.

row vector

5	7	8	9
---	---	---	---

5
7
8
9

matrix

11	12	13	14
21	22	23	24
31	32	33	34

column vector

This is part of the field of mathematics called *Linear Algebra*. In Linear Algebra we call single values such as 1, 4, and 8 **scalars**. While multiple values in a row or column are **vectors**, tables are **matrices** and if the data is arranged in a 3D-array they may be referred to it as a **cube**.

#### NOTE Tensors

If you end up working on Deep Learning<sup>19</sup> you will encounter yet another terminology which is **tensors**<sup>20</sup>. This is a generalization where a scalar is a 0-dimensional tensor. A vector is a 1-dimensional tensor, a matrix a 2-dimensional tensor and so on.

Tensors can exist for any number of dimensions. Tensors and matrices are not the same thing, but they are often treated as if they are the same in *Machine Learning* circles.

In this chapter we will look at different ways of creating arrays of different dimensions as well as looking at their properties and what you can do with them.

## Spreadsheet Calculations

One of the first things we did in this book was to use the `map` function to apply a function to every element in an array. For instance I can get the square root of multiple elements like this: `map(sqrt, [4, 9, 16])`. What we didn't show then is that `map` can also be used for functions taking multiple arguments. `+` can be used as a function.

```
julia> 3 + 4
7
```

```
julia> +(3, 4)
7
```

In fact every operator in Julia is like a function call. This allows us to pass `+` or `*` to the `map` function.

```
julia> map(+, [2, 3], [3, 1])
2-element Array{Int64,1}:
 5
 4
```

```
julia> map(*, [2, 3], [10, 100])
2-element Array{Int64,1}:
 20
 300
```

Because this is so useful there is a shorthand for doing these kinds of calculations, by adding a dot `.`, to an operator when you want it apply it to individual elements.

```
julia> [2, 3] .+ [3, 1]
2-element Array{Int64,1}:
 5
 4
```

```
julia> [2, 3] .* [10, 100]
2-element Array{Int64,1}:
 20
 300
```

This is very practical when working with data in a spreadsheet. Suppose we have a shopping list as shown below. Every row in the table is a different item. It could be potatoes, beans, steak or a flamethrower. Who knows?

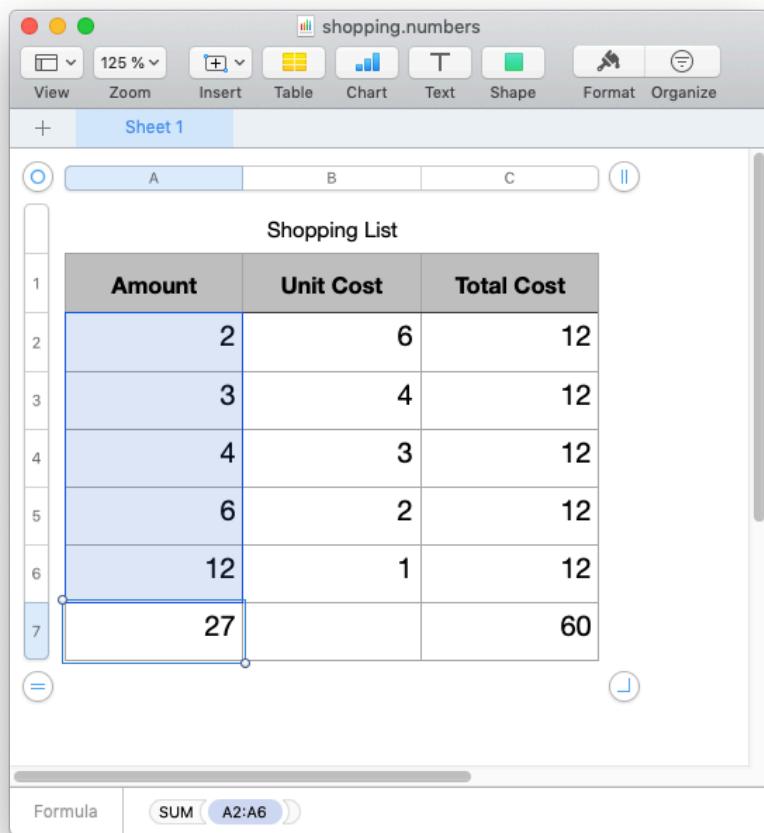
You don't need to use a spreadsheet to do these kinds of calculations. We can use Julia arrays instead. This is how we would represent table columns:

```
amount = [2, 3, 4, 6, 12]
unit_cost = [6, 4, 3, 2, 1]
```

However if you want a row, you separate values with white space<sup>21</sup> rather than comma.

---

<sup>21</sup>This is the convention you will find in mathematics in *Linear Algebra* as well.



The screenshot shows a spreadsheet application window titled "shopping.numbers". The toolbar includes View, Zoom, Insert, Table, Chart, Text, Shape, Format, and Organize. The main area displays a table titled "Shopping List" with columns for Amount, Unit Cost, and Total Cost. The table has 7 rows, with rows 2 through 6 having identical values (Amount: 2, 3, 4, 6, 12) and row 7 being a summary (Amount: 27, Total Cost: 60). The formula bar at the bottom shows "SUM A2:A6".

	Amount	Unit Cost	Total Cost
1			
2	2	6	12
3	3	4	12
4	4	3	12
5	6	2	12
6	12	1	12
7	27		60

Figure 39: Shopping list in a spreadsheet. Calculating cost of buying different number of items with different unit price.

```
eggs = [2 6 12]
spam = [3 4 12]
```

Above is example representing the two top rows. Here we assume that the first line represent buying two packs of egg costing say 6 euro each. The second row is buying three cans of spam costing 4 euro each.

Notice how the Julia REPL is helpful in visualizing the difference between a column vector and a row vector.

```
julia> amount
5-element Array{Int64,1}:
 2
 3
 4
 6
12

julia> eggs
1×3 Array{Int64,2}:
 2 6 12
```

Using `.*` we can do an element-wise multiplication for each product we are buying to get the cost of buying each item.

```
amounts .* unitcosts
5-element Array{Int64,1}:
12
12
12
12
12
```

We can then add up this whole column of row sums, just like in a spreadsheet application:

```
julia> sum(amounts .* unitcost)
60
```

## Row Based Matrix Construction

If you want to, you can even create this whole table from scratch in Julia in one statement. Such a table is define one row at a time. We separate each row with a semicolon `;`.

```
julia> table = [2 6 12;
               3 4 12;
               6 2 12;
               12 1 12]
4×3 Array{Int64,2}:
 2 6 12
 3 4 12
 6 2 12
12 1 12
```

## Column Based Matrix Construction

Instead of creating an array by specifying a list of rows separate by semicolon ;, you can specify a list of columns instead.

```
julia> x1 = [2, 3, 6, 12]
julia> x2 = [6, 4, 2, 1]
julia> x3 = [12, 12, 12, 12]
julia> table = [x1 x2 x3]
4×3 Array{Int64,2}:
 2  6  12
 3  4  12
 6  2  12
12  1  12
```

This is identical to writing the column vectors inline like this:

```
table = [[2, 3, 6, 12] [6, 4, 2, 1] [12, 12, 12, 12]]
```

---

Notice how Julia gives us a summary of what kind of `Array` we are getting as a result with the line `4×3 Array{Int64,2}`. This tells us that Julia made an array with 4 rows and 3 columns, where each element is of type `Int64`.

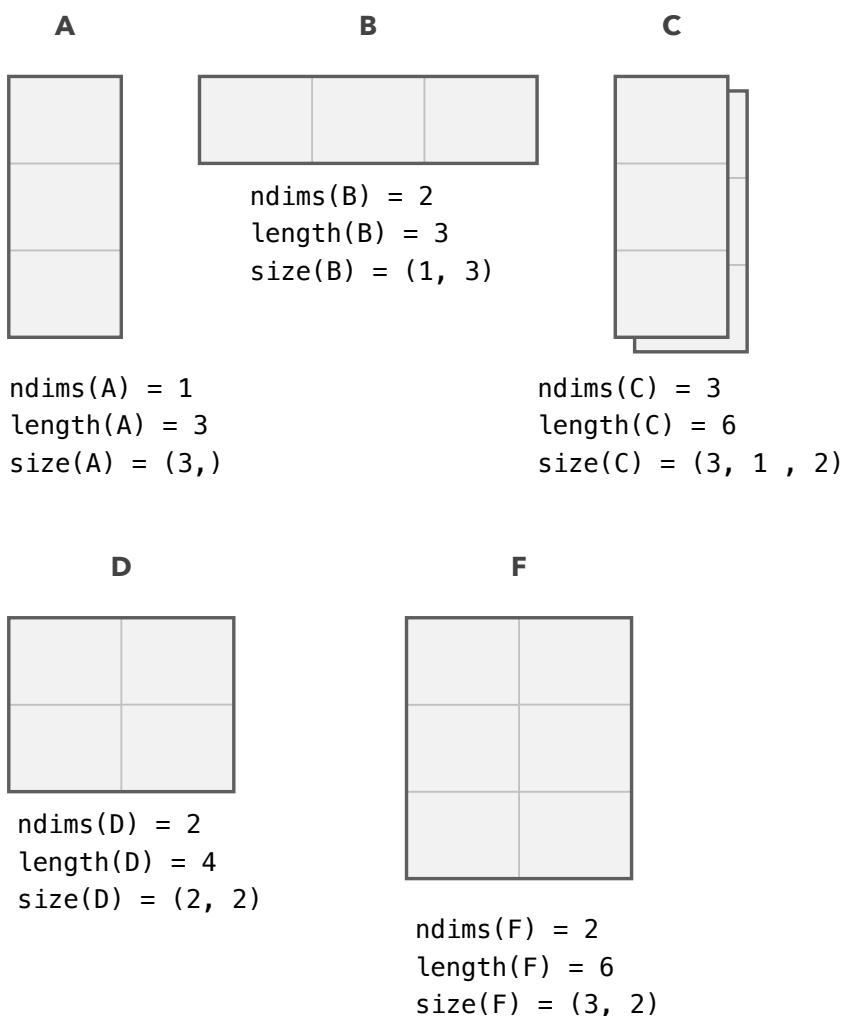
You can query an arbitrary array about these properties:

```
julia> eltype(table)
Int64

julia> size(table)
(4, 3)

julia> ndims(table)
2
```

`eltype` gives us the type of each element in the array. `ndims` tells us the number of dimensions, while `size` tells us the number of components (elements) along each dimension. Normally we think of dimensions as length, height and depth, but in this case we will normally speak of rows and columns.



## Size, Length and Norm

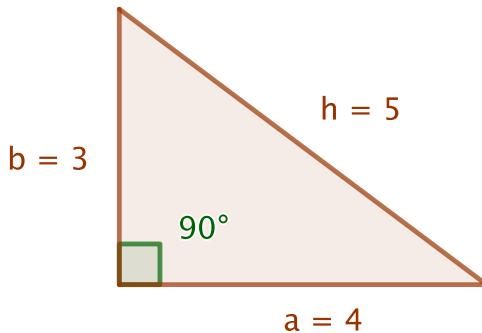
If you come from other programming languages it can be easy to confuse these array concepts:

- `size` the dimensions of an array.
- `length` total number of elements in array.
- `norm` magnitude of vector.

```
julia> length(table)
12
```

`norm` is easiest to grasp with a simpler vector.

```
julia> norm([3, 4])
5.0
```



Looking at a right-angled triangle will help you visualize what `norm` is doing. You can think of the elements of the vector as the sides  $a$  and  $b$  in the triangle. `norm` gives us the length of the longest side, the *hypotenuse*

$$5^2 = 3^2 + 4^2$$

The *Pythagoras' theorem* tells us the relationship between all the sides in a right-angled triangle. You can think of `norm` as applying the Pythagorean theorem to figure out the length of the *hypotenuse*.

Later we will explore how applying Linear Algebra to this kind of shopping list problem gives us a more elegant and flexible solution, than using broadcast (the dot).

## Slicing and Dicing

Julia has great support for selecting slices of arrays of different dimensions. This flexibility comes from the fact that the `setindex!` and `getindex` functions which are invoked when we use square brackets `[]` to access elements or assigning to them.

Previously we saw this flexibility when implementing our `OrdDict` collection.

Let us explore this further by first looking at accessing individual elements on a 1-dimensional array. Below is an overview of the different ways of doing this. While the illustrations shows rows, this applies equally well to column vectors.

Notice how there are many different ways of accessing the same elements. For an array `A` with starting index 1, which is the default in Julia, `A[3]` and `A[begin+2]` would represent the exact same element.

For an array with four elements, as in the first two examples, `A[4]` and `A[end]` refers to the same element. Likewise `A[3]` and `A[end-1]` grabs the same array element.

If you don't care about the specific index at all but just want all the elements you can write `A[:, :]`.

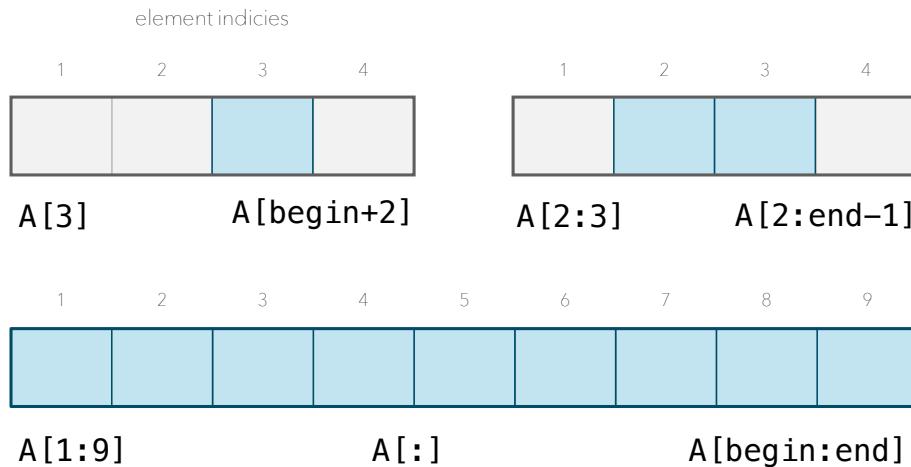


Figure 40: Slicing a 1-dimensional array  $\mathbf{A}$  in different ways.

Here are some examples demonstrating this in the Julia REPL:

```
julia> A = collect('A':'F')
6-element Array{Char,1}:
 'A'
 'B'
 'C'
 'D'
 'E'
 'F'

julia> A[begin+1]
'B': ASCII/Unicode U+0042

julia> A[end-1]
'E': ASCII/Unicode U+0045

julia> A[2:5]
4-element Array{Char,1}:
 'B'
 'C'
 'D'
 'E'

julia> A[begin+1:end-1]
4-element Array{Char,1}:
 'B'
 'C'
 'D'
 'E'
```

So far none of this should be too surprising as we have already used slices in

earlier examples. It gets more interesting when we are dealing with slices for multi-dimensional arrays, such as matrices.

Let us create a 2D matrix A to experiment on, using Julia's reshape function.

```
julia> A = reshape(1:12, 3, 4)
3x4 reshape(::UnitRange{Int64}, 3, 4) with eltype Int64:
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

The way this works is that we are giving reshape an `AbstractArray` of elements which we want to rearrange to have different dimensions than the original. We start with twelve elements from 1 to 12. And we rearrange these to be a 3x4 matrix, that is 3 rows and 4 columns.

#### **NOTE Matrix Shape**

The shape of a matrix is how many rows and columns it has. Hence the function for changing the number of rows and columns is called `reshape` in Julia. Keep in mind that the length of the matrix cannot be changed by `reshape`. So you cannot reshape a 10 element array into a 5x5 matrix. However you can reshape a 12 element array to a 3x4, 4x3, 2x6 or 2x3x2 matrix for instance.

The way I like to think about array slicing, is to think in terms of the set intersection operation  $\cap$ . Thus `A[2, 3]` can be read as:

Give me the intersection of all the elements for row 2, and all the elements of column 3.

The first figure below shows this. The light blue represents the row and columns we have selected, and the *darker blue* represent the intersection between these row and column selections.

This makes it easier to understand the selection `A[2:3, 2:4]`. We can read this as:

Give me the intersection of all the elements in row 2 to 3, and columns 2 to 4.

Following this logic it becomes apparent how you would select an entire row, or entire column in a matrix. We can experiment with this in the REPL:

```
julia> A[1, 2]
```

```
4
```

```
julia> A[3, 4]
```

```
12
```

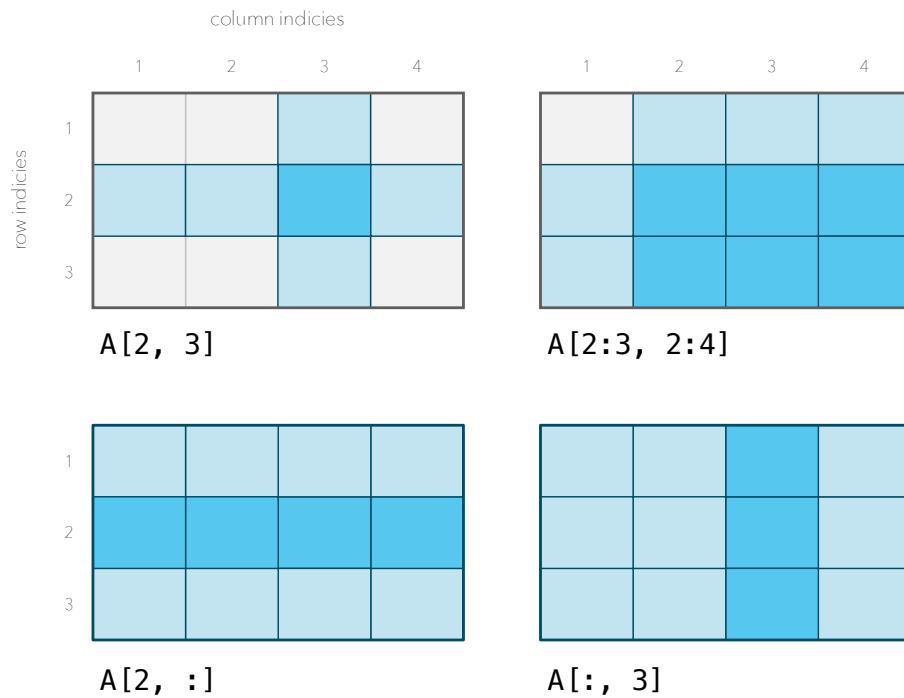


Figure 41: Slicing 2-dimensional arrays.

```
julia> A[:, 4]
3-element Array{Int64,1}:
 10
 11
 12
```

```
julia> A[2, :]
4-element Array{Int64,1}:
 2
 5
 8
 11
```

It is also worth noting that even multidimensional arrays can be treated as 1-dimensional ones.

```
julia> A[1]
```

```
1
```

```
julia> A[4]
```

```
4
```

## Linear Algebra: Spreadsheets on Steroids

In our introduction we calculated the total cost of our shopping list with the following expression:

```
sum(unit_cost .* amount)
```

However it turns out you can do exactly the same in a far more compact way using *matrix multiplication*. The catch is, we need to define our inputs slightly differently:

```
julia> unit_cost = [6 4 3 2 1]
1×5 Array{Int64,2}:
 6  4  3  2  1

julia> amount = [2, 3, 4, 6, 12]
5-element Array{Int64,1}:
 2
 3
 4
 6
12

julia> unit_cost * amount
1-element Array{Int64,1}:
 60
```

If your `unit_cost` already is a column vector, you can easily turn it into a row vector using the `transpose` function. `transpose` will swap rows and columns of a matrix or vector.

```
julia> A = [2 4 8]
1×3 Array{Int64,2}:
 2  4  8

julia> transpose(A)
3×1 LinearAlgebra.Transpose{Int64,Array{Int64,2}}:
 2
 4
 8

julia> transpose(transpose(A))
1×3 Array{Int64,2}:
 2  4  8

julia> B = [2, 4, 8]
3-element Array{Int64,1}:
 2
 4
 8

julia> transpose(B)
```

```
1×3 LinearAlgebra.Transpose{Int64,Array{Int64,1}}:
2 4 8
```

This gives you an idea of how transpose works for matrices:

```
julia> C = [11 12 13; 21 22 23]
2×3 Array{Int64,2}:
 11 12 13
 21 22 23

julia> transpose(C)
3×2 LinearAlgebra.Transpose{Int64,Array{Int64,2}}:
 11 21
 12 22
 13 23

julia> transpose(transpose(C))
2×3 Array{Int64,2}:
 11 12 13
 21 22 23
```

But what if I want to see the sum for each row rather than total for all the products I am buying?

You can do this by putting unit costs along the diagonal of a matrix. This may look weird, but we will get to why it works this way and how to think about matrix operations later.

```
julia> using LinearAlgebra

julia> unit_cost = [6, 4, 3, 2, 1]
5-element Array{Int64,1}:
 6
 4
 3
 2
 1

julia> Diagonal(unit_cost)
5×5 Diagonal{Int64,Array{Int64,1}}:
 6 ⋅···
 ⋅ 4 ⋅···
 ⋅· 3 ⋅·
 ⋅··· 2 ·
 ⋅···· 1

julia> Diagonal(unit_cost) * amount
5-element Array{Int64,1}:
 12
 12
 12
 12
 12
```

12

```
julia> transpose(unit_cost) * amount
60
```

To use the `Diagonal` matrix we need to use the builtin `LinearAlgebra` module in Julia. Hence using `LinearAlgebra` on the first line. Without it Julia would not know what `Diagonal` means.

This `Diagonal` matrix is not any different from defining an ordinary matrix like this:

```
julia> D = [6 0 0 0 0;
           0 4 0 0 0;
           0 0 3 0 0;
           0 0 0 2 0;
           0 0 0 0 1]
5×5 Array{Int64,2}:
 6  0  0  0  0
 0  4  0  0  0
 0  0  3  0  0
 0  0  0  2  0
 0  0  0  0  1
```

So why bother with the `Diagonal` matrix at all? There are two nice benefits to this:

1. It is quicker to write than to spell out every single zero, as shown for matrix `D`.
2. It gives faster code, and requires less memory. A `Diagonal` matrix only stores the values along the diagonal. When you use it in calculations, Julia knows all values except those on the diagonal are zero. It does not have to inspect every index to learn that.

For reasons like this there are lots of specialized matrix types in Julia. However you can treat them like normal matrices. Their main advantage is that they encode some special condition or property about the matrix which allows the Julia to take shortcuts when performing calculations on them.

When working with a spreadsheet you have a bunch of data or input and some operations you want to perform on that data, which is usually embedded in the spreadsheet itself.

We can imagine a spreadsheet containing the formulas for calculating the sum of shopping items. The problem with this approach is how to deal with multiple orders. Say different people are filing orders for different amount of items, and want separate bills and totals.

It would be cumbersome to duplicate the spreadsheet and then fill in the new data manually. Matrix multiplication on the other hand allows you to process multiple inputs as one operation:

```
julia> amounts = [6 1 10;
```

```

        4 1 10;
        3 1 10;
        2 1 10;
        1 1 10]
5x3 Array{Int64,2}:
6 1 10
4 1 10
3 1 10
2 1 10
1 1 10

julia> transpose(unit_cost) * amounts
1x3 Transpose{Int64,Array{Int64,1}}:
66 16 160

```

However you are not limited to processing multiple inputs (orders), you can also perform multiple operations on each input (order). In a Matrix multiplication  $\mathbf{A} \cdot \mathbf{X}$  it is common to think of  $\mathbf{X}$  as the input data and  $\mathbf{A}$  as the operations performed on that data. When working with just one input at a time, we may write  $\mathbf{Ax}$ , where  $\mathbf{x}$  is some input vector we want to compute a result for.

Every row in  $\mathbf{A}$  can be thought of as a different operation or function you want to perform on input  $\mathbf{x}$

**NOTE row and column vectors in mathematics**

In mathematics column vectors are thought of as input data or geometric objects, while row vectors are thought of as operations on these objects.

For instance we may want to calculate both the sum and the average number of items of each type for each order. That means dividing total number of items in one order by 5, which is the same as dividing every item amount by 5 and adding up, which in turn is the same as multiplying every item with 0.2 and adding up.

```

julia> operations = [6.0 4.0 3.0 2.0 1.0;
                     0.2 0.2 0.2 0.2 0.2];

julia> operations * amounts
2x3 Array{Float64,2}:
66.0 16.0 160.0
3.2   1.0   10.0

```

In this example the `operations` matrix contains two different operations to perform. The first operation is to calculate the total cost of all items in an order. Thus the first row contains unit costs.

The second row of `operations` contains 0.2 in every position because we want to calculate the average number of items ordered.

When we perform the multiplication, you can see that the first row in the results contains the total costs for each order, while the second row contains the aver-

age number of items of each type ordered. In the second order we got one item of each, so it makes sense that the average is 1, while in the last order we got 10 of each item. Thus the average should be 10 as well.

This example gives you a hint at how you can judge the shape of the output matrix when multiplying two matrices.

- The number of rows corresponds to the number of operations performed. So the number of rows in the operations has to match the number of rows in the output.
- Each column is an input vector. These are stored column-wise in the input. So the output has the same number of columns as the input data.

That is why order is important in matrix multiplication. Consider these two cases shown below.

$$\begin{array}{c}
 \mathbf{A} \\
 \left[ \begin{array}{c} 6 \\ 4 \\ 3 \end{array} \right]
 \end{array}
 \bullet
 \begin{array}{c}
 \mathbf{X} \\
 \left[ \begin{array}{c|c|c} 2 & 3 & 4 \end{array} \right]
 \end{array}
 =
 \begin{array}{c}
 \left[ \begin{array}{c|c|c} 12 & 18 & 24 \\ 8 & 12 & 16 \\ 6 & 9 & 12 \end{array} \right]
 \end{array}$$

Figure 42: Multiplying vector producing a matrix.

$$\begin{array}{c}
 \mathbf{A} \\
 \left[ \begin{array}{c|c|c} 2 & 3 & 4 \end{array} \right]
 \end{array}
 \bullet
 \begin{array}{c}
 \mathbf{X} \\
 \left[ \begin{array}{c} 6 \\ 4 \\ 3 \end{array} \right]
 \end{array}
 =
 \begin{array}{c}
 \left[ \begin{array}{c} 36 \end{array} \right]
 \end{array}$$

Figure 43: Multiplying vectors to produce a scalar.

In the first case **A** has 3 rows and **X** has 3 columns. That means we got 3 different operations being applied to 3 separate inputs. Hence we should get 3 rows in the output, one representing each operation, and there should be 3 columns, one for each input.

Thus we get a  $3 \times 3$  matrix as output when multiplying a  $3 \times 1$  matrix with a  $1 \times 3$  matrix. I say matrix rather than vector because it is perfectly valid to view column vector with 3 rows, as a matrix with 3 rows and 1 column.

In this case the operations are simple. The **first operation** is to multiply 6 to each of the inputs, which gives us 12, 18 and 24.

The **second operation** is to multiply 4 to each of the inputs, and so on.

---

In the second case we have one one row for **A**, which means only one operation will be performed.

Likewise there is only one column for **X** which means there is only one input. Thus the result has to be a single value. We only have one operation performed on one input, which must give a single number as output.

## How to Specify an Operation

When you are new to this, it is not obvious how to think about designing operations, or what you can even do, so let us look at some examples.

An operation has the same number of elements as there are elements in one of the inputs. So say we have 3 elements:  $x_1, x_2, x_3$  in the input.

One of the simplest things you can do is to specify a picker. An operation defined as [1 0 0] picks the  $x_1$ , while [0 1 0] would pick the  $x_2$ . Let us look at an example:

```
julia> A = [1 0 0];

julia> x = [8, 12, 3]
3-element Array{Int64,1}:
 8
12
 3

julia> A*x
1-element Array{Int64,1}:
 8

julia> A = [0 1 0];

julia> A*x
1-element Array{Int64,1}:
12
```

## Identity Operation

Thus if you don't want to do anything with the input you can setup your operations to be:

```
julia> A = [1 0 0;
           0 1 0;
           0 0 1];
```

```
julia> A*x
3-element Array{Int64,1}:
 8
12
 3
```

You see in this case, the first operation picks  $x_1$ , the second operation picks  $x_2$  and the third picks  $x_3$ . Thus we are back to where we started.

In Linear Algebra there is a shortcut for doing this. It is called the *Identity Matrix* and is usually written as a capitalized **I**. In Julia this is kind of magical. It magically becomes whatever size it needs to be:

```
julia> I*x
3-element Array{Int64,1}:
 8
12
 3
```

Notice, I never said anywhere what shape this **I** matrix should have.

## Reverse Element Order

What else can we do with this picker functionality? We can use it to rearrange arguments. Say you want to change the order of [8 12 3] to [3 12 8].

Numbered from top to bottom, this is what each operation would have to do:

1. Instead of picking the  $x_1$  value or the first value, we pick the last one, the  $x_3$  value.
2. Same pick as before.
3. Pick  $x_1$  instead of  $x_3$ . Pick the first value instead of the last in other words.

```
julia> A = [0 0 1;
           0 1 0;
           1 0 0];
```

```
julia> A*x
3-element Array{Int64,1}:
 3
12
 8
```

## Amplifying Elements

Another simple operation is to make elements larger. Say we want to double values, we can just multiple the picker matrix with the factor we want to enlarge the elements. Here we are doubling the elements.

```
julia> 2A
3x3 Array{Int64,2}:
 0  0  2
 0  2  0
```

```
2  0  0
```

```
julia> 2A*x
3-element Array{Int64,1}:
 6
24
16
```

So if we are using a picker then a value of 1 simply picks the element, while a value of  $k$  will multiply the element with  $k$ .

## The How and Why of Matrix Multiplication

At this point I hope I have made it clear that matrix multiplication can be useful, even if we have only scratched the surface.

However thus far I have not really explained how it works, or even *why* it works that way. Nor have I explained why this book is getting into all these details of about matrices. Most introductory programming books for other languages will not even mention multidimensional arrays. They will certainly not talk about *Linear Algebra*.

## Where is Matrix Math Used?

However Julia is a language that was made for scientists, researchers and mathematicians. In science matrices are a big deal. They are kind of the bread and butter for solving a lot of problems.

It is also of important historical significance for computers. What many will regard as the worlds first computer the Z1, was created primarily to perform matrix calculations. Its creator Konrad Zuse was struggling with solving math problems related to **wing flutter**<sup>22</sup> on airplanes.

Solving this problem requires performing calculations of massive matrices. That was very tedious to do by hand and so Zuse got the idea that he should make a machine to do it for him.

Today matrix math is a hot topic because of the rise of machine learning techniques such as **Deep Learning** which relies heavily on performing tons of matrix multiplications.

However if you should not care about Machine Learning, you may be interested in **3D graphics**. In 3D graphics matrices and vectors are also very important.

Or perhaps you want to pursue a career in **robotics**. Calculating and planning robot arm motions and placement relies extensively on matrix multiplication as well.

**Data science** is also fond of matrices. Data which has been recorded about a topic such as the spread of dangerous disease, sales numbers of a product,

---

<sup>22</sup>Wing flutter is vibrations of the wings which can cause the destruction of an airplane and thus smart to be able to understand and predict.

effect of a drug etc will often be collected in matrices and analyzed in matrix form. Data science is a field where people collect and study data to find trends and relationships which are useful to know about.

A common problem when learning mathematics is that your teacher will show you a solution to a problem, you did not even know was a problem in the first place. It is hard to follow an answer if you did not get to ponder the question first.

Matrix multiplication is a bit like that. It is quick to learn the mechanics of it, but it will still be puzzling because the way it works seems completely arbitrary.

To understand this requires you to get into the right frame of mind. Math is divided into things which are derived or concluded from other truths and things which are simply defined. You can deduce why  $3 + 4 = 7$  for yourself. However you cannot deduce that  $+$  means addition. That fact is not derived from anything else. It is simply something mathematicians have defined the symbol  $+$  to mean.

Mathematicians are constantly defining new operators and concepts. Whether something gets defined or not depends on whether mathematicians find some utility in this concept.

The matrix came about because mathematicians struggled with large number of linear equations like this:

$$\begin{aligned} 2x + y &= 3 + 2z \\ x - z &= y \\ y + x + 3z &= 12 \end{aligned}$$

The number of variables and the number of equations could get really large. It would become messy to write out the variable names over and over again, when what really mattered was the coefficients.

Somebody, then came up with the idea:

What if we order the variables in each equation in the same order?

That means we make sure that in every equation  $x$  with its coefficient comes first.  $y$  comes second and so on. Then our previous equations can be rearranged to this:

$$\begin{array}{rcl} 2x &+& y &-& 2z &=& 3 \\ x &-& y &-& z &=& 0 \\ x &+& y &+& 3z &=& 12 \end{array}$$

Because the variables are now clearly defined by their order, we don't really need to write them out anymore. This allowed mathematicians to come up with an alternative notation for describing this set of equations, a matrix and vectors:

$$\begin{bmatrix} 2 & 1 & -2 \\ 1 & -1 & -1 \\ 1 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 12 \end{bmatrix}$$

This gives us a far more compact representation. It also allows us to deal with multiple equations as a unit. We can give a *name* to each matrix. We do the same when programming. If you have 100 different values which are related, you don't create 100 different variables. Instead you create an array which can hold 100 elements. Mathematicians wanted the same flexibility.

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & -2 \\ 1 & -1 & -1 \\ 1 & 1 & 3 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 3 \\ 0 \\ 12 \end{bmatrix}$$

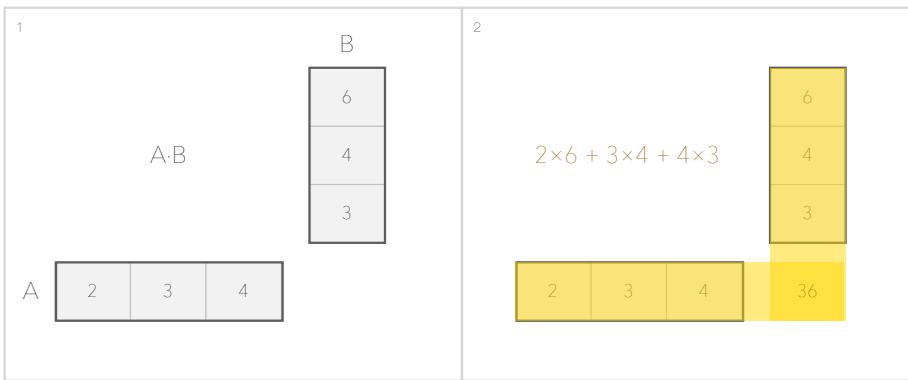
Then we could express the whole thing in a more compact form:

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$$

The mathematicians could have come up with any number of alternative notations for this. Why did they pick this one? What is the advantage of this form?

Before answering this question let us look at how matrix multiplication is done in different cases to get a clear sense of the mechanics of matrix multiplication.

We start with the simple case of multiplying a row vector with a column vector.



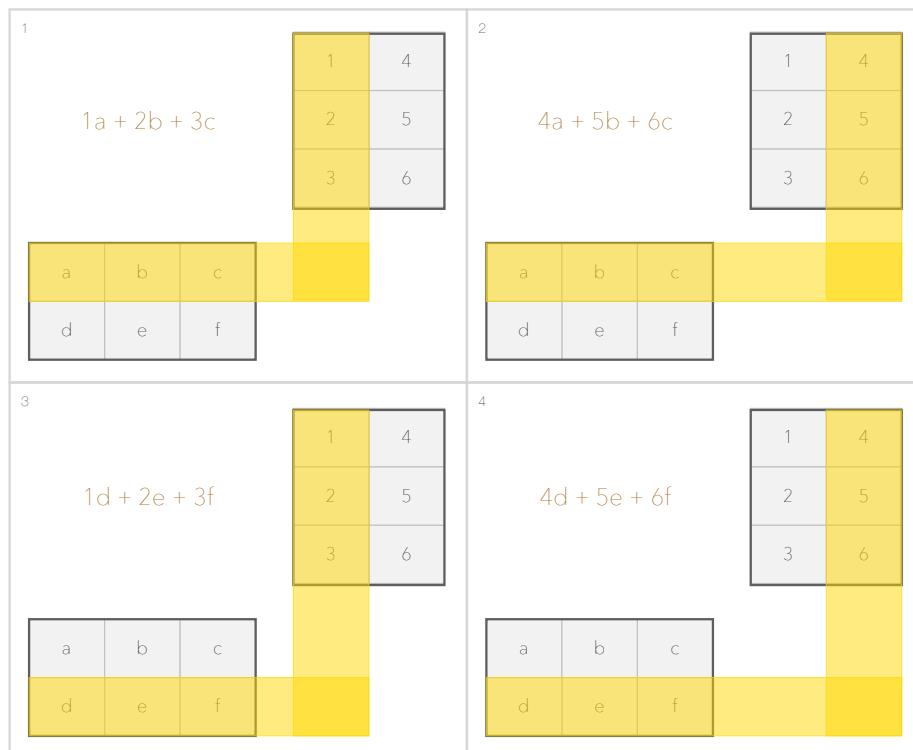
You see it is a combination of multiplications and additions. Every element in the first vector gets multiplied by the corresponding element in the other vector. Then the individual results are added up.

It gets more interesting when we swap the order.



Notice how every element in the result matrix is determined by combining rows and columns in a systematic fashion. This shows clearly why the number of columns in the output corresponds to the number of columns in the second argument, while the number of rows corresponds to the number of rows in the first argument.

When multiplying matrices rather than vectors, we end up with multiple outputs but now each element is not merely a simple multiplication. The value for each element in the output matrix is determined the same way as when we multiplied a single row vector with a single column vector.



This should help you understand why matrix multiplication cannot be done between two row vectors or two column vectors.

The way mathematicians have defined matrix multiplication is very practical, because it makes it easy to determine how many rows and columns should be in the output.

If it was not defined this way, then you could only multiply matrices with the exact same number of rows and columns with each other.

$$\begin{bmatrix} 2 & 1 & -2 \\ 1 & -1 & -1 \\ 1 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 12 \end{bmatrix}$$

This would have made it impossible to view the second matrix  $\mathbf{X}$  as input data and the first matrix  $\mathbf{A}$  as operations or functions to perform on that input.

We can think of every row as a function. So say the first row is defined as:

$$f(x, y, z) = 2x + y - 2z$$

We are interested in finding the values for  $x, y, z$  which are such that  $f(x, y, z) = 3$ . The second row could be another function defined as:

$$g(x, y, z) = x - y - z$$

The point is that we want to be able to apply each of these functions to the same input  $(x, y, z)$  and that would have been impossible if  $\mathbf{A}$  and  $\mathbf{X}$  had to have the same shape. We want to do different operations repeatedly on the same input.

Thinking in terms of matrices gives us very effective ways of solving these equations with multiple unknowns.

We got to look at some rules first to help us. Multiplying with the identity matrix is a bit like multiplying a scalar with 1.

$$\mathbf{A} \cdot \mathbf{I} = \mathbf{A}$$

Multiplying a matrix with its inverse matrix is similar to dividing by itself since it will produce the identity matrix.

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$$

We can compare this scalar operations such as  $a \cdot a^{-1} = 1$  which is the same as  $\frac{a}{a} = 1$ . These matrix concepts give us a way of expressing how a *system of linear equations*<sup>23</sup> can be solved.

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$$

We can multiply each side with the inverse matrix of  $\mathbf{A}$  to get the  $\mathbf{X}$  alone on the left side of the equation.

$$\mathbf{A}^{-1} \mathbf{A} \mathbf{X} = \mathbf{A}^{-1} \mathbf{B}$$

$$\mathbf{I} \mathbf{X} = \mathbf{A}^{-1} \mathbf{B}$$

$$\mathbf{X} = \mathbf{A}^{-1} \mathbf{B}$$

This means we can determine the values of  $x, y, z$  by finding the inverse matrix. Fortunately Julia has a function `inv` in the Linear Algebra module that does exactly that.

```
julia> A = [1  1 -2;
           1 -1 -1;
           1  1  3]
3×3 Array{Int64,2}:
 1  1  -2
 1 -1  -1
 1  1   3
```

---

<sup>23</sup>A system of linear equations is simply a number of related equations, where the unknowns are just combined with multiplication and addition. So no squaring, logarithms, square roots, sine or cosine.

```
julia> B = [3, 0, 12]
3-element Array{Int64,1}:
 3
 0
12
```

Let us calculate the X using the inverse of A:

```
julia> X = inv(A)*B
3-element Array{Float64,1}:
 4.2
 2.400000000000004
 1.8000000000000003
```

We can verify that this is correct by using the original expression:

```
julia> A*X
3-element Array{Float64,1}:
 3.0
 -4.440892098500626e-16
 12.0

julia> B
3-element Array{Int64,1}:
 3
 0
12
```

As you can see this is very close to what we should expect. Floating point numbers used by computers are not entirely accurate so one must expect tiny deviations.

## Combining Matrices and Vectors

Data does not always come in the shape and form you like to perform matrix operations on them. You may have  $n$  vectors but really wanted a matrix with  $n$  columns instead.

Fortunately Julia has a number of functions for concatenating matrices. This first example shows how we can concatenate two row vectors either horizontally using `hcat` or vertically using `vcat`.

1	3	5
---	---	---

A = [1 3 5]

7	9	11
---	---	----

B = [7 9 11]

1	3	5	7	9	11
---	---	---	---	---	----

hcat(A, B)

cat(A, B, dims = 2)

1	3	5
7	9	11

vcat(A, B)

cat(A, B, dims = 1)

The `cat` function allows you to specify along which dimension you are concatenating. This is useful if you are dealing with higher dimension arrays.

We can perform similar operations with column vectors.

1
3
5

A = [1, 3, 5]

7
9
11

B = [7, 9, 11]

1
3
5
7
9
11

1	7
3	9
5	11

hcat(A, B)  
cat(A, B, dims = 2)

vcat(A, B)  
cat(A, B, dims = 1)

The same principles applies to when combining matrices. We can concatenate along any dimension. Horizontal and vertical concatenation have their own functions `hcat` and `vcat` because they are done so frequently.

1	3	5
2	4	6

`A = [1 3 5; 7 9 11]`

7	9	11
8	10	12

`B = [7 9 11; 8 10 12]`

1	3	5	7	9	11
2	4	6	8	10	12

`hcat(A, B)`

`cat(A, B, dims = 2)`

1	3	5
2	4	6
7	9	11
8	10	12

`vcat(A, B)`

`cat(A, B, dims = 1)`

These concatenation functions can take any number of argument. You are not limited to two.

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
```

```

3

julia> y = [8, 6, 4]
3-element Array{Int64,1}:
 8
 6
 4

julia> hcat(x, y, x, y)
3×4 Array{Int64,2}:
 1  8  1  8
 2  6  2  6
 3  4  3  4

julia> hcat(x, 2y, 2x, 3y)
3×4 Array{Int64,2}:
 1  16  2  24
 2  12  4  18
 3   8  6  12

```

## Creating Matrices

When working with matrices we often need special kinds of matrices. Creating matrices with only zeros or ones is so common that we have special functions to do that:

```

julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> ones(2, 3)
2×3 Array{Float64,2}:
 1.0  1.0  1.0
 1.0  1.0  1.0

```

Notice how you can optionally specify as the first argument what type you want each element to be. If you don't specify type, then it will default to `Float64`.

Creating a whole array of random numbers is also often practical. For instance in deep learning, large matrices with random values are frequently used.

```

julia> rand(UInt8, 2, 2)
2×2 Array{UInt8,2}:
 0x8e  0x61
 0xcf  0x0d

```

Sometimes you just want to fill a whole matrix with a specific value:

```

julia> fill(12, 3, 3)
3×3 Array{Int64,2}:
 12  12  12

```

```
12 12 12  
12 12 12
```

Normally you don't have to create an identity matrix as you can just use `I` which will morph into whatever shaped matrix it needs to. However if you want a concrete identity matrix you can easily make one using `zeros`:

```
julia> using LinearAlgebra
```

```
julia> zeros(3, 3) + I  
3x3 Array{Float64,2}:  
1.0 0.0 0.0  
0.0 1.0 0.0  
0.0 0.0 1.0
```

In the next chapter we will take a more geometric view of matrices and vectors.

# Moving a Rocket

- **Vectors and Geometry.** Looking at a geometric interpretation of vectors as points and displacements.
- **Affine Transformations.** Using matrices to move, scale and rotate geometric objects.
- **Simulate Movement** of a space rocket using vectors and matrices.
- **Dot product and cross product.** Understand why we have different ways of multiplying vectors and how each form can be useful in our code.

In previous chapter we looked at how vectors and matrices can be used to simulate a spreadsheet and solve *systems of linear equations*. However it is more common to think of vectors and matrices in geometric terms.

Computer games will use vectors and matrices to move objects around the screen. 2D vector drawing applications will use vectors and matrices to define drawings and allowing you to rotate, stretch and scale your drawings.

In our case we are interested in finding ways to simulate the movement of our multi-stage rocket.

## Vectors and Points

To be able to position our rocket in space, we need to discuss *points*. To express velocities and forces we need *vectors*.

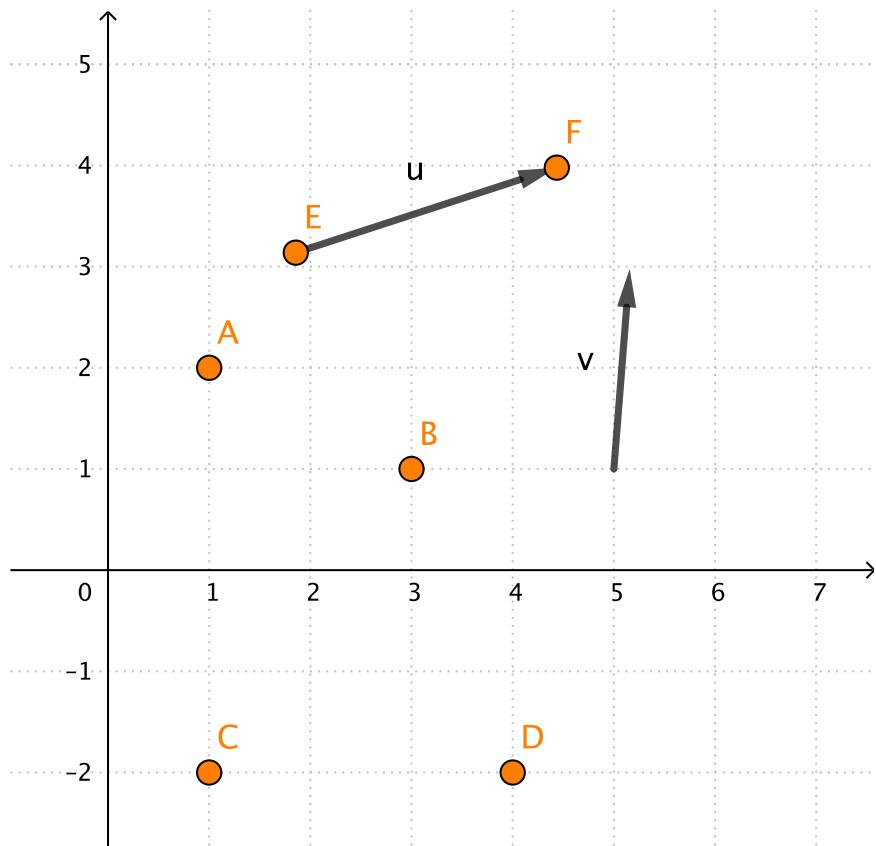
Everybody has been exposed to points in geometry class. They are dots with no size, placed at specific positions within a coordinate system. Their position is defined relative to the origin of a coordinate system.

Vectors in contrast are usually represented as arrows with a particular orientation and length. Their length is given by the `norm` function. Since they represent displacements they don't exist at any particular place in the coordinate system.

Points and vectors can interact with each other. Look at the vector  $\mathbf{u}$  in the figure above. We can think of it as the difference between the points  $F$  and  $E$ .

We can flip this argument around and assume we start with the point  $E$  and vector  $\mathbf{u}$ . Adding  $\mathbf{u}$  to the point  $E$  we can produce point  $F$ .

It is also common to think of points as simply vectors starting at our coordinate system origin as shown above. For instance point  $B$  has the coordinates  $(3, 1)$ ,

Figure 44: Points in orange and gray vector  $\mathbf{u}$  and  $\mathbf{v}$

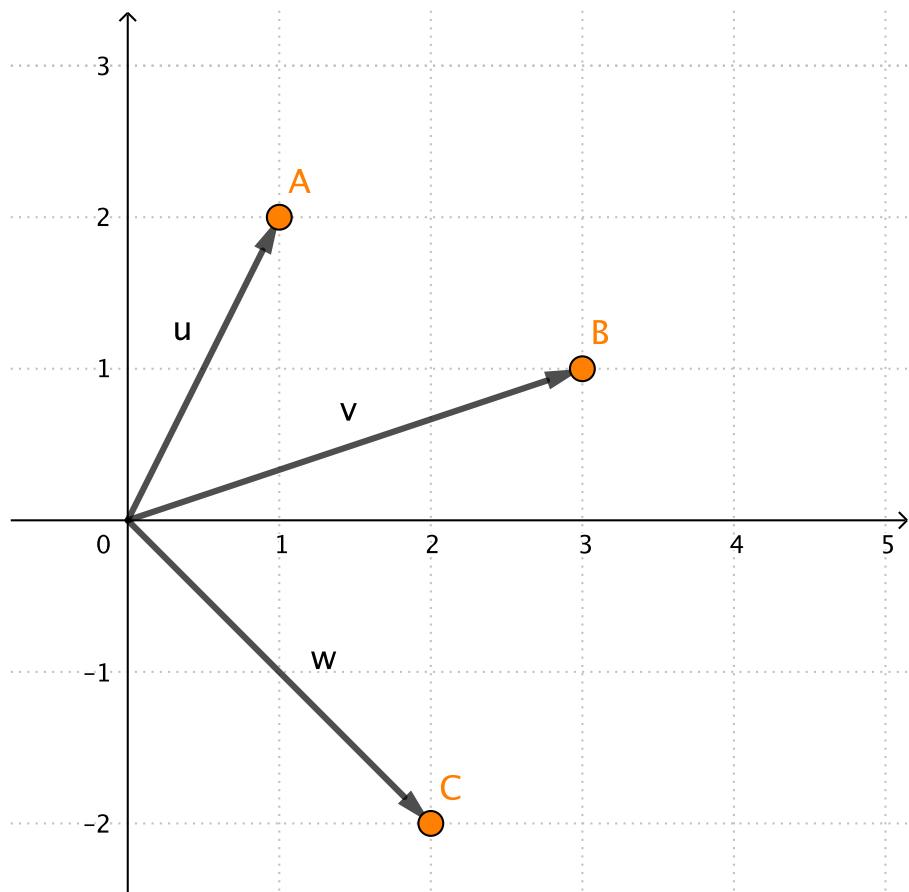


Figure 45: Using vectors as a way of defining point

while the vector  $\mathbf{v}$  is defined as a displacement  $[3, 1]$ . That is 3 length units along the x-axis and 1 length unit along the y-axis.

## Vector Operations

What can we actually do with a vector? Adding and subtracting vectors is the simplest thing to imagine.

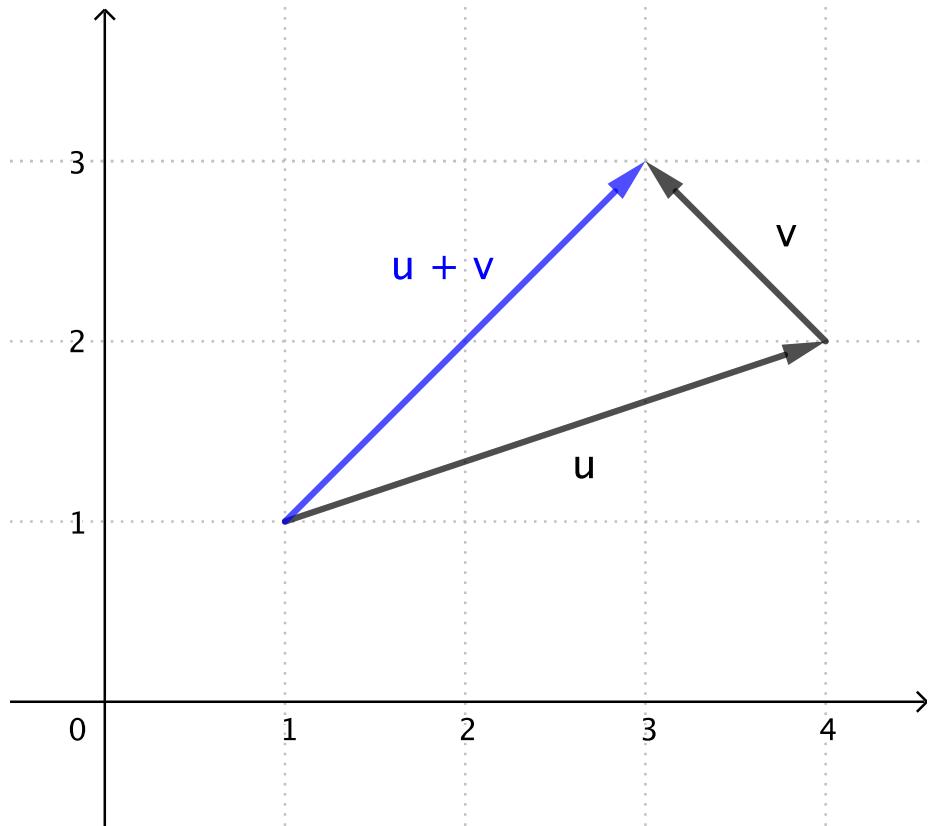


Figure 46: Vector addition

Say we have the vectors  $\mathbf{u} = [3, 1]$  and  $\mathbf{v} = [-1, 1]$  then we can visualize adding these vectors by attaching  $\mathbf{v}$  to the tip of  $\mathbf{u}$ . Then we draw a new vector from the start of  $\mathbf{u}$  to the end of  $\mathbf{v}$ . Julia can perform this task numerically:

```
julia> u = [3, 1]
2-element Array{Int64,1}:
 3
 1

julia> v = [-1, 1]
2-element Array{Int64,1}:
 -1
 1
```

```
julia> u + v
2-element Array{Int64,1}:
 2
 2
```

Notice we do not need to use element-wise operations for adding using `.+`. That is because adding vectors of the same size is a well defined operation in mathematics.

Adding a scalar to a vector however is not well defined, which means that in this case an element-wise operation is required.

```
julia> u + 1
ERROR: MethodError: no method matching +
```

```
julia> u .+ 1
2-element Array{Int64,1}:
 4
 2
```

Also notice that element-wise assignment is not the same. In the example below we make `w` point to the same vector as `u`. Later we assign `v` to `w` using element-wise assignment.

This does not make `w` point to the same vector as `v`, instead it alters the content of `w` to have the same content as `v`.

```
julia> w = u
2-element Array{Int64,1}:
 3
 1
```

```
julia> w .= v
2-element Array{Int64,1}:
 -1
 1
```

```
julia> u
2-element Array{Int64,1}:
 -1
 1
```

Hence this assignment indirectly cause `u` to change. Notice if we alter `w`, then `v` will remain unchanged.

```
julia> w[1] = 42
42

julia> v
2-element Array{Int64,1}:
 -1
 1
```

$u$  on the other hand, gets altered because it still refers to the same vector  $w$ .

```
julia> u
2-element Array{Int64,1}:
42
1
```

**Subtraction** is equally easy to perform. However in this case we need to be a bit more creative about how we think about the operation geometrically.

Below you can see  $v$  subtracted from  $u$ . This is done by adding the vector  $-v$  to  $u$ . In a sense we are still doing addition. It is just that we flip the orientation of vector in the second argument.

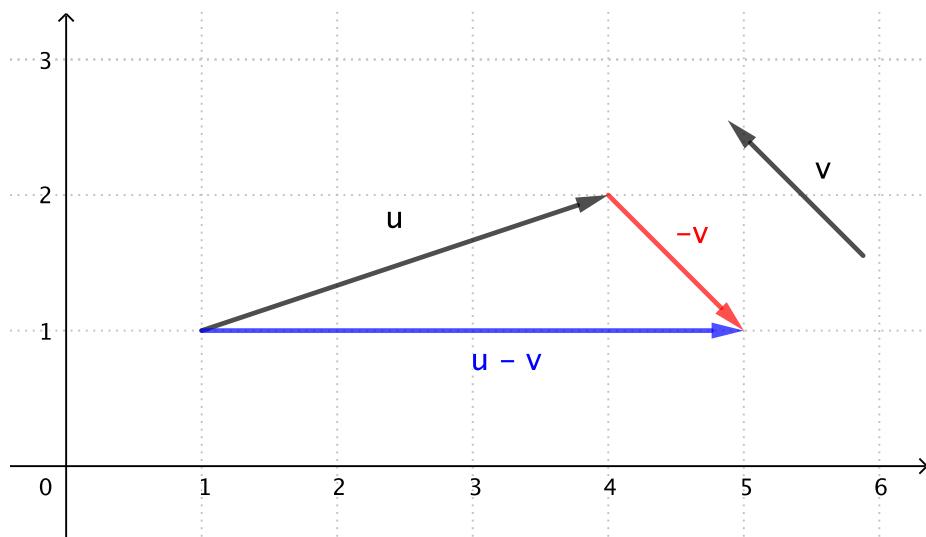


Figure 47: Vector subtraction

## Multiply Vectors with Scalars

It is not obvious what multiplication of two vectors should mean, but multiplying a vector with a scalar has an obvious meaning.

```
julia> 2*[2, 4]
2-element Array{Int64,1}:
4
8

julia> v = [3, 2]
2-element Array{Int64,1}:
3
2

julia> 3v
2-element Array{Int64,1}:
9
```

6

Geometrically this means stretching or shortening a vector. If I multiple a vector with 2, I am double its length. If I multiple with 3, I triple its length.

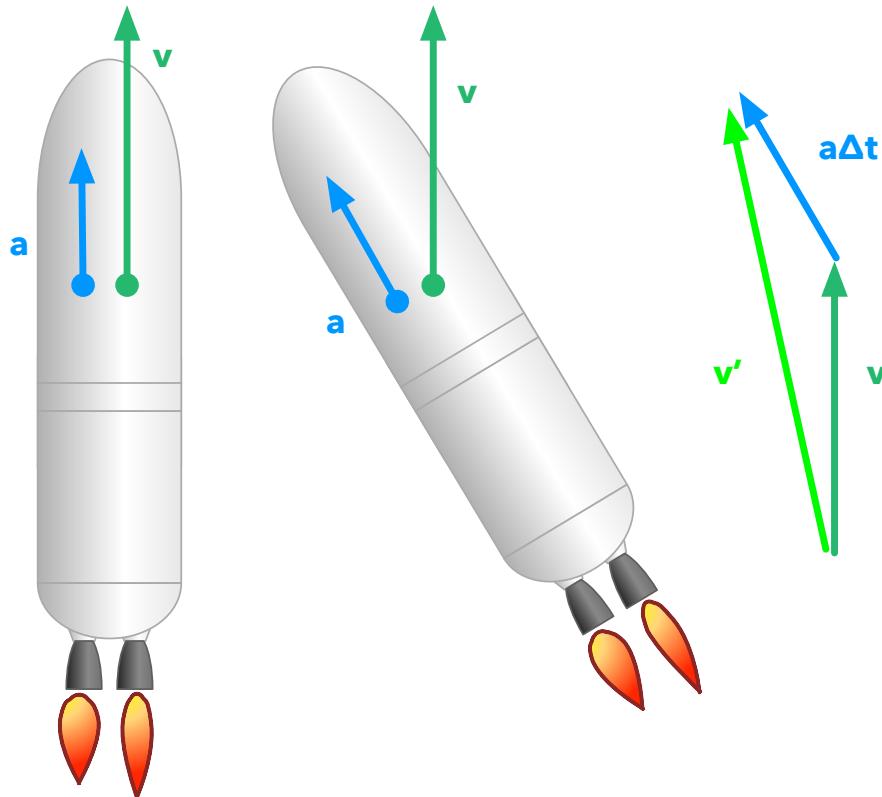
Where is this useful when simulating rocket flight?

Velocity, acceleration and forces can be expressed as vectors.

Both in simulations and computer games we update the state of the world on regular intervals with small time steps often called  $\Delta t$ . Thus if we start off with the velocity  $\mathbf{v}$  at the beginning of the time step  $\Delta t$ , then we will get a new velocity  $\mathbf{v}'$  at the end. This can be expressed as:

$$\mathbf{v}' = \mathbf{v} + \mathbf{a}\Delta t$$

Below is a visual representation of this case. The rocket is launched upright and at some point it is tilted over to reach orbit. At this point the acceleration  $\mathbf{a}$  is gradually altering the direction of the velocity vector.



We cannot simply add the acceleration vector to the velocity, since it merely represent a rate of change in velocity. To get a  $\Delta\mathbf{v}$  we need to scale the acceleration with the time step  $\Delta t$ . Here is a simple example of doing this in Julia:

```
julia> v = [0, 3];
```

```
julia> a = [-1, 2];
julia> Δt = 2;
julia> v' = v + a*Δt
2-element Array{Int64,1}:
 -2
  7
```

## Unit Vectors and Basis of Coordinate System

Together with scaling of vectors, addition and subtraction, we can express any vector as a combination of two or more vectors. For a 2D space you would need two vectors while for a 3D space you would need three vectors and so on.

This is a common way to think about vectors. In a normal cartesian<sup>24</sup> coordinate system. We can think of a 2D vector  $\mathbf{v}$  with coordinates  $x$  and  $y$  defined as:

$$\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix} = x\mathbf{e}_x + y\mathbf{e}_y = x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ y \end{bmatrix}$$

The  $\mathbf{e}_x$  and  $\mathbf{e}_y$  vectors are unit vectors defining the *standard basis* for a coordinate system. A unit vector is simply a vector of length one, or in Julia terms a vector  $\mathbf{v}$  where `norm(v) == 1`.

Here is a simple example in Julia demonstrating that we can write the vector  $[3, 4]$  this way.

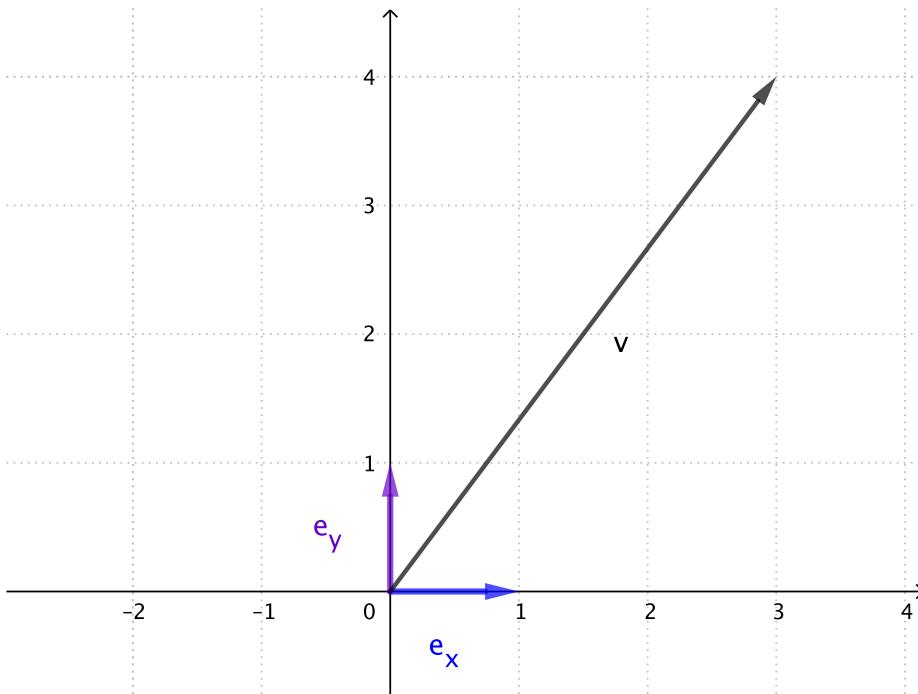
```
julia> v = 3*[1, 0] + 4*[0, 1]
2-element Array{Int64,1}:
 3
 4
```

This concept may seem overly abstract and pointless. But it is a very useful way of thinking about vectors when we will later demonstrate how a matrix can be used to rotate a vector.

Vector rotation allows us to express rotation of our rocket. Without this our rocket cannot make turns.

---

<sup>24</sup>The cartesian coordinate system is the one all school children get introduced to first. It is where points are measured along an x-axis, y-axis and potentially a z-axis. As opposed to say *polar coordinates* which uses angles.



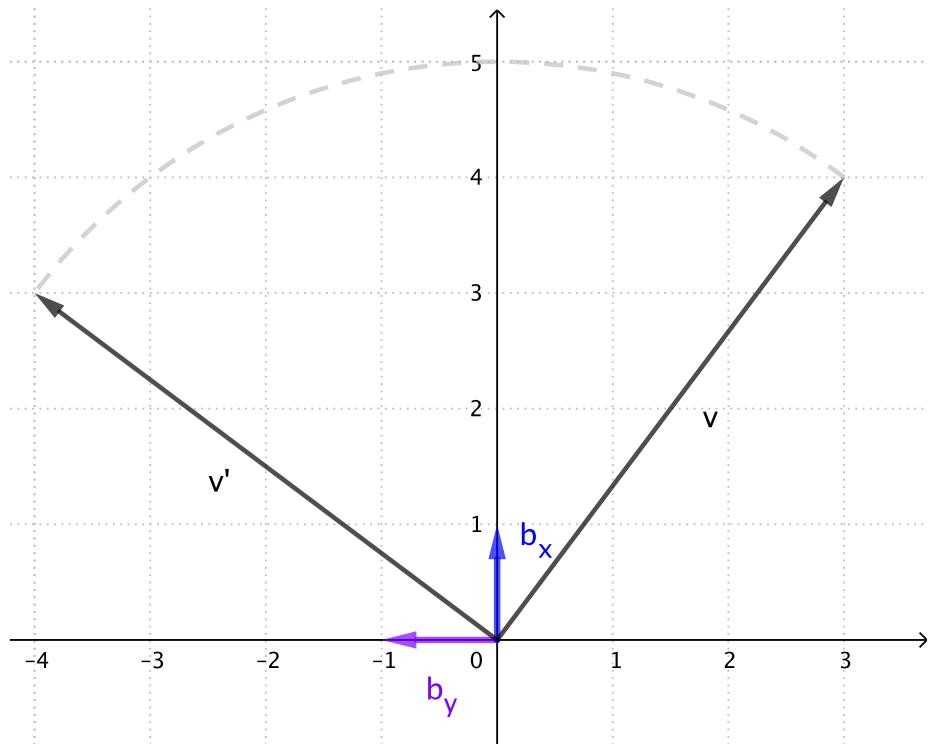
When rotating a vector we think of this as rotating its basis vectors. Say we want to rotate vector  $\mathbf{v}$   $90^\circ$ . It is not immediately obvious how you would do that. But rotating the unit vectors is simple. If I rotate  $\mathbf{e}_x$   $90^\circ$  it would point straight up. While  $\mathbf{e}_y$  rotated  $90^\circ$  will point backwards along the x-axis. I will use the names  $\mathbf{b}_x$  and  $\mathbf{b}_y$  for this new basis to not confuse them with the standard basis.

$$\mathbf{b}_x = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \mathbf{b}_y = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

This allows us to easily calculate a rotation of  $\mathbf{v}$ :

```
julia> v' = 3*[0, 1] + 4*[-1, 0]
2-element Array{Int64,1}:
 -4
  3
```

Below you can see  $\mathbf{v}$  rotated, resulting in a new vector  $\mathbf{v}'$ .



## Multiplying Vectors with Matrices

Here is the interesting part: We can combine these two unit vectors into a matrix and perform a matrix multiplication. This will give us the same result.

```
julia> v = 3*[1, 0] + 4*[0, 1]
2-element Array{Int64,1}:
 3
 4

julia> A =[ 0 1;
           -1 0];

julia> v' = A*v
2-element Array{Int64,1}:
 4
 -3
```

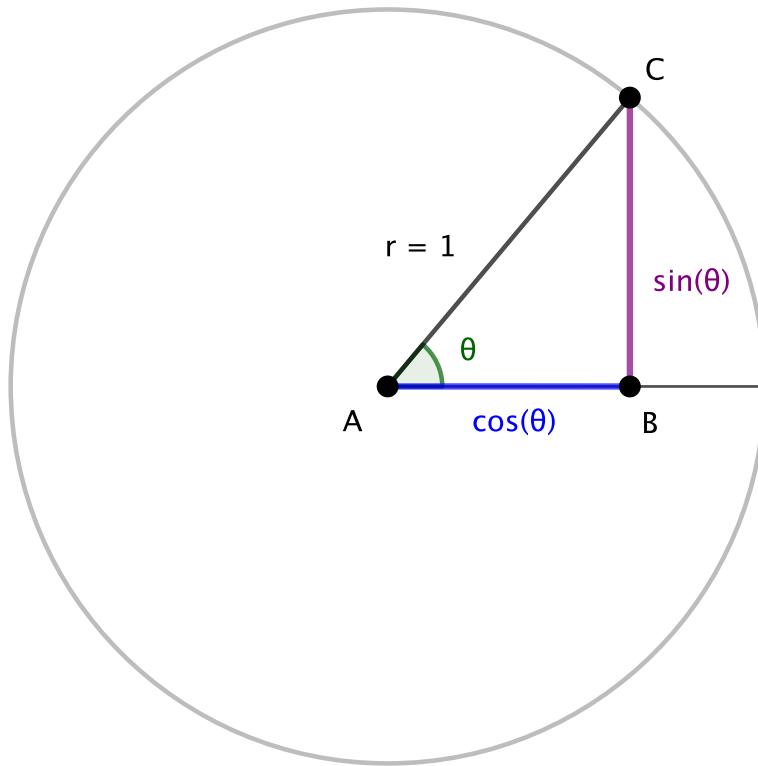
Finally we have a geometric interpretation of matrix multiplication. You can think of every column in a matrix as a basis vector for an axis in a coordinate system. When you multiply a vector with this matrix you calculate what a vector given in this coordinate system will look like in the standard cartesian coordinate system.

## Vector Rotation

Let us expand this idea to rotate a vector an arbitrary number of degrees  $\theta$ . Below you can see how cos and sin is defined using a unit circle (a circle with radius 1).

If we think of  $AB$  as a vector, then its x-component is  $\cos \theta$ , and its y-component is  $\sin \theta$ .

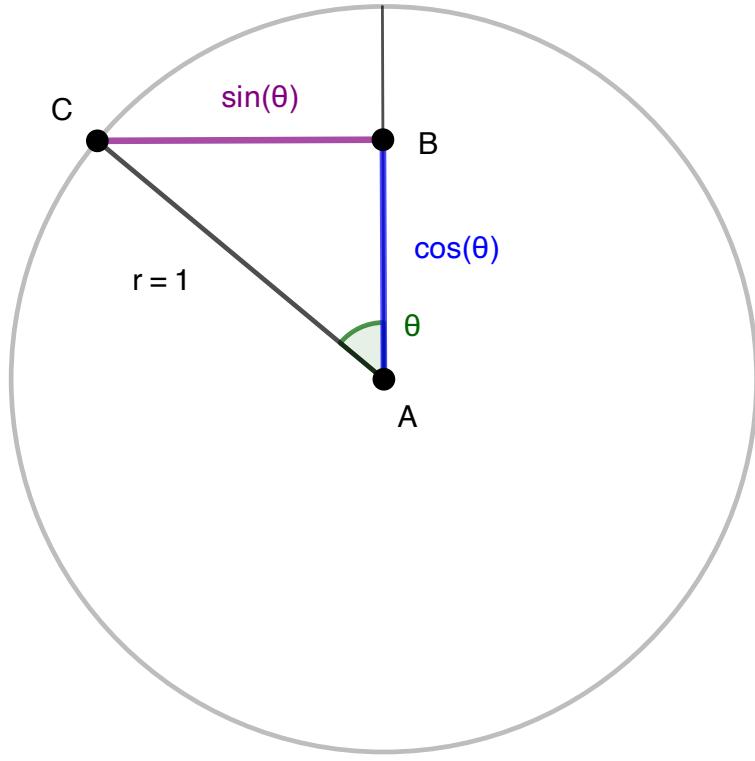
Let us imagine  $AB$  is the  $\mathbf{b}_x$  basis vector, which has been rotated  $\theta$  degrees. Then it is clear how sin and cos can be used to calculate a new basis vector.



Thus a basis vector  $\mathbf{b}_x$  rotated  $\theta$  degrees, can be expressed as:

$$\mathbf{b}_x = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

Likewise we can look at what happens to  $\mathbf{b}_y$  if we rotate it  $\theta$  degrees.



Again  $AC$  represent our rotated basis vector. We can express this new basis vector as:

$$\mathbf{b}_y = \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$$

Now we can express a vector  $[x, y]$  which has been rotated  $\theta$  degrees as:

$$x \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + y \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Here is a convenient Julia function to construct this rotation matrix for us:

```
"""
    rotation(degrees)
Create a rotation matrix for rotating a 2D vector `deg` degrees.
"""
function rotation(deg::Real)
    rad = deg2rad(deg)
    cosθ = cos(rad)
    sinθ = sin(rad)
```

```
[cosθ -sinθ;
 sinθ cosθ]
end
```

This is an example of using this function to rotate a bunch of different vectors.

```
julia> M = rotation(90)
2×2 Array{Float64,2}:
 6.12323e-17 -1.0
 1.0          6.12323e-17

julia> v = [3, 4]
2-element Array{Int64,1}:
 3
 4

julia> M*v
2-element Array{Float64,1}:
 -4.0
 3.000000000000004

julia> M*[3, 3]
2-element Array{Float64,1}:
 -3.0
 3.0

julia> M*[3, 0]
2-element Array{Float64,1}:
 1.8369701987210297e-16
 3.0

julia> M*[0, 3]
2-element Array{Float64,1}:
 -3.0
 1.8369701987210297e-16
```

## Simulating Rocket Movement with Rigid Body

In physics we have a concept called ``rigid body''. This is an imagined solid object which is completely stiff. It does not wobble or deform in any way as it moves. Or rather the degree to which this happens is so small that we can ignore it.

In computer games this is how we usually simulate moving objects as that simplifies calculations. We will utilize this concept to simulate how our rocket moves over time.

```
mutable struct RigidBody
    position::Vector{Float64}
    velocity::Vector{Float64}
```

```

force::Vector{Float64}
orientation::Float64
mass::Float64
end

```

We have to keep track of a long list of properties:

- The **position** which tells us *where* the object is currently located.
- **velocity** is a vector pointing in the direction the object is moving. The magnitude of the vector tells us how *fast* the object is moving in that direction.
- **force** is the sum of all forces working on our rigid body. There may be multiple forces, but they are all vectors which we can add up to produce a single vector.
- **orientation** is an angle in degrees specifying in which direction our rocket is pointing. The thrust of the rocket engines works in the same direction.
- **mass** will change as the rocket burns propellant and eject stages.

Let us make a constructor, because most of the time we don't want to specify every little detail. At the beginning of most simulation runs, a default value will do fine.

```

function Rigidbody(mass::Real, force::Real)
    p = [0, 0]
    v = [0, 0]
    F = [force, 0]
    θ = 0

    Rigidbody(p, v, F, θ, mass)
end

```

We make some assumptions. We assume the x-axis points straight up. While that may seem strange we are allowed to define the orientation of the coordinate system whatever way we like. It makes it easier to work with, assuming movement is along the x-axis at 0° angle.

Let us make some convenience functions. These paper over the difference between what values are properties and which ones are calculated. Remember how `step` masked the difference between a `UnitRange` and a `StepRange`.

```

force(body::Rigidbody) = body.force
mass(body::Rigidbody) = body.mass
acceleration(body::Rigidbody) = force(body) / mass(body)

```

In classical mechanics we would be calculating distance traveled with the equation:

$$r = r_0 + v_0 t + \frac{1}{2} a t^2$$

Where  $r$  is the distance traveled after time  $t$  has elapsed. The starting distance at  $t = 0$  is  $r_0$  and the velocity at  $t = 0$  is  $v_0$ . Starting at  $t = 0$  we have a constant acceleration  $a$ .

However this equation is not very useful in a simulation as orientation and acceleration can change substantially over time. Instead we add up small movements over small time increments  $\Delta t$ .

This approximates performing integration. We are going to use a fairly simple numerical approximation to an integral called the *explicit Euler method*. The integral of  $f(x)$  would mean the area underneath the graph produced by  $f(x)$ .

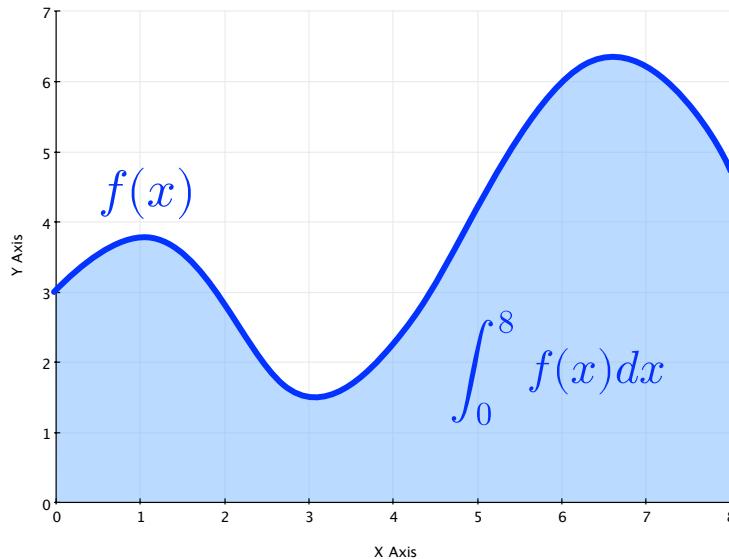


Figure 48: Light blue colored area is the integral of  $f(x)$  for the range  $(0, 1)$ .

For every time step it calculates a new position  $\mathbf{p}$  and velocity  $\mathbf{v}$ .

$$\mathbf{p}' = \mathbf{p} + \mathbf{v}\Delta t$$

$$\mathbf{v}' = \mathbf{v} + \mathbf{a}\Delta t$$

Which we can implement easily in Julia code:

```
function integrate!(body::RigidBody, Δt::Number)
    body.position += body.velocity * Δt
    body.velocity += acceleration(body) * Δt
    body
end
```

We have can test how accurate this is by writing some test code which compares this approximation with the result you would have gotten from using the analytical solution<sup>25</sup>.

---

<sup>25</sup>Analytical solution in this case means using a simple mathematical equation to calculate a result as opposed to using an iterative approach to approximate a solution.

```

using Statistics

mass = 1.0
force = 4.0
Δt = 0.1

body = RigidBody(mass, force)
ts = 0:Δt:200

approx = Float64[]
for t in ts
    integrate!(body, Δt)
    push!(approx, body.position[1])
end

acceleration = force/mass
distance(t) = 0.5*acceleration*t^2

expected = map(distance, ts)

diff(v) = abs(last(v) - first(v))
error = map(diff, zip(expected, approx))

```

In this code example we are calculating position of an object with mass 1.0 kg, with a constant force of 4.0 Newton applied to it. The units we use don't matter as long as we are consistent in how we use them.

`approx` is a list of positions at different time intervals approximated by integration. `expected` is a list of position calculated analytically using the equation:

$$r = r_0 + v_0 t + \frac{1}{2} a t^2$$

Implemented as the Julia function `distance(t) = 0.5*acceleration*t^2`, where we are treating  $r_0$  and  $v_0$  as zero.

At the end we calculate how much each expected value differs from the approximated value with the statement:

```

diff(v) = abs(last(v) - first(v))
error = map(diff, zip(expected, approx))

```

We are storing the absolute difference between the `expected` and `approx` values in `error`

In the Julia REPL we could check how big this error is on average and how much that compares to the average expected value.

```
julia> mean(error)
19.9999999968928
```

```
julia> mean(expected)
26673.33333333332
```

```
julia> mean(error)/mean(expected)
0.0007498125468516351
```

As you can see the difference between the approximation and the expected value is minuscule.

## Adding Rigid Body to Space Ship

To be useful we need the RigidBody to be used together with our space ship. E.g. as you burn fuel the mass of the space ship is reduced and this needs to be reflected in the RigidBody mass.

We add the rigid body to the SpaceVehicle structure. And we add a variable to allow us to toggle on an off gravity. Making gravity optional is common in a lot of physics simulations to make it easier to do testing.

```
mutable struct SpaceVehicle
    active_stage::Payload
    body::RigidBody
    gravity::Bool
end
```

We need to rewrite the constructors to initialize the RigidBody object properly. Replace all previous SpaceVehicles constructors with these:

```
function SpaceVehicle(rockets::Array{Rocket})
    first_stage = nopayload
    for r in reverse(rockets)
        if first_stage != nopayload
            r.payload = first_stage
        end
        first_stage = r
    end
    body = RigidBody(mass(first_stage), 0.0)
    SpaceVehicle(first_stage, body, true)
end

SpaceVehicle(rockets::Rocket...) = SpaceVehicle(collect(rockets))
```

The use of `nopayload` requires mention. We have replicated the style used for `Nothing` and `Missing` in Julia, where constants `nothing` and `missing` have been defined.

```
const nopayload = NoPayload()
```

## Simulate a Launch

Time to implement a simple function for simulating a rocket launch. This function does the following:

- *Consume propellant* for the bottom stage until the tank is empty.

- When a stage is empty, perform a *stage separation*, throwing out the bottom spent stage.
- Fire up next stage and keep going. When we reach a stage without rocket engines, such as a capsule, we end the simulation.

The point of this code is primarily to see how far up we get with multiple stages.

```
function launch!(ship::SpaceVehicle, Δt::Number; max_duration::Number = 5000)
    t = 0 # start time
    while ship.active_stage isa Rocket
        while propellant(ship) > 0 && t <= max_duration
            update!(ship, Δt)
            t += Δt
        end
        stage_separate!(ship)

        # Current stage cannot get us higher
        if thrust(ship) <= 0
            return ship
        end
    end
    ship
end
```

Some parts need further explanation. `max_duration` does not strictly need to be included, but this type of variables are often practical to add, because one can end up with infinite loops by accident. With `max_duration` you set an upper time limit for the duration of the simulation.

The time units used are irrelevant, but you have to be consistent. In my code I typically assume SI units such as meters, kilograms and seconds.

`update!(ship, Δt)` has not been implemented previously. It is what calls the integration function.

```
function update!(ship::SpaceVehicle, Δt::Number)
    stage = ship.active_stage
    body = ship.body

    update!(stage, Δt)

    body.mass = mass(ship)
    body.force = rotation(body.orientation) * [thrust(ship), 0.0]
    if ship.gravity
        body.force += gravity_force(body)
    end

    integrate!(body, Δt)
end
```

This is where the interesting bits happen which actually utilizes our rotation matrix. Notice how we update values for mass and force for the rocket based on how much fuel is spent and the thrust of the rocket engines currently firing.

This information is transferred to the `RigidBody` object.

**NOTE Separation of Concern**

It may be tempting to keep a reference to the space ship inside the `RigidBody` object to avoid copying this data. Resist that temptation! Such solutions end up easily creating a ``spaghetti'' mess of relationships. It makes it hard to build and test lower level objects such as `RigidBody` in isolation from higher level objects such as `SpaceShip`.

We multiply a vector representing the force produced by the rocket engine with a rotation matrix. This matrix is setup to rotate the force to the same direction as the one the ship is pointing.

```
body.force = rotation(body.orientation) * [thrust(ship), 0.0]
```

However we also need to check if gravity is toggled on and potentially add the force of gravity to `body.force`, which represent the sum of all forces applied to our rigid body.

```
body.force += gravity_force(body)
```

We calculate the gravitational forces based on the properties of the rigid body such as its position and mass.

```
function gravity_force(body::RigidBody)
    _, elevation = body.position
    [-earth_gravity(elevation) * body.mass, 0]
end
```

`earth_gravity(elevation)` gives the acceleration of gravity for a given elevation above the earth's surface. At the ground it is equal to  $g_0 = 9.80665$ . However as we get further into space it will get smaller.

Here is how we figure out how to implement it. We know that the force of gravity  $F_g$  on an object with mass  $m$  is proportional to the acceleration,  $g$ .

$$F_g = mg$$

Gravity causes a force between two objects in space of mass  $M$  and  $m$  inversely proportional to the square of the distance  $r$  between them:

$$F_g = G \frac{Mm}{r^2}$$

We can combine these two equations to get an expression for  $g$  dependent on the mass  $M$  of earth and the distance from earth's core,  $r$ , to the the object attracted by earth.

$$mg = G \frac{Mm}{r^2}$$

$$g = G \frac{M}{r^2}$$

In code this becomes:

```
const earth_radius = 6.38e6
const earth_mass   = 5.98e24
const gravitation_constant = 6.673e-11

"""
    earth_gravity(d)
The acceleration of gravity distance `d`
from earth surface. Should be `g₀` when `d == 0`.
"""
function earth_gravity(d::Real)
    (gravitation_constant * earth_mass)/(earth_radius + d)^2
end
```

## Updating Propellant Tanks

The last detail which we have actually covered earlier is updating the mass of the tanks to reflect spent propellant.

```
function update!(r::Rocket, Δt::Number)
    mflow = mass_flow(thrust(r.engine), Isp(r.engine))
    r.tank.propellant -= min(mflow * Δt, r.tank.propellant)
end
```

Basically the mass flow is proportional to the efficiency of the rocket engine and how much we are ``stepping on the gas pedal'' (thrust).

## Mass, Propellant and Thrust Properties

The `update!(ship, Δt)` method relies on being able to access the thrust, propellant and the mass of the ship. However we have not defined those functions before. This ought to be trivial. You can try to write them yourself or look at the full definition here.

The various `thrust` functions are layered on top of each other:

```
thrust(engine::SingleEngine) = engine.thrust
thrust(cluster::EngineCluster) = thrust(cluster.engine) * cluster.count
thrust(ship::SpaceVehicle) = thrust(ship.active_stage)
thrust(payload::Payload) = 0.0
thrust(r::Rocket) = thrust(r.engine)
```

The same goes for `mass`. We get mass for higher level objects by getting the mass of lower level objects:

```
mass(tank::Tank) = tank.dry_mass + tank.propellant
mass(payload::NoPayload) = 0.0
mass(probe::SpaceProbe) = probe.mass
```

```

mass(capsule::Capsule) = capsule.mass
mass(satellite::Satellite) = satellite.mass

function mass(r::Rocket)
    mass(r.payload) + mass(r.tank) + mass(r.engine)
end

mass(ship::SpaceVehicle) = mass(ship.active_stage)

```

The `launch!` function relies on knowing how much propellant is left to decide whether stage separation needs to be performed.

```

propellant(non_rocket) = 0.0
propellant(r::Rocket)      = r.tank.propellant
propellant(r::SpaceVehicle) = propellant(r.active_stage)

```

Notice how the default implementation says that an object has no propellant at all.

## Practical Example of Launching a Rocket

Evaluate this code in the REPL or put it in a file. Load code in file either by writing `julia -i filename.jl` at the shell prompt or writing `include("filename.jl")` inside the Julia REPL.

This is a simple simulation of a Falcon 9 rocket launch. Its tank and engine specifications are very similar.

```

merlin = Engine(845e3, 282, 470)
cluster = EngineCluster(merlin, 9)

firststage = Rocket(Tank(23.1e3, 418.8e3), cluster);
secondstage = Rocket(SpaceProbe(22e3),
                     Tank(4e3, 111.5e3),
                     Engine(934e3, 348, 470));

ship = SpaceVehicle(firststage, secondstage);

```

We set a time increment for our simulation. Smaller increments give more accurate simulation but will consume more processing power.

$\Delta t = 0.1$

We can then simulate a launch in the REPL by writing:

```
julia> launch!(ship, Δt)
```

It is useful to have a quick way of accessing the velocity and position of the ship rather than having to extract the rigid body every time.

```

pos(ship::SpaceVehicle) = ship.body.position
velocity(ship::SpaceVehicle) = ship.body.velocity

```

We can then check the current position and velocity of the ship that was launched.

```
julia> velocity(ship)
2-element Array{Float64,1}:
 3703.572857010847
 0.0

julia> pos(ship)
2-element Array{Float64,1}:
 937346.6102144092
 0.0
```

What this is telling us, is that the ship went up 937 kilometers and gained a velocity of 3.7 kilometer per second which is roughly 13 330 km/hour.

## Vector Dot Product and Cross Product

So far we have looked at adding and subtracting vectors as well as multiplying them with a scalar or with a matrix. However we have not actually looked at multiplying vectors with each other.

You can fake it by multiplying a row vector with a column vector. However this is really just matrix multiplication for matrices which are single rows or columns.

But how about multiplying two row vectors or two column vectors? Is there a sensible definition of how that should behave, and which has a sensible geometric interpretation?

In this case the math and physics guys came up with two different ways of doing it:

- Dot product
- Cross product

We will look at the dot product first, because it is perhaps most intuitive and has some resemblance to matrix multiplication, which we are already familiar with.

### Dot Product

The dot product of two vectors **a** and **b** is written as  $\mathbf{a} \cdot \mathbf{b}$  and is the same as the matrix multiplication  $\mathbf{a}^T \mathbf{b}$ . Here the  $T$  means the transpose (swapping rows and columns).

Assuming we have these two column vectors in Julia:

```
julia> a = [2, 3]
2-element Array{Int64,1}:
 2
 3

julia> b = [3, 1]
2-element Array{Int64,1}:
 3
 1
```

Then the following expressions would all be equivalent:

```
julia> transpose(a)*b
9

julia> dot(a, b)
9

julia> a · b
9
```

The dot  $\cdot$  is written \cdot in the Julia REPL.

The basic idea of the dot product is that it should represent *how much* two vectors are pointing in the same direction. The rational is that if the vectors make a  $90^\circ$  with each other, then the product should be zero because they are not pointing in the same direction at all. While if the angle between them is  $0^\circ$  it should be like multiplying the lengths (magnitude) of both vectors.

This has a practical utility in many physics equations. For instance if you take equations defined for 1-dimensional space and try to make them 2-dimensional or 3-dimensional, then regular multiplication will often need to be replaced with dot products for the equations to still work.

If I move an object of mass  $m$  from standstill, a distance  $r$  using a force  $F$ , then it will acquire some velocity  $v$ . The relationship between all these variables is expressed as:

$$W = Fr$$

$$E_k = \frac{1}{2}mv^2$$

$$Fr = \frac{1}{2}mv^2$$

#### **NOTE Work and Kinetic Energy**

What these equations express is that the work  $W$  done on an object will equal its kinetic energy  $E_k$ . Kinetic energy is an expression of the energy stored in an object, based on its movement and mass. Work is just what we call the energy we have added to an object. Work could increase an object's velocity, temperature, potential energy etc.

It is more practical to express these equations with vectors. An object moves with a velocity  $\mathbf{v}$  in a particular direction. Force  $\mathbf{F}$  is also applied to the object from a particular direction.

We can then write the equation as:

$$\mathbf{F} \cdot \mathbf{r} = \frac{1}{2} m \mathbf{v}^2$$

Notice that the mass  $m$  will naturally remain a scalar. It makes no sense to speak of the direction of mass.

We want to know how much the force  $\mathbf{F}$  works in the same direction as the displacement  $\mathbf{r}$  of the object. That only depends on the direction of  $\mathbf{r}$ .

Remember when working with basis, we used unit vectors to represent axis or directions. Let us define a unit vector  $\mathbf{u}$  pointing in the same direction as  $\mathbf{r}$ :

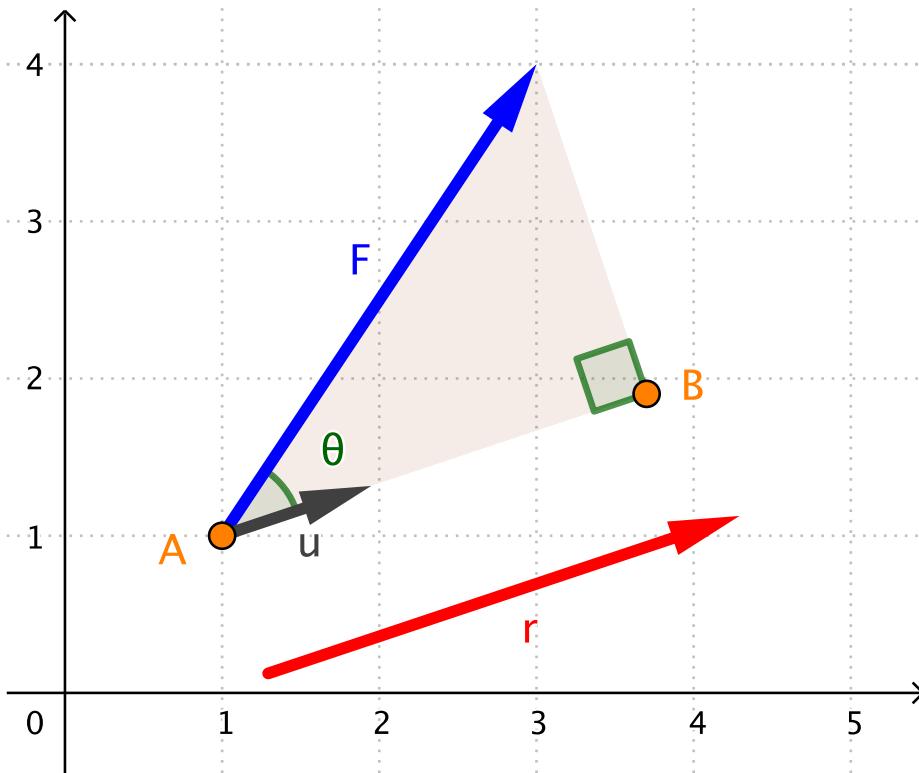
$$\mathbf{u} = \frac{\mathbf{r}}{|\mathbf{r}|}$$

We can define and calculate these vectors in Julia:

```
julia> r = [3, 1]
2-element Array{Int64,1}:
 3
 1

julia> u = r/norm(r)
2-element Array{Float64,1}:
 0.9486832980505138
 0.31622776601683794
```

To find out how much force is used in the same direction as  $\mathbf{r}$  we perform the dot product  $\mathbf{F} \cdot \mathbf{u}$ . If you look at the illustration below you can get a better idea of how this works.



The product  $\mathbf{F} \cdot \mathbf{u}$  gives us the length of the line segment  $AB$ . If you think of  $\mathbf{u}$  as a basis vector defining a coordinate axis, then this dot product is telling us the x-axis component in this coordinate system.

Another popular way to look at this is to think of  $\mathbf{F}$  projected onto the line going through the points  $A$  and  $B$ . What do we mean by projected? If you shone light onto  $\mathbf{F}$  towards the  $\mathbf{u}$  vector then a shadow would be cast spanning the line segment  $AB$ .

If we make this calculation in Julia you can observe that the result looks similar to what you could eyeball as the length of line segment  $AB$ .

```
julia> F = [2, 3]
2-element Array{Int64,1}:
 2
 3

julia> dot(F, u)
2.846049894151541
```

This demonstrates that these two equations are equal:

$$\mathbf{F} \cdot \mathbf{u} = |\mathbf{F}| \cos \theta$$

Let us substitute our original definition of  $\mathbf{u}$  and rework the equation a bit. From this you will see the geometric interpretation of the dot product comes naturally:

$$\mathbf{F} \cdot \frac{\mathbf{r}}{|\mathbf{r}|} = |\mathbf{F}| \cos \theta$$

$$\mathbf{F} \cdot \mathbf{r} = |\mathbf{F}| |\mathbf{r}| \cos \theta$$

Using this definition, do you see how everything works like normal multiplication as long as  $\theta = 0^\circ$ ? We can verify this relation in Julia:

```
julia> theta = acos(F*u/norm(F))
0.661043168850687
```

```
julia> norm(F)*norm(r)*cos(theta)
8.999999999999998
```

```
julia> F*r
9
```

## Cross Product

The cross product of two vectors  $\mathbf{v}$  and  $\mathbf{u}$  is defined as a vector  $\mathbf{w}$ , with magnitude equal to the area of the parallelogram spanned by the vectors  $\mathbf{v}$  and  $\mathbf{u}$ . This is illustrated below.

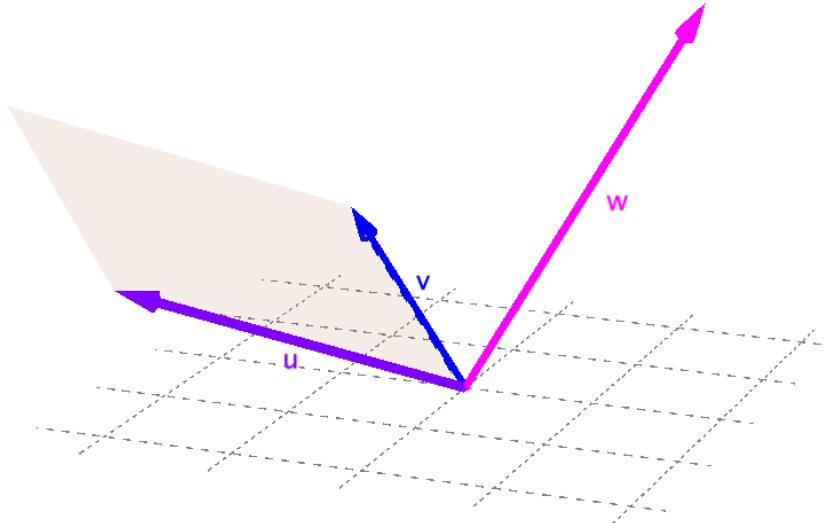
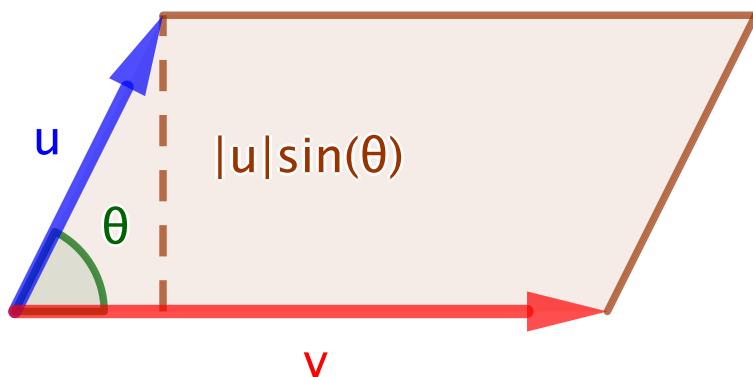


Figure 49: Vector  $\mathbf{w}$  is the cross product of vectors  $\mathbf{u}$  and  $\mathbf{v}$

The area of the parallelogram can be calculated with:

$$|\mathbf{u}||\mathbf{v}| \sin \theta$$

The figure below explains why this would calculate the area of the parallelogram. Notice that  $|\mathbf{v}| \sin \theta$  gives the height of the parallelogram. This allows us to calculate the area of the same parallelogram.



Observe how the cross product is like an opposite of the dot product. The magnitude of the resulting vector  $\mathbf{w}$  becomes smaller when  $\theta$  is small, while it gets its max value when  $\theta = 90^\circ$ . The dot product in contrast will have the value zero in this case.

Another seemingly peculiar difference is that the cross product produces as vector rather than a scalar value.

### Why is the cross product a vector?

The reason for this, is that some physics equations, where we multiply scalars, when we want to use vectors instead, get their max values when the vectors are perpendicular rather than parallel. In particular this happens when dealing with equations for things which rotate.

It also explains the fact that we get a vector as result rather than a scalar. It is not enough to know simply how much a circular movement accelerates or moves. We also need to know the rotation axis. A rotation axis can be represented by vector.

A vector is very versatile. You can use it to represent a velocity in a particular direction. Then the vector points in the direction of movement, and its magnitude represents how fast the object is moving in that direction.

However there is an alternative way to think of a vector. The *orientation* of the vector could represent a **rotation axis**, and the magnitude how fast an object rotates around this axis. Angular velocity  $\vec{\omega}$  can thus be represented as a vector, where the magnitude indicate the velocity and the orientation of the  $\vec{\omega}$  vector is the rotation axis.

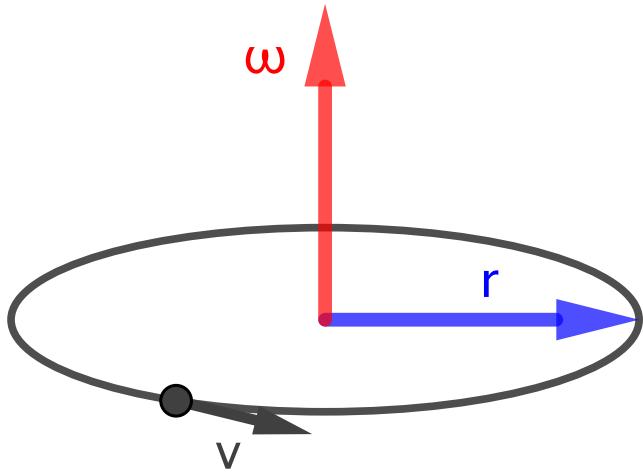


Figure 50: An object rotating around and axis  $\omega$  at distance  $r$  and velocity  $v$

We can go further. A vector can represent an *acceleration* in a particular direction, or it can represent a *force* applied in a particular direction.

A vector could likewise represent *angular acceleration* around a rotating axis. Just like we can have angular velocity and angular acceleration we can have a sort of ``angular force'', called torque.

When you are turning a bolt with a wrench, you are applying **torque**. If you push the edges of a wheel to make it spin around, then the magnitude of the torque vector says how much you are altering the angular acceleration. The orientation of the torque vector says around which rotation axis the angular acceleration is changing. We can express this as:

$$\mathbf{T} = \mathbf{r} \times \mathbf{F}$$

$$\mathbf{T} = \mathbf{I}\vec{\alpha}$$

Where  $\mathbf{T}$  is the torque vector,  $\mathbf{r}$  is a vector representing distance from the rotation axis to the force  $\mathbf{F}$  being applied and causing an angular acceleration  $\vec{\alpha}$ . The mass of the object, the distribution of that mass and the axis you try to rotate around cause a resistance to movement.

This resistance is called rotational inertia  $\mathbf{I}$ . It is analogous to mass  $m$  for linear motion, just like angular acceleration  $\vec{\alpha}$  is analogous to linear acceleration  $\mathbf{a}$ .

$$\mathbf{F} = m\mathbf{a}$$

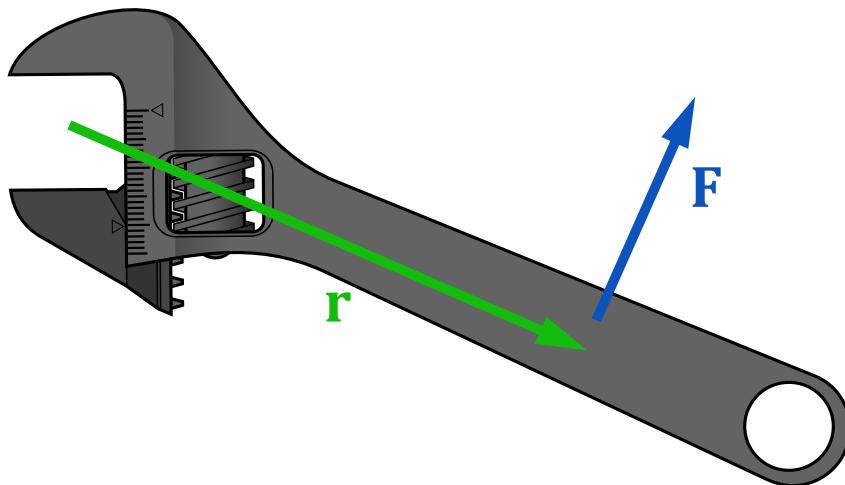


Figure 51: An adjustable wrench where a force  $\mathbf{F}$  is being applied at distance  $\mathbf{r}$  along the shaft.

We know that a heavy object will not accelerate as fast when pushed as lighter object will.

### Calculating Cross Product in Julia

We can calculate the cross product in Julia using the `cross` function or the  `$\times$`  operator. You write  `$\times$`  by typing `\times` and hitting tab.

```
julia> u = [6, 0, 0]
3-element Array{Int64,1}:
 6
 0
 0

julia> v = [3, 4, 0]
3-element Array{Int64,1}:
 3
 4
 0

julia> w = cross(u, v)
3-element Array{Int64,1}:
 0
 0
24

julia> w = u × v
3-element Array{Int64,1}:
 0
```

0  
24

Unlike the dot product, cross products only work for 3D vectors. You cannot use 2D vectors, 4D vectors or anything else.

### How is the cross product useful to us?

If we are to expand our simulation of a rocket launch, then we would want to do *more* than simply going straight up. We would also like our rocket to be able to make turns.

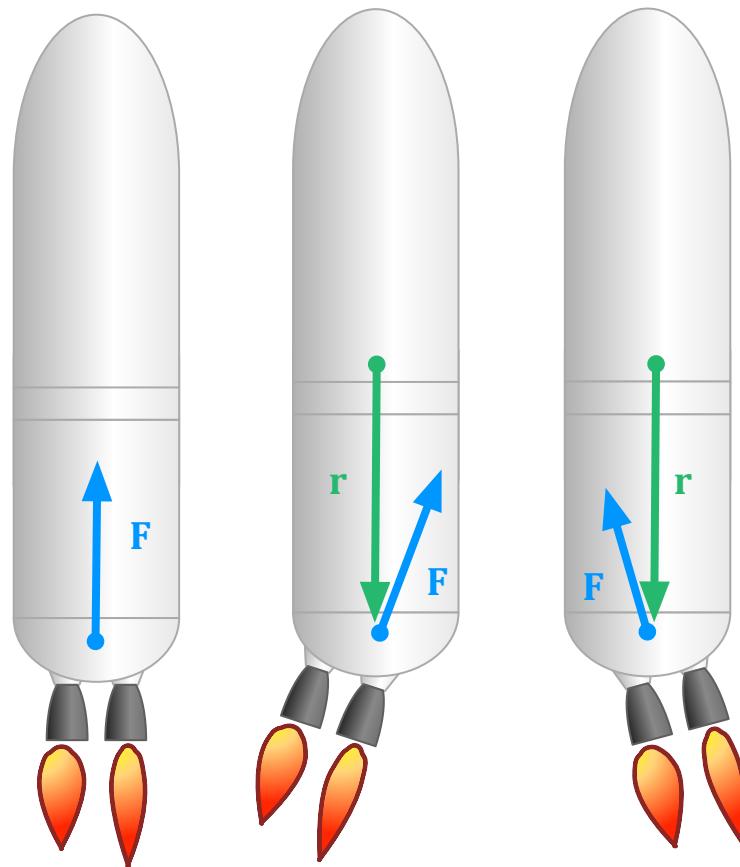


Figure 52: Gimbaled thrust on a rocket. Rocket engine can be swiveled to create a torque, causing rotation.

On real rockets, the rocket engine can be swiveled from side to side, to alter the direction of the force pushing the rocket. This allows us to induce a torque on the rocket causing it to rotate.

Calculating what happens will require performing cross product calculations.

## Summary of Vectors and Matrices

We covered a lot of different things about vectors and matrices. It can be easy to loose the thread since we took a number of detours to explain different situations where various vector and matrix operations are useful. I also tried to give you some motivations for why they work the way they do.

```
M*v
transpose(u)*v
dot(u, v)
cross(u, v)
```

Initially we looked at how matrices can be used as a form of mini spreadsheets, where we use them to perform multiple operations on input data.

Then we looked at the connection between vectors and their geometrical representation. **Column vectors** are seen as points or arrows. That is something with geometric interpretation, while **row vectors** are seen as operations on column vectors.

Hence a matrix can be seen as either a stack of operations to perform on some input data, or as multiple columns of input data.

```
output = [op1; op2; op3] * [inp1 inp2 inp3 inp4]
```

This was explored further in this chapter to demonstrate how a matrix could be used to rotate a vector. Since points can be viewed as vectors starting at the origin of the coordinate system, these matrices allow us to rotate, scale or translate an arbitrary geometric figure made up of multiple points. In our example however we used it to rotate the thrust vector of a space rocket.

Vector rotation was explained by looking at how the matrices manipulate vectors defining the **basis** of a coordinate system.

Finally we looked at the many ways in which we can think of multiplying vectors:

1. With a scalar to stretch or shrink them.
2. As a matrix multiplication between a row vector and column vector.
3. Dot product between two vectors.
4. Cross product between two vectors.

Finally we covered the rational for why there are different ways of multiplying vectors. When calculating work  $\mathbf{W} = \mathbf{F} \cdot \mathbf{r}$  we use the dot products while for calculating torque we use the cross product  $\mathbf{T} = \mathbf{r} \times \mathbf{F}$ .

You will most likely find yourself use the dot product more. Matrix multiplications are already similar to performing a series of dot products.



# Functional Programming

- **Different types of arguments** to functions, such as variable number of arguments, named arguments and default arguments.
- **Higher Order Functions**. Functions taking functions as arguments and returning them. How do you define them and use them.
- **Function Composition**. Combining functions to create new functions.
- **Function Call Chaining**. Arrange function calls in a pipeline.
- **Partial Application**. Creating new functions by only providing some of the arguments. Why is this useful and when do you use it?

While Julia was designed to support many different types of programming paradigms it does have strong support for functional programming. Compared to mainstream programming languages such as Java, C#, C++, Python and JavaScript, Julia code tends to follow a functional style.

What functional programming *is* can be debate at length. One problem is that there is no a single clear definition.

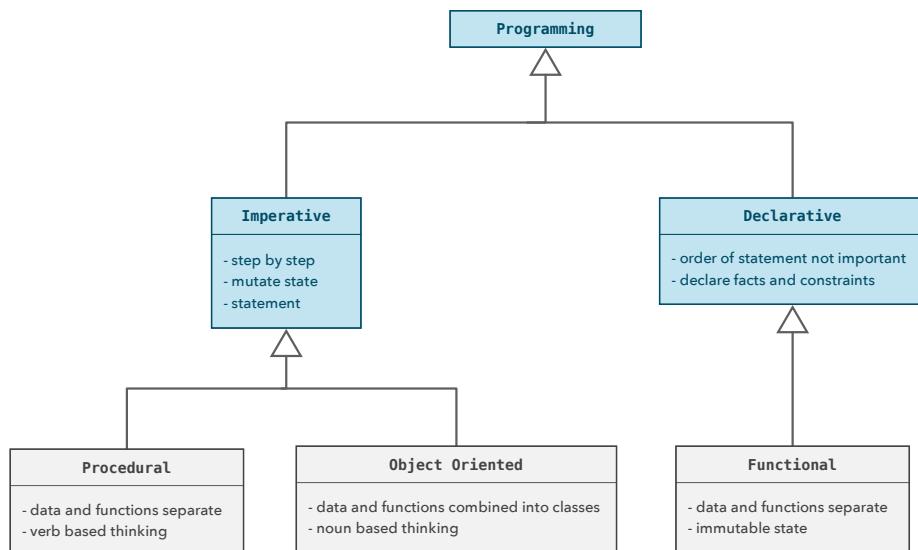


Figure 53: Different programming paradigms

The diagram above is a simplified and not entirely accurate illustration of how the different programming paradigms relate. The most important thing I want

to point out is that procedural programming and function programming is *not* the same thing. Writing code with a bunch of functions rather than using an object-oriented approach does not make your code functional. That has been done for a long time in languages such as C, Fortran and Pascal, and it called procedural programming.

Instead functional programming usually involves a variety of different practices and approaches:

- Handling functions as *first class objects*. Meaning we can pass around functions and store them as if they were regular data.
- *Higher order functions*. Functions taking functions as arguments. Typical examples of this are functions such as `map`, `filter` and `reduce`. Use of these kinds of functions rather than for-loops is common practice developed within the functional programming community.
- *Closures or lambdas*, meaning functions which can capture state of their environment, allowing functions to serve many of the same purposes as objects in object-oriented languages.

Functional programming offers a lot of ways to combine functions in different way and modularize your code at a function level. However before getting into the details of functional programming, we will cover important details about the myriad of ways you can pass arguments to functions.

We will continue to develop our rocket example and look at ways of dealing with cases where you have numerous arguments, which are hard to remember the position of.

## Named Function Arguments

Most function calls you have seen in our example code thus far have used *positional* arguments. Positional means the position of the argument in the function definition matters. For instance if a function is defined as `f(x::Int, s::AbstractString)`, then you *cannot* call this function using `f("five", 4)` and expect Julia to figure out that the text string "five" should be assigned to the variable `s`.

Since `s` is at the second position in the argument list, it also has to be the second argument provided when using the function. Position matters.

An alternative way of passing arguments is through the use of *named arguments*. In Julia these are called **Keyword Arguments**. Below you can see an example of what creating a `SpaceVehicle` instance would look like if we used almost exclusively keyword arguments, rather than positional arguments.

```
SpaceVehicle(
    Rocket(
        tank    = Tank(
            dry_mass = 4e3,
            total_mass = 111.5e3
        ),
        engine = Engine(
```

```

        thrust = 845e3,
        Isp     = 282,
        mass    = 470
    )
),
Rocket(
    payload = SpaceProbe(22e3),
    tank    = Tank(
        dry_mass = 23.1e3,
        total_mass = 418.8e3
    ),
    engine  = Engine(
        thrust = 31e3,
        Isp    = 311,
        mass   = 52
    )
)
)
```

Why is this useful? You may have seen similar looking arrangement when looking at JavaScript code, JSON or HTML. Naming the arguments makes it easier to handle complex expressions with lots of arguments.

Let us look at adding functions, which allows us to construct a space vehicle in the manner shown above. In Julia keyed arguments are separated from positional arguments with a semicolon like this:

```
repeat(word::AbstractString; times::Integer) = join(fill(word, times), ' ')
```

Using repeat with one positional and one keyword argument:

```
julia> repeat("hello", times = 3)
"hello hello hello"
```

```
julia> repeat("hi", times = 5)
"hi hi hi hi hi"
```

Restart the REPL and define repeat with only keyword arguments.

```
repeat(;word::AbstractString, times::Integer) = join(fill(word, times), ' ')
```

Calling repeat with one positional argument will fail.

```
julia> repeat("hi", times = 5)
ERROR: MethodError: no method matching repeat(::String; times=5)
Closest candidates are:
  repeat(; word, times)
```

Both arguments must be supplied as keyword arguments.

```
julia> repeat(word = "hi", times = 5)
"hi hi hi hi hi"
```

Notice that order of arguments is irrelevant:

```
julia> repeat(times = 5, word = "hi")
```

```
"hi hi hi hi hi"
```

As you can see in this simple demonstration, the placement of the semicolon determines which arguments are positional and which ones are named. Putting semicolon in front of all arguments, make all of them named.

This is all the added code we need to support creating a space vehicle using just named arguments. The code below also introduces a number of other features we will cover more in detail:

```
function Tank(;dry_mass::Number, total_mass::Number)
    Tank(dry_mass, total_mass)
end

Rocket(;payload::Payload = nopayload, tank::Tank, engine::Engine) = Rocket(payload, ta

SpaceVehicle(rockets...) = SpaceVehicle(collect(rockets))

function Engine(;thrust::Number, Isp::Number,
                mass::Number=0, count::Integer=1)
    engine = Engine(thrust, Isp, mass)
    if count > 1
        EngineCluster(engine, count)
    elseif count < 1
        msg = "number of rocket engines must be > 0"
        throw(DomainError(count, msg))
    else
        engine
    end
end
```

Do you see that most function definitions are unchanged, except for an added semicolon? Also note we could have completely changed the order of arguments. It would not have mattered, since they are not positional anymore but uniquely identified by their name.

## Optional Arguments

The new `Tank` construction method should be straightforward, however `Rocket` has the named argument `payload::Payload = nopayload`. This means that if you don't provide a value to the `payload` argument when invoking the `Rocket` constructor function, it will automatically get the value `nopayload`. We call this the *default* value.

You can in principle run any function or use any variable to assign this default value, even the value of preceding arguments. Remember from the Static vs Dynamic Typing chapter, that functions are created at runtime.

Default values means we make the arguments optional for the user. This does not apply to named arguments alone but to positional arguments as well. Previously we defined the constructor using positional arguments this way:

```
function Rocket(tank::Tank, engine::Engine)
```

```
Rocket(nopayload, tank, engine)
end
```

We could use optional arguments, but it is often harder to achieve with positional arguments, because you have to make sure they are placed **after** required arguments. It would **not** be legal to define the function as:

```
Rocket(payload::Payload = nopayload, tank::Tank, engine::Engine)
```

Instead you would have to rearrange the arguments:

```
function Rocket(tank::Tank, engine::Engine, payload::Payload = nopayload)
    Rocket(payload, tank, engine)
end
```

In this case that would have been pointless. There is nothing to gain, because Julia is already offering a constructor by default taking all the arguments. We only need to provide a variant taking fewer arguments.

## Variable Number of Arguments

You may remember we briefly covered variable number of function arguments, using three dots .... These dots can also be used as the splat operator to turn a tuple or array into arguments.

```
SpaceVehicle(rockets...) = SpaceVehicle(collect(rockets))
```

This constructor allows us to write `SpaceVehicle(stage1, stage2, stage3)` rather than having to provide an array `SpaceVehicle([stage1, stage2, stage3])`. We use `collect` because when using variable number of arguments (varargs), we get a tuple and not an array.

Whenever you are unsure about specifics, such as this, in the Julia language, it is often very quick to do your own test. Here is an example of how to do that:

```
julia> foo(items...) = items
foo (generic function with 1 method)

julia> foo(4, 5, 7)
(4, 5, 7)
```

```
julia> typeof(ans)
Tuple{Int64, Int64, Int64}
```

This test code demonstrates that variable number of arguments are received as tuple.

## Checking Invariants

In the `Engine` constructor:

```
function Engine(;thrust::Number, Isp::Number,
               mass::Number=0, count::Integer=1)
    engine = Engine(thrust, Isp, mass)
    if count > 1
```

```

        EngineCluster(engine, count)
elseif count < 1
    msg = "number of rocket engines must be > 0"
    throw(DomainError(count, msg))
else
    engine
end
end

```

We are using optional arguments again with default values twice. What is new here is *invariant* checking and error handling. Most of our example code does not have error handling for the purpose of clarity and focusing on the topic being taught. However that does not mean that *your* code should not have proper error handling.

In this case we are checking what are called invariants. Invariants are conditions which should always be true in your code. E.g. the number of engines in an engine cluster should always be a positive integer. Zero engines or negative number of engines does not make any sense.

It is not uncommon to check invariants upon entering a function and upon exit: We want to know before doing any computations that assumptions about our inputs hold true. If they are not, we signal that by throwing an exception.

That could be as simple as writing `error("something bad happen")`, or you could go a step further and create your own custom exception type. In this case we are using the `DomainError` exception. It is used to indicate that an input is outside the valid domain of the function. In mathematics we call the collection of valid input values to a function its domain.

Strictly speaking we could have done similar checks for `mass`, `Isp` and `thrust` as well. Mass for instance cannot be negative. Nor can mass be zero, but it makes sense to allow mass to be zero for simulation convenience or simplification. Setting the number of engines to zero however serves no purpose.

## Destructuring

Destructuring is a way to unpack data from some type of collection and place this data in individual variables. This is possible to do in a multitude of ways in Julia. You have already seen some ways of doing it e.g. using the splat operator `....`

Let us define a simple function to demonstrate using destructuring in function calls.

```
decimal(x::Number, y::Number, z::Number) = 100x + 10y + z
```

We can call it in a normal way:

```
julia> decimal(3, 4, 2)
342
```

Or we can place the arguments in an array and use destructuring to use it to supply arguments to `decimal`.

```
julia> digits = [4, 5, 3]
3-element Array{Int64,1}:
 4
 5
 3

julia> decimal(digits...)
453
```

In this case we do the opposite. We are taking an array as input and destructuring into individual variables inside the function.

```
function decimal(nums::Array)
    x, y, z = nums
    100x + 10y + z
end
```

Now we can pass the array directly without destructuring it on the call site.

```
julia> decimal(digits)
453
```

Here is another interesting case. We define a function which takes a tuple as an argument.

```
julia> adder((x, y, z)) = x + y + z
```

Notice that you cannot call it with regular argument. You need to pass a tuple of numbers.

```
julia> numbers = (5, 6, 9) # tuple of numbers
(5, 6, 9)
```

```
julia> adder(5, 6, 9)
ERROR: MethodError: no method matching adder(::Int64, ::Int64, ::Int64)
```

```
julia> adder(numbers) # Passing tuple which get destructureed
20
```

Another common case for destructuring composite data types into individual variables is when looping.

```
julia> list = [("two", 2), ("four", 4), ("five", 5)]
3-element Array{Tuple{String,Int64},1}:
 ("two", 2)
 ("four", 4)
 ("five", 5)
```

Looping without destructuring.

```
julia> for tuple in list
        println("""", first(tuple), " ", last(tuple))
    end
'two' 2
'four' 4
'five' 5
```

Looping with destructuring.

```
julia> for (word, num) in list
           println("""", word, " ", num)
       end
'two' 2
'four' 4
'five' 5
```

The array does not need to be made up of tuples. It could also be mad up of pairs. Dictionaries e.g. contain pairs.

```
julia> for (word, num) in ["two" => 2, "four" => 4, "five" => 5]
           println("""", word, " ", num)
       end
'two' 2
'four' 4
'five' 5
```

So when you got a pair where you want to extract individual values, you don't have to do it like this

```
julia> pair = "eight" => 8
"eight" => 8

julia> word = first(pair)
"eight"

julia> num = last(pair)
8
```

Instead you can use destructuring

```
julia> w, n = pair
"eight" => 8

julia> w
"eight"

julia> n
8
```

And this works for tuples and arrays as well:

```
julia> x, y, z = [3, 4, 8]
3-element Array{Int64,1}:
 3
 4
 8

julia> x
3

julia> y
```

4

## Higher Order Functions in Depth

One of the first things we did in this book in section Make Your Own Map Function was to create our own higher order function called `coolmap`:

```
function coolmap(fun, xs)
    ys = Float64[]
    for x in xs
        push!(ys, fun(x))
    end
    ys
end
```

One of the important traits of Julia which allows us to do that, is that functions can be treated just like any other data. `fun` is passed like any other argument, and later invoked like like a regular function.

However we are not limited to passing functions to functions, we can also return them. The definition of a function is also an expression which returns that function as an object.

```
julia> g = function f(x) 2x + 3 end
f (generic function with 1 method)

julia> g(1)
5

julia> h = f(x, y) = 10x + y
f (generic function with 2 methods)

julia> h(2, 5)
25
```

These are function definitions you have already seen. However in cases like this we don't need to name functions. We can use anonymous functions.

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

julia> g(1)
5

julia> h = (x, y) -> 10x + y
#3 (generic function with 1 method)

julia> h(2, 5)
25
```

These are practical to use for short functions needed when using `map`, `filter` and `reduce`. Let us revisit our artillery trajectory example. First we setup the

variables and constants we need.

```
g = 9.81    # Acceleration of gravity in m/s2
v = 20.0    # Exit velocity of projectile
distances = [5, 7, 8, 9, 12] # Distances we want angle of
```

We can provide the angle function as an anonymous function. Notice we are not giving it any name:

```
julia> map(distance -> 0.5*asin(g*distance/v^2), distances)
5-element Array{Float64,1}:
 0.06146720737250767
 0.0862648302763268
 0.0987405431474216
 0.11127887078489511
 0.14936149929913833
```

## Multiline Anonymous Functions

Often your anonymous functions are longer than what will easily fit on a single line. Julia offers some syntactic sugar to deal with this.

```
map(distance -> 0.5*asin(g*distance/v^2), distances)
```

The code above is equivalent to the following code:

```
map(distances) do distance
    0.5*asin(g*distance/v^2)
end
```

The requirement for this `do ... end` form to work is that the function argument is the first argument. If it isn't, you cannot use `do ... end`. Here is another example to clarify how this works. Say you have a function call looking like this:

```
dostuff((x, y, z)->stuff(x, y) + z, a, b, c)
```

This can be replaced with the following code:

```
dostuff(a, b, c) do x, y, z
    stuff(x, y) + z
end
```

This is why you will notice that it is very common for higher order functions in Julia such as `map`, `filter`, `reduce`, `open` and many others to take a function as their first argument.

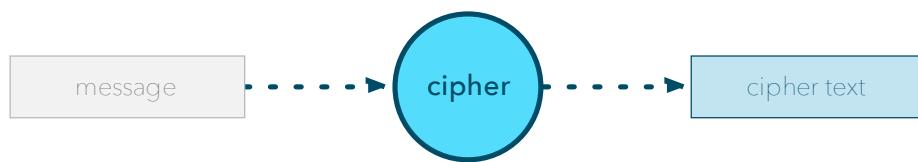
## Implementing a Caesar Cipher in Functional and Object-Oriented Style

To help you understand better what functional programming is about and make it easier for you to see the advantages and disadvantages we will develop the same kind of functionality in both a functional and object-oriented style.

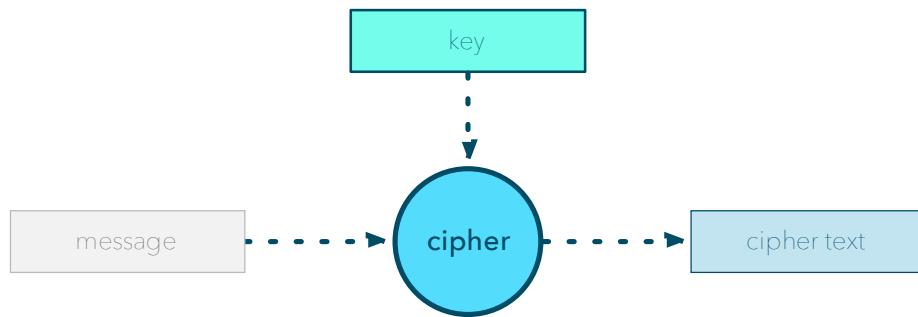
In our example we will look at some of the earliest and simplest forms of encrypting secret messages. We will look at the Caesar cipher which was used by the Roman emperor Caesar to send secret messages to his armies. Next we will implement a slightly more advance encryption algorithm, called a substitution cipher.

## Symmetric Encryption

The type of encryption we are going to use is classified as *symmetric encryption*. In its simplest form we have some plain text, referred to as the ``message'' in cryptography, which gets encrypted by an algorithm. An algorithm performing encryption or decryption is called a ``cipher.'' The encrypted message is called the ``cipher text.''



With the early approaches, the cipher had to be kept secret, because anybody who knew the cipher could decrypt the secret message. So one realized that a secret key will have to be used, so that if the secret method of information exchange was compromised, one would only have to replace the key and not the cipher.



## Cipher Disc

The Caesar cipher was implemented as a physical disc. Each dial would contain all the letters in the alphabet. Alternatively one of the alphabets could be made up of special characters not in the latin alphabet.



When creating the cipher text, you would look at each character in the message in e.g. the inner dial. Then you would look at the corresponding character in the other dial. The encryption key in this case is equal to the *number of characters* you have shifted the outer dial. If the key is 1, it means occurrences of A in the message would be replaced by B in the cipher text. B becomes C, C becomes D etc. However if the shift was set to 2, then A would become C, C would become E etc.

We can accomplish this easily in code:

```
encrypt(ch::Char, shift::Integer) = ch + shift  
decrypt(ch::Char, shift::Integer) = ch - shift
```

Here are some examples of using these functions:

```
julia> encrypt('A', 1)  
'B'
```

```
julia> encrypt('B', 1)  
'C'
```

```
julia> encrypt('B', 2)
```

```
'D'
```

```
julia> decrypt('D', 2)
'B'
```

However there is an important piece missing. The rotating wheel can deal with letters towards the end of the alphabet, while we cannot:

```
julia> encrypt('Z', 1)
'['
```

We don't want the output text to contain characters not in the alphabet used by the input text. What the disc does is to wrap around, so with shift 1, a Z becomes A. We can simulate this mathematically with the modulo operator % or the corresponding mod function. Mathematically speaking it works a bit like an analog clock:

```
julia> 1 % 12
1

julia> mod(1, 12)
1

julia> mod(9, 12)
9

julia> mod(5, 12)
5

julia> mod(13, 12)
1

julia> mod(21, 12)
9

julia> mod(17, 12)
5

julia> 17 % 12
5
```

mod is similar to the rem function which gives the remainder from division. We can utilize mod to make our Caesar cipher wrap around. To get it to work we have to map characters to a value between 0 and 25, because this allows us to shift the values using modulo.

```
const n = length('A':'Z')

function encrypt(ch::Char, shift::Integer)
    'A' + mod((ch - 'A') + shift, n)
end

function decrypt(ch::Char, shift::Integer)
```

```
'A' + mod((ch - 'A') - shift, n)
end
```

We can test this with some edge cases to see it if works:

```
julia> encrypt('A', 2)
'C'
```

```
julia> encrypt('Z', 2)
'B'
```

```
julia> decrypt('C', 2)
'A'
```

```
julia> decrypt('B', 2)
'Z'
```

```
julia> ch = encrypt(' ', 2)
'V'
```

```
julia> decrypt(ch, 2)
'T'
```

As you can see handling of space does not work because it is not in the range A-Z. A simple fix is to simply ignore spaces.

```
function encrypt(ch::Char, shift::Integer)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') + shift, n)
    else
        ch
    end
end

function decrypt(ch::Char, shift::Integer)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') - shift, n)
    else
        ch
    end
end
```

Using `map` we can encrypt a whole message:

```
julia> message = "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
```

```
julia> shift = 2
2
```

```
julia> cipher_text = map(ch -> encrypt(ch, shift), message)
"VJG SWKEM DTQYP HQZ LWORU QXGT VJG NCBA FQI"
```

```
julia> map(ch -> decrypt(ch, shift), cipher_text)
"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
```

## Substitution Cipher

A Substitution Cipher is a generalization of the Caesar cipher. It is based on mapping every letter in the alphabet to another one. However the mapping is not defined by a simple rotation. Instead it is a dictionary where the letters could be randomly selected.

To create a substitution cipher we need to create a mapping between two alphabets. For this purpose we need to use the `shuffle` function found in the `Random` module. To use functions in the `Random` module we need to import it. That is done with the `using Random` statement.

Notice below the usage of the range '`'A':'Z'`' to quickly create a string containing all the letters in the alphabet. `collect` applied to this range would have given us an array of letters. But in this case we want a string, so we use `join` instead.

`shuffle` will randomly rearrange the elements in an array. Remember that a range is a subtype of `AbstractArray`, which is why we can shuffle a range as if it was a regular array.

```
julia> using Random

julia> join('A':'Z')
"ABCDEFGHIJKLMNPQRSTUVWXYZ"

julia> shuffle([1, 2, 3])
3-element Array{Int64,1}:
 2
 3
 1

julia> shuffle(1:5)
5-element Array{Int64,1}:
 1
 2
 5
 4
 3

julia> join(shuffle('A':'Z'))
"PKTAVEQDXGWJMBZ0FSLICRUNYH"
```

This gives us the tools to create a dictionary mapping between the two alphabets. When creating a dictionary we normally need key-value pairs, but in the code below, keys and values are created as separate arrays, so how can we make a dictionary out of this?

```
julia> alphabet = join('A':'Z')
```

```
"ABCDEFGHIJKLMNPQRSTUVWXYZ"
```

```
julia> substitute = join(shuffle('A':'Z'))
"MJVYWPZOSBEFHXTQIAURNKCDL"
```

The `zip` function solves this problem. `zip` can take two arrays of elements and turn them into an iterable object, which when collected gives us an array of pairs.

```
julia> collect(zip('A':'Z', shuffle('A':'Z')))
26-element Array{Tuple{Char,Char},1}:
 ('A', 'O')
 ('B', 'J')
 ('C', 'V')
 ('D', 'Y')
 ('E', 'R')
 ('F', 'N')
 ('G', 'T')
 ('H', 'K')
 ('I', 'B')
 ('J', 'A')
 ('K', 'X')
 ('L', 'F')
 ('M', 'G')
 ('N', 'M')
 ('O', 'E')
 ('P', 'P')
 ('Q', 'D')
 ('R', 'Q')
 ('S', 'L')
 ('T', 'W')
 ('U', 'C')
 ('V', 'I')
 ('W', 'U')
 ('X', 'S')
 ('Y', 'H')
 ('Z', 'Z')
```

Next we use this iterable over key-value pairs to create a dictionary:

```
julia> mapping = Dict(zip('A':'Z', shuffle('A':'Z')))
Dict{Char,Char} with 26 entries:
'E' => 'Z'
'Z' => 'K'
'X' => 'W'
'C' => 'G'
'B' => 'B'
'D' => 'N'
'A' => 'F'
'R' => 'M'
'G' => 'J'
'F' => 'U'
```

```
'N' => 'S'
'M' => 'I'
'K' => 'L'
'J' => 'A'
'O' => 'E'
'I' => 'Q'
'P' => 'P'
'H' => 'R'
'Q' => 'X'
'W' => 'O'
'S' => 'T'
'T' => 'D'
'U' => 'V'
'L' => 'C'
'Y' => 'Y'
'V' => 'H'
```

This gives us input data we can use with an encryption function for the substitution cipher.

```
function encrypt(ch::Char, mapping::Dict{Char, Char})
    get(mapping, char, char)
end
```

And you could use the same function to decrypt, given that you provide a reverse dictionary. However that would potentially be error prone. For anyone looking at the code it would be hard to determine whether a decryption or encryption is going on.

We have not yet gotten into the difference between functional and object-oriented thinking. To do that we need to develop our code example further.

## An Encryption Agnostic Service

Imagine having some sort of service using a cipher and we want to make it easy to swap out which cipher we are using. We are using a toy example just to convey the concept: A password keeping service. It maintains a dictionary with logins as keys and encrypted passwords as values.

```
mutable struct Vault
    passwords::Dict{String, String}
    shift::Int64
end

Vault(shift::Integer) = Vault(Dict{String, String}(), shift)

function addlogin!(vault::Vault, login::AbstractString, password::AbstractString)
    vault.passwords[login] = map(ch -> encrypt(ch, vault.shift), password)
end

function getpassword(vault::Vault, login::AbstractString)
    map(ch -> decrypt(ch, vault.shift), vault.passwords[login])
```

```
end
```

While the code works, there are numerous problems with this approach:

1. We have hard coded it to only support one particular encryption scheme, using a Caesar cipher. There should be a choice of any cipher.
2. The service assumes encryption and decryption is done one character at a time. Encryption should be generalized to deal with whole strings, because it is not necessarily implemented as character substitution.

## Object-Oriented Approach

What we want is to have an abstract interface to any type of cipher so users of a cipher does not need to know any particular details about each type of cipher. We will start with an object oriented approach. First we define a `Cipher` as an abstract type, with a number of functions it has to support.

You will have to add methods to each of these functions to add support for your particular cipher.

```
abstract type Cipher end

function encrypt(cipher::Cipher, char::Char)
    error("Implement encrypt(::", typeof(cipher), ", ", char)()")
end

function decrypt(cipher::Cipher, char::Char)
    error("Implement decrypt(::", typeof(cipher), ", ", char)()")
end

function encrypt(cipher::Cipher, message::AbstractString)
    map(ch -> encrypt(cipher, ch), message)
end

function decrypt(cipher::Cipher, ciphertext::AbstractString)
    map(ch -> decrypt(cipher, ch), ciphertext)
end
```

They way this has been setup, we have made implementing `encrypt` and `decrypt` for *message strings* and *cipher text strings* optional. The default implementation will use `encrypt` and `decrypt` of single characters. However if you have not implemented these, you will get an error message if you try to perform encryption or decryption with your cipher.

First we will make the Caesar cipher implement the `Cipher` interface.

```
struct CaesarCipher <: Cipher
    shift::Int
end

const n = length('A':'Z')
```

```

function encrypt(cipher::CaesarCipher, ch::Char)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') + cipher.shift, n)
    else
        ch
    end
end

function decrypt(cipher::CaesarCipher, ch::Char)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') - cipher.shift, n)
    else
        ch
    end
end

```

This is almost exactly like the previous implementation, except we obtain the shift from the cipher object instead of getting it directly.

The substitution cipher is a bit different because we are maintaining two dictionaries to be able to handle both encryption and decryption.

```

struct SubstitutionCipher <: Cipher
    substitute::Dict{Char, Char}
    alphabet::Dict{Char, Char}

    function SubstitutionCipher(substitute)
        sub = Dict(zip('A':'Z', collect(substitute)))
        alpha = Dict(zip(collect(substitute), 'A':'Z'))
        new(sub, alpha)
    end
end

function encrypt(cipher::SubstitutionCipher, ch::Char)
    get(cipher.substitute, ch, ch)
end

function decrypt(cipher::SubstitutionCipher, ch::Char)
    get(cipher.alphabet, ch, ch)
end

```

This lets us define our password keeping service to be cipher agnostic.

```

mutable struct Vault
    passwords::Dict{String, String}
    cipher::Cipher
end

function Vault(cipher::Cipher)
    Vault(Dict{String, String}(), cipher)
end

```

```

function addlogin!(vault::Vault, login::AbstractString, password::AbstractString)
    vault.passwords[login] = encrypt(vault.cipher, password)
end

function getpassword(vault::Vault, login::AbstractString)
    decrypt(vault.cipher, vault.passwords[login])
end

```

Here are some examples of using our new password keeping service with different ciphers. First with the Caesar cipher.

```
julia> vault = Vault(CaesarCipher(23))
Vault(Dict{String, String}(), CaesarCipher(23))
```

```
julia> addlogin!(vault, "google", "BING")
"YFKD"
```

```
julia> addlogin!(vault, "amazon", "SECRET")
"PBZ0BQ"
```

```
julia> getpassword(vault, "google")
"BING"
```

```
julia> getpassword(vault, "amazon")
"SECRET"
```

Next an example with the substitution cipher:

```
julia> substitute = "CQPYXVFHRNZMWOITJSUBKLEGDA";
julia> cipher = SubstitutionCipher(substitute);
julia> vault = Vault(cipher);
```

```
julia> addlogin!(vault, "amazon", "SECRET")
"UXPSXB"
```

```
julia> addlogin!(vault, "apple", "JONAGOLD")
"NIOCFIMY"
```

```
julia> getpassword(vault, "amazon")
"SECRET"
```

```
julia> getpassword(vault, "apple")
"JONAGOLD"
```

## Functional Approach

The point of showing how to accomplish the abstraction using an object-oriented approach is because more programmers are already familiar with this approach. In this case it is also probably the best approach to solving this particular problem.

What does object-oriented mean in this case? It means we are solving the problem by thinking in terms of type hierarchies and objects. We represented our cipher as an object and defined functions with methods that operated on these cipher objects.

With our functional approach we will instead aim to solve the problem by thinking in terms of functions: higher order functions and closures.

We start by defining the caesar cipher.

```
const n = length('A':'Z')

function char_encrypt(ch::Char, shift::Integer)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') + shift, n)
    else
        ch
    end
end

function caesar_encrypter(shift::Int)
    str -> map(ch -> char_encrypt(ch, shift), str)
end

function caesar_decrypter(shift::Int)
    caesar_encrypter(-shift)
end
```

Notice how there are no cipher types anymore. There is no data object representing the caesar cipher. Instead we have two higher order functions `caesar_encrypter` and `caesar_decrypter` returning functions. `caesar_encrypter` returns a function which can perform a Caesar encryption of a text string.

`caesar_decrypter` is implemented with a trick. It return the same function as `caesar_encrypter` but with the `shift` argument negated.

Here is how you would create and use the encrypt and decrypt functions:

```
julia> encrypt = caesar_encrypter(1)

julia> encrypt("ABC")
"BCD"

julia> decrypt = caesar_decrypter(1)

julia> decrypt("BCD")
"ABC"
```

One interesting thing to notice here is that you don't have to pass in state to the function call such as the `shift` value or a cipher object. Instead the functions *remember* the shift value. Functions which capture their surrounding environment and remembers it, are called closures. Think of a closure as a function

plus state. In this regard closures are a bit like an object in object-oriented programming with a single method.

In a lot of cases this gives a more elegant solution than an object-oriented approach. However in this particular case it is inconvenient. Since we can only associate one function with some data at a time, we end up having to create two separate closures: one for encryption and one for decryption. You don't have any guarantees that the decryption function is using the same `shift` value as the encryption function.

## Closures

It is worth examining closures more in detail as they are frequently misunderstood by developers. A very common misconception is to think of closures are just anonymous functions, that is functions without names.

There are several reasons why people tend to think this:

1. Anonymous functions are very frequently used where a closure is needed.
2. Developers have a tendency to think every language feature has a specific corresponding syntax. Just like making a compound type requires the `struct` keyword. Making a function uses the `function` keyword etc.

However there is no special keyword for making a closure. It is not like you prefix a function with the word `closure`. Thus some people go ``aha, so the syntax for writing a closure is to remove the function name!'' No! Wrong, wrong and wrong.

I will demonstrate why this is not the case.

```
function caesar_encrypter(shift::Int)
    str -> map(ch -> char_encrypt(ch, shift), str)
end
```

We will rewrite this function to use a named function instead:

```
function caesar_encrypter(shift::Int)
    encryp(str) = map(ch -> char_encrypt(ch, shift), str)
end
```

If we try this new definition in the Julia REPL, you will find that it works exactly the same.

```
julia> encrypt = caesar_encrypter(1)

julia> encrypt("ABC")
"BCD"
```

The reason we frequently use anonymous functions rather than named functions when creating a closures is simply due to a combination of convenience and clarity. We don't *need* the name. Functions defined inside other functions will not have their name visible outside that function.

It is also a form of communication to the reader of the code. By not providing a function name, we also make it clear to the reader, that the name will not be used anywhere else in the code.

## Substitution Cipher

We do a bit of trickery to define the substitution cipher. We let the encryption and decryption creator functions both take an alphabet and a substitution string.

This makes it easy to implement the `substitution_decrypter` function by simply calling `substitution_encrypter` with the arguments reversed. Hence the ``alphabet'' of the decrypter becomes the substitution characters used by the encryptor.

```
function char_encrypt(ch::Char, mapping::Dict{Char, Char})
    get(mapping, ch, ch)
end

function substitution_encrypter(
    substitute::AbstractString,
    alphabet = join('A':'Z'))
    mapping = Dict(zip(collect(alphabet),
                        collect(substitute)))
    str -> map(ch -> char_encrypt(ch, mapping), str)
end

function substitution_decrypter(
    substitute::AbstractString,
    alphabet = join('A':'Z'))
    substitution_encrypter(alphabet, substitute)
end
```

## Password Keeper

Now let us put it all together and create a password keeper which uses our encryption and decryption functions to allow logins and passwords to be stored and retrieved.

```
mutable struct Vault
    passwords::Dict{String, String}
    encrypt::Function
    decrypt::Function
end

function Vault(encrypter, decrypter)
    Vault(Dict{String, String}(), encrypter, decrypter)
end

function addlogin!(vault::Vault, login::AbstractString, password::AbstractString)
    vault.passwords[login] = vault.encrypt(password)
end

function getpassword(vault::Vault, login::AbstractString)
    vault.decrypt(vault.passwords[login])
```

```
end
```

Let us look at an example of using this implementation to define a password keeper using a Caesar cipher:

```
julia> shift = 23
23

julia> vault = Vault(caesar_encrypter(shift), caesar_decrypter(shift))

julia> addlogin!(vault, "google", "BING")
"YFKD"

julia> addlogin!(vault, "amazon", "SECRET")
"PBZ0BQ"

julia> getpassword(vault, "google")
"BING"

julia> getpassword(vault, "amazon")
"SECRET"
```

And here we have the Substitution cipher:

```
julia> using Random

julia> substitute = join(shuffle('A':'Z'))
"KCGFWBHDPJAXMELOUYSNVRQZTI"

julia> vault = Vault(substitution_encrypter(substitute), substitution_decrypter(subst
    ...

julia> addlogin!(vault, "amazon", "SECRET")
"SWGYNW"

julia> addlogin!(vault, "apple", "JONAGOLD")
"JLEKHLXF"

julia> getpassword(vault, "amazon")
"SECRET"

julia> getpassword(vault, "apple")
"JONAGOLD"
```

Now it is time to take a few steps back and reflect upon, why exactly we designed our closures the way we did. The objective was the same as with the object-oriented case, to present a generic interface to ciphers, so that we can change what cipher is used without changing the implementation of the password keeper implementation.

The way we did this was by returning encryption and decryption functions which don't expose any implementation details in their function signature. A function signature is what arguments a function takes, their order and type.

The caesar cipher and substitution cipher produce encryption and decryption functions with the same signatures. That is why they are interchangeable.

## Function Composition

While functional programming has some downsides in this example, using simple functions which only take generic data as input, makes them easily composable. For instance if I want my encryption function to be case insensitive I can create a new function by combining the caesar encryption function with uppercase:

```
julia> encrypt = caesar_encrypter(1) ∘ uppercase

julia> encrypt("abc")
"BCD"

julia> encrypt("aBc")
"BCD"
```

The function composition operator  $\circ$ , is a higher order function defined like this:  
 $\circ(f, g) = (x...)->f(g(x...))$

### NOTE

To write  $\circ$  in the Julia REPL you write `\circ` and hit the tab key. It will complete to the  $\circ$  symbol. This is utilizing the Julia LaTeX completion feature.

What you see from this definition is that the function composer takes two functions  $f$  and  $g$  as arguments and returns a new function, let us call it  $h$  for convenience. Every argument  $h$  takes when it is called is passed to the  $g$  function, and the result of  $g$  is passed to  $f$ .

So to explain what happens when you `caesar_encrypter(1) ∘ uppercase`, let us pretend `caesar_encrypter(1)` returns a function called `caesar_encrypt`. Thus we are looking at what `caesar_encrypt ∘ uppercase` does:

```
function ∘(caesar_encrypt, uppercase)
    (message)->caesar_encrypt(uppercase(message))
end
```

## Partial Application

We don't intend to go deep into functional programming in this book, but it is useful to be aware of functional concepts because they crop up in quite frequently in Julia APIs.

Julia has various `find` functions for searching an array for particular elements. Just write `find` in the Julia REPL and you get this list:

```
julia> find
findall    findlast    findmax!    findmin!    findprev
```

```
findfirst findmax    findmin    findnext
```

We will look at the function called `findall` which locates the indices of *all* the elements matching a predicate (functions returning `true` or `false`). Now that you know about the shorthand for anonymous functions, you may write something like this to find all occurrences of the number 6:

```
julia> findall(x -> x == 6, [3, 4, 6, 7, 6])
2-element Array{Int64,1}:
 3
 5
```

However there is an even more compact way of writing this:

```
julia> findall==(6), [3, 4, 6, 7, 6])
2-element Array{Int64,1}:
 3
 5
```

This utilizes a concept often referred to as *partial application* in functional programming. The idea is that when we don't apply all the arguments to a function, a function taking the rest of the arguments will be returned. Some functional languages such as Haskell has built in support for this, meaning every single function in the language supports partial application.

Julia has no builtin support for partial application. Rather it offers the features allowing you to build such functions if you so wish.

A few selected functions in the standard library such as `==`, provides partial application capability, because it is something one would frequently use.

However it is very easy to add partial application to any function you like. A simple straightforward way to do it would be to implement partial application for `==` and `!=` like this:

```
==(y) = x -> x == y
!=(y) = x -> x != y
```

However the standard library actually has convenience functions for doing this called `Fix1` and `Fix2`. The name is a play on the fact that the fix either the first or the second argument of a binary function.

The code example below are all equivalent, which should help you understand how `Fix1` and `Fix2` works:

```
>=(y) = Fix2(>=, y)
>=(y) = x -> x >= y

>=(x) = Fix1(<=, x)
>=(x) = y -> x <= y
```

Julia has partial application built in for a lot of functions. Here are some more examples of using partial application on binary predicates:

```
julia> findall(>(6), [3, 4, 6, 7, 6])
1-element Array{Int64,1}:
 4
```

```
julia> findall(<(6), [3, 4, 6, 7, 6])
2-element Array{Int64,1}:
 1
 2

julia> filter(<(6), [3, 4, 6, 7, 6])
2-element Array{Int64,1}:
 3
 4

julia> filter(>(6), [3, 4, 6, 7, 6])
1-element Array{Int64,1}:
 7
```

This also beautifully illustrates the power of multiple-dispatch. It shows the benefits of allowing you to register methods on a function taking different number of arguments and doing entirely different things.

The other reason this works is because operators are just functions in Julia. The `==`, `>`, `<` etc operators are made to function differently depending on the number of arguments provided. If you give two arguments it returns a boolean. If give a single argument it returns a closure instead.

## Broadcast: map's powerful sibling

Almost any discussion of functional programming will touch upon higher order functions such as `map`, `filter` and `reduce`. Julia has a more powerful function called `broadcast`, which is probably used more by Julia developers than `map`.

To motivate the need for broadcast we need to look at what `map` can actually do and where it falls short.

```
julia> map(+, 2, 3)
5

julia> map(+, [2], [3])
1-element Array{Int64,1}:
 5

julia> map(+, [2, 1], [3, 2])
2-element Array{Int64,1}:
 5
 3

julia> map(sqrt, 9)
3.0
```

As you can see `map` can deal sensibly with both scalars and vectors separately. The problem starts when you mix them:

```
julia> map(+, [2, 1], 3)
```

```
1-element Array{Int64,1}:
5
```

That is not quite what we wanted or expected. In these cases `broadcast` is really what you need.

```
julia> broadcast(+, [2, 1], 3)
2-element Array{Int64,1}:
5
4
```

You could use it exactly like `map`, so you don't have to choose if you don't want to:

```
julia> broadcast(+, 3, 2)
5

julia> square(x) = x^2;

julia> broadcast(square, [1, 2, 3])
3-element Array{Int64,1}:
1
4
9
```

However the real power is being able to handle combinations of scalars and arrays seamlessly.

```
julia> broadcast(*, "number ", ["one", "two", "three"])
3-element Array{String,1}:
"number one"
"number two"
"number three"

julia> broadcast(+, [10, 100, 1000], [2, 4, 8])
3-element Array{Int64,1}:
12
104
1008
```

What you see here is that `broadcast` can deal with functions taking any number of arguments. However it is smart about how it deals with combinations of scalars (single values) and arrays. `+` takes two arguments. If both arguments are scalars such as `3` and `2`, `broadcast` will invoke `+` with these arguments once. However if one of the arguments is an array, it will invoke `+` repeatedly providing values from the array in succession as an argument. If the other argument is a scalar, the same value will be used repeatedly on each invocation.

Thus `broadcast(+, 3, [1, 2, 3])` will cause the following calculations to happen:

```
[3 + 1, 3 + 2, 3 + 3]
```

But we have only gotten started. It gets better. Almost nobody actually calls `broadcast` directly. Julia offers a syntax-sugar version of `broadcast`, which

most Julia developers use. How you use it depends on whether you are using infix or prefix notation<sup>26</sup>.

Common operators such as `+`, `-` and `*` are usually written with infix notation. In this case we prefix a `.` to perform broadcast.

```
julia> 3 .+ [1, 2, 3]
3-element Array{Int64,1}:
 4
 5
 6

julia> [10, 20, 30] .+ [1, 2, 3]
3-element Array{Int64,1}:
 11
 22
 33
```

However when using prefix notation, such as performing a regular function call, we need to suffix the function name with a `.` to perform broadcast.

```
julia> square.([1, 2, 3])
3-element Array{Int64,1}:
 1
 4
 9

julia> first(["hello", "to", "you"])
"hello"

julia> first.(["hello", "to", "you"])
3-element Array{Char,1}:
 'h'
 't'
 'y'

julia> last(["hello", "to", "you"])
"you"

julia> last.(["hello", "to", "you"])
3-element Array{Char,1}:
 'o'
 'o'
 'u'

julia> heroes = ["batman", "superman", "flash"]
3-element Array{String,1}:
 "batman"
 "superman"
```

---

<sup>26</sup>Infix notation means the operator is between the arguments. `2 + 3` is an example of infix notation, while `+(2, 3)` is an example of prefix notation.

```
"flash"
```

For index or key access we need to get a bit creative to get it to work. Let us look at a simple example.

```
julia> heroes[2]
"superman"
```

This is identical to the function call below:

```
julia> getindex(heroes, 2)
"superman"
```

With this in mind we have a way of performing index access on every element of collection.

```
julia> getindex.(heroes, 2)
3-element Array{Char,1}:
 'a'
 'u'
 'l'

julia> parse.(Int, ["42", "1337"])
2-element Array{Int64,1}:
 42
 1337
```

The benefits of using `.` notation is that it can easily be chained.

```
julia> 1 .+ parse.(Int, ["42", "1337"])
2-element Array{Int64,1}:
 43
 1338

julia> string.([15, 11, 26, 42], base=16)
4-element Array{String,1}:
 "f"
 "b"
 "1a"
 "2a"

julia> uppercase.(string.([15, 11, 26, 42], base=16))
4-element Array{String,1}:
 "F"
 "B"
 "1A"
 "2A"
```

This makes it easy to combine functions operating on multiple elements without writing the code any different than if you operated on single elements (scalars).

## Function Chaining with |>

One occasional complaint from people coming from object-oriented programming to functional programming, is that reading code invoking multiple functions can be hard.

In object oriented programming it is easy to chain several method calls. Look at the example below. The result of `foo()` is passed to `bar()`, which passes its result to `stuff()` and so on.

```
obj.foo().bar().stuff().more_stuff()
```

With Julia this would be harder to read, when trying to follow the sequence of evaluation:

```
more_stuff(stuff(bar(foo(obj))))
```

This is less natural for a human to parse. The Julia solution is to either split the expression over multiple lines or use the Julia pipe operator `|>`.

```
obj |> foo |> bar |> stuff |> more_stuff
```

Interestingly this operator can be used with broadcast, so we can rewrite the line `uppercase.(string.([15, 11, 26, 42], base=16))` to:

```
julia> string.([15, 11, 26, 42], base=16) .|> uppercase
4-element Array{String,1}:
 "F"
 "B"
 "1A"
 "2A"
```

## How to Think Functional

There are many benefits to programming in a functional style, but it is not always obvious how you do it. One of the simplest ways of getting into the habit of functional programming is to do a lot of your programming right in the Julia REPL environment.

This encourages functional thinking for several reasons:

1. It encourages writing lots of small functions doing one simple things. The reason being that, writing long functions is cumbersome in a REPL environment.
2. It discourages writing mutating functions. In a REPL environment you are constantly testing the code you are writing. Mutating functions are inconvenient to use in a REPL setting. State has to be frequently reset to test functions.
3. Functions taking some input and returning some output are more convenient to use in a REPL environment. Functions without return values or input are much harder to play around with.

When writing in a text editor, it is easy for the beginner to write what the Julia creators would call scripts: long reams of code not contained in any function.

Writing code in a REPL force you to put code into functions, otherwise it is very hard to modify that code or run it.

Common advice on how to approach problems in a more function manner is to stop automatically reaching for for-loops when processing multiple elements. Try instead to see if you can solve the problem using `map`, `filter` or `reduce`. In Julia's case `broadcast` may often be a better choice.

But be pragmatic. Julia was not designed to be exclusively written in a functional style. If you try to insist on this in everything you do, you will just make it harder for yourself to write Julia code. Keep it simple. Sometimes an imperative approach is perfectly fine.

In the cipher example we covered I would argue that the object-oriented approach turned out better. In practice I find myself often prototyping in a functional style, because it can be a very fast way to work, and then later adopt a more object-oriented style when needed.

## **The Big Picture**

We have covered many aspects of functional programming, and I don't necessarily expect you to be using all these features or fully understand them. The important thing is to be aware of them. This will aid understanding when we explore other Julia functionality later in this book.

For instance when working on input and output to files or the network, it is common to use closures. Basically later chapters will expose you to more practical usage of the features introduced in this chapter.

Julia is a flexible language, so you don't need to program in functional style if you don't want to. You can just pick the features you like and ignore the rest.

# Object-Oriented Programming

- How to think about **Inheritance** in Julia.
- **Design Patterns** in Julia. Understand how common object-oriented design patterns translate to Julia code.

Julia is *not* an object-oriented language, but what does that even mean? What makes a language object-oriented in the first place?

Most languages used today are not purely functional, object-oriented, procedural, imperative or declarative. They support multiple programming paradigms. Julia is no different.

As we discussed with functional programming, a programming paradigm is about the *practices* and conventions you follow when using the language. We tend to label a language as functional if it *encourages* and supports a functional programming paradigm. Likewise object-oriented languages frequently support other paradigms of programming but they have been designed specifically to support the object-oriented paradigm.

Julia is not an object-oriented language in the sense that object-oriented programming is not what Julia was designed for. However that does not mean that you cannot apply object-oriented principles and practices to your Julia code.

If this book was only written for people without any prior programming experience, then this chapter would most likely not exist. There is no particular good reason why a Julia developer should learn to think object-oriented.

However we must deal with reality. Like perhaps most developers today, I have spent years steep in object-oriented thinking as it has been the most dominant paradigm for decades. If you have struggled for years to grasp object-oriented programming and then suddenly get thrown into a new language where object-oriented thinking isn't central, it can be disorienting.

The purpose of this chapter is to help developers with extensive object-oriented experience reuse and adapt their skills to Julia.

## What is Object-Oriented Programming?

This book is not intended to be an extensive primer on object-oriented programming. Object-oriented programming is a rather large topic so we have to keep it simple. Basically it is about organizing your programs around the ideas of objects imbued with behavior which can interact with each other.

### Contrasting Object-Oriented Programming with Other Paradigms

The alternative would be e.g. to organize your program around activities, services or data transformations.

Let us take our rocket example. The object-oriented developer will think about what objects the rocket is made up of, such as the rocket engine, propellant tanks, stages, payload and what each of them can do as well as their relationship with each other.

Someone thinking in a data-oriented fashion in contrast may instead think in terms of what are the input data? Upon reflection he or she will conclude that the inputs are things such as the amount of propellant, efficiency of rocket engine etc. The next question is what is the output? What are we trying to achieve? That answer to that could be final velocity or elevation of rocket. The next stage for a data-oriented programmer is to figure out what data *transformations* need to be performed to get from the input data to the output data.

The functional programmer in contrast thinks more like a mathematician. He or she is thinking in terms of functions such as Newton's equations for motion. E.g that acceleration is a function of force and mass. He can then decompose this problem further by concluding that mass is a function of mass flow etc.

Regardless of approach followed one will end up creating functions and types. But there is a difference in what comes first. The functional or data-oriented programmer will focus on functions first. Types are just a *consequence* of what these functions need as input.

For an object-oriented programmer it is the reverse. They will design the types (or the classes specifically) first. He or she will think about the relations between the objects. Which object knows about which other object? What do they do to each other? Meaning what member function does e.g. object a call on object b.

To an object-oriented programmer types should be something tangible in the real world. To a data-oriented programmer a type may not necessarily correspond closely to any concrete object in the real world. It will be just whatever bundle of data is needed to be feed into his/her data processing pipeline to get the end result.

#### **NOTE Methods in object-oriented programming**

In object-oriented terminology, *methods* are functions attached to classes. Sometimes called *member functions*. There is some resemblance to the Julia concept of methods as they also relate

to the ability to execute different function implementations depending on the type of arguments. The difference is that for OOP, only the first implied argument, corresponding to `this` or `self` matters, while in Julia every argument influence what method gets picked.

## Language Features

Object-oriented programming languages typically support a number of features less common in languages focusing on other paradigms.

- **Implementation inheritance.** In object-oriented languages we can define a base class with data members and member functions providing functionality. Sub-classes can then add data or member functions as well as redefine the behavior of member functions defined in the base class.
- **Runtime polymorphism.** This is what we have discussed earlier as *single dispatch*. It means deciding on what particular method to call at runtime. This is a subset of what Julia does through multiple-dispatch and hence not unique to object-oriented languages. However it is usually a key feature.
- **Encapsulation.** While we also encapsulate in Julia, encapsulation tends to be more heavily promoted and emphasized in OOP. In Julia hiding implementation is to a larger degree focused at the module level rather than on individual types within a module.

Of these features implementation inheritance is what is most poorly supported in Julia. One can mimic this in Julia but it is not a natural fit for the Julia language.

Let us look at examples of how common patterns from object-oriented thinking can be adapted to Julia.

## Inheritance Using The Template Method Pattern

A well known design pattern from object-oriented programming, is called the *template method*. The idea is that a member function implemented in a base class, the *template method*, calls more primitive member functions which has to be implemented in subclasses.

This is the best way of dealing with implementation inheritance in Julia. We have already looked at an example of this with the Cipher examples in the previous chapter.

### NOTE UML Diagrams

Throughout this book we have used a lot of UML diagrams. These diagrams was developed as notation to describe object-

oriented systems. In standard UML diagrams we would write `encrypt(ch : Char)` rather than the Julia notation `encrypt(cipher::Cipher, ch::Char)`. This is because the first argument is assumed to be of the same type as the class (type).

In UML notation which I am using here, member functions which have to be implemented in subclasses are marked in italics. `encrypt(msg : AbstractString)` is a template method here. It is implemented in the base class using the `encrypt(ch : Char)` method which must be implemented in a subclass.

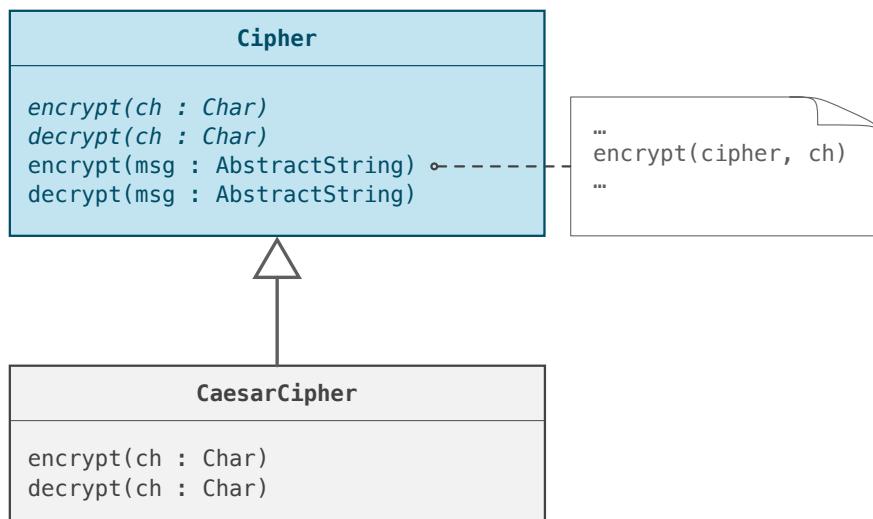


Figure 54: A UML inheritance diagram showing the template method design pattern.

At the top we have an interface or abstract base class called **Cipher** and finally a concrete subclass called **CaesarChiper**.

The equivalent of subclassing was demonstrated in the this code that you saw earlier.

```

struct CaesarCipher <: Cipher
  shift:Int
end

const n = length('A':'Z')

function encrypt(cipher::CaesarCipher, ch::Char)
  if ch in 'A':'Z'
    'A' + mod((ch - 'A') + cipher.shift, n)
  else
    ch
end
  
```

```

end

function decrypt(cipher::CaesarCipher, ch::Char)
    if ch in 'A':'Z'
        'A' + mod((ch - 'A') - cipher.shift, n)
    else
        ch
    end
end

```

In OOP terminology `encrypt(cipher::CaesarCipher, ch::Char)` would be a virtual method that was overridden. In Julia terminology we have simply registered a method in the method table for the `encrypt` function with a more concrete type for the `cipher` argument.

This is basically how we defined the rough equivalent of an abstract base class in Julia:

```

abstract type Cipher end

function encrypt(cipher::Cipher, char::Char)
    error("Implement encrypt(::", typeof(cipher), ", ", char)()")
end

function decrypt(cipher::Cipher, char::Char)
    error("Implement decrypt(::", typeof(cipher), ", ", char)()")
end

function encrypt(cipher::Cipher, message::AbstractString)
    map(ch -> encrypt(cipher, ch), message)
end

function decrypt(cipher::Cipher, ciphertext::AbstractString)
    map(ch -> decrypt(cipher, ch), ciphertext)
end

```

This approach mimics a common approach in dynamic object-oriented programming languages where one cannot enforce that a subclass implement a specific set of methods.

However this may not always be the best approach in Julia. Many Julia libraries will simply not create an implementation throwing an error exception.

With error exception we get the following output if we forgot to implement the `encryption` function for individual characters.

```
julia> encrypt(CaesarCipher(5), "HELLO")
ERROR: Implement encrypt(::CaesarCipher, char)
```

However the error message we get if we didn't provide this default implementation is in many cases even more useful.

```
julia> encrypt(CaesarCipher(5), "HELLO")
ERROR: MethodError: no method matching encrypt(::CaesarCipher, ::Char)
```

```
Closest candidates are:
  encrypt(::Cipher, ::AbstractString)
```

Thus some people in the Julia community consider the `NotImplemented` error exception as an anti-pattern. But what is the alternative? How can we communicate to a user of our abstract types what they need to implement?

You can do this in the documentation. If no `encrypt` or `decrypt` method is implemented for the abstract type `Cipher`, then you can simply define the functions:

```
abstract type Cipher end
"""
    encrypt(cipher, char::Char)
Encrypts a character `ch` using `cipher`.
"""
function encrypt end
"""
    decrypt(cipher, ch::Char)
Decrypts a character `ch` using `cipher`.
"""
function decrypt end
```

If we then try to use a `CaesarCipher` which has not implemented any of the encryption and decryption functions we will get the error message:

```
julia> encrypt(CaesarCipher(5), "HELLO")
ERROR: MethodError: no method matching encrypt(::CaesarCipher, ::String)
```

This tells us valuable information, such as the fact that a function named `encrypt` actually exists. If we had not defined the function, we would instead have gotten the following error message:

```
julia> encrypt(CaesarCipher(5), "HELLO")
ERROR: UndefVarError: encrypt not defined
```

Thus this will naturally spur the developer to investigate further.

```
julia> methods(encrypt)
# 0 methods for generic function "encrypt":

help?> encrypt
search: encrypt

    encrypt(cipher, char::Char)

    Encrypts a character ch using cipher.
```

This tells the developer that there is in fact a function named `encrypt` but it has no methods. The developer ought to then check the documentation, where

he/she will see a description of what arguments is expected for an `encrypt` method and what it is supposed to do.

## Mimic Overriding Methods

In object-oriented programming it is common in an overridden method to call the method implementation of the super-type. That would be unusual to do in Julia. It is also a pattern hard to get right. When working with object-oriented libraries people often struggle with determining whether they need to call the super class and if they have to, whether they should do that *before* calling their own custom code.

However such behavior is possible to mimic in Julia with the `invoke` function. Here we replace the `encrypt` function defined for the abstract `Cipher` type with one for the concrete `CaesarCipher`.

```
function encrypt(cipher::CaesarCipher,
                 ciphertext::AbstractString)
    println("inside CaesarCipher")
    invoke(encrypt,
           Tuple{Cipher, AbstractString},
           cipher,
           ciphertext)
end
```

If we run this in the REPL environment we get:

```
julia> encrypt(CaesarCipher(5), "HELLO")
inside CaesarCipher
"MJQQT"
```

So you can see we are able to call the ``original'' version of `encrypt` and add behavior to it.

## Callable Objects and Command Pattern

Most of the catalog of design patterns was tailored for statically typed object-oriented languages. For this reason design patters is seldom discussed with dynamic languages. Most of the time they are superfluous.

One example of this is the *Command Pattern*. Usually it is implemented by having an abstract base class `Command` which defines an `execute` method which must be implemented by concrete subclasses.

The point of this is to be able to bundle up some elaborate action one wants to perform later, which may have a lot of dependencies.

One example may be from a drawing program. You have menu entries for e.g. opening a new document, duplicating a selected shape or group selected shapes. You don't want the subsystem handling the menu entries to have direct knowledge of how all this works in the editor.

Hence you can register commands for doing different things from different subsystems. The subsystem handling drawing can create a command for grouping shapes which it passes over to the menu system.

However you don't need anything like defining interfaces and subclasses in Julia to do that, because functions are first class objects. Meaning you can pass around functions just like you do with any other object.

A function can be a closure, capturing external state, which it then carries with itself.

One of the benefits of the command pattern is that the command object expose properties which can be adjusted by the users of the command object. A simple function does not give you this option in Julia. However Julia has a much better alternative which is *callable objects*.

Let us look at an example of how this can be used. Say we got this polynomial:

$$f(x) = ax^2 + bx + c$$

We can represent this polynomial in code:

```
mutable struct Polynomial
    a::Real
    b::Real
    c::Real
end
```

We make the object itself callable with this definition:

```
(f::Polynomial)(x::Real) = f.a*x^2 + f.b*x + f.c
```

This allows us to construct a polynomial object and later use it as a regular function:

```
julia> g = Polynomial(0, 3, 0)
Polynomial(0, 3, 0)
```

```
julia> g(2)
6
```

One possible use of the command pattern is to support undo. In this case the Command class would have an `undo` method which needs to be implemented by subclasses.

However we can achieve this with callable objects as well. Because a callable object is a regular object with a type we can create an `undo` method which takes a Command object as an argument.

## Factory Method

With the factory method pattern we are creating a special class who's main task is to create instances of other objects. However this is pointless in any language where types are first class objects.

Instead of a factory object, you can simply use the type itself.

```
julia> struct Foo
    end

julia> struct Bar
    end

julia> factory = Foo
Foo

julia> object = factory()
Foo()

julia> factory = Bar
Bar

julia> object = factory()
Bar()
```

## Abstract Factory Pattern

The *abstract factory* pattern is used when the factory object can create a whole family of related products.

What situation may that be useful? Imagine you are assembling a rocket out of different parts such as stages, engines and payload.

You write a function called `assemble_rocket` which spits out say a 2-stage rocket. However there are many 2-stage rockets you could build. Falcon 9, is a 2-stage rocket. So is the New Glenn and Electron rocket.

Why repeat the same assembly code for rocket assembly several times over?

Instead we can pass an abstract factory object to the `assemble_rocket` function. Then you can ask that factory to give you engines for the bottom stage, engines for the second stage, propellant tanks etc. Then you put them together.

By supplying different factory objects we can build different rockets. One factory object could produce Falcon 9 type of propellant tanks and engines, while another factory object could produce parts for Rocket Lab's Electron rocket.

What this diagram is showing is some abstract types in blue and concrete types in gray boxes. The idea is that a `FalconFactory` produces a variety of parts specific to Falcon rockets, while the `ElectronFactory` produce parts specific to the electron rocket.

For instance depending on the factory used, you will get different rocket engines:

```
julia> factory = ElectronFactory()

julia> sl_engine(factory)
```

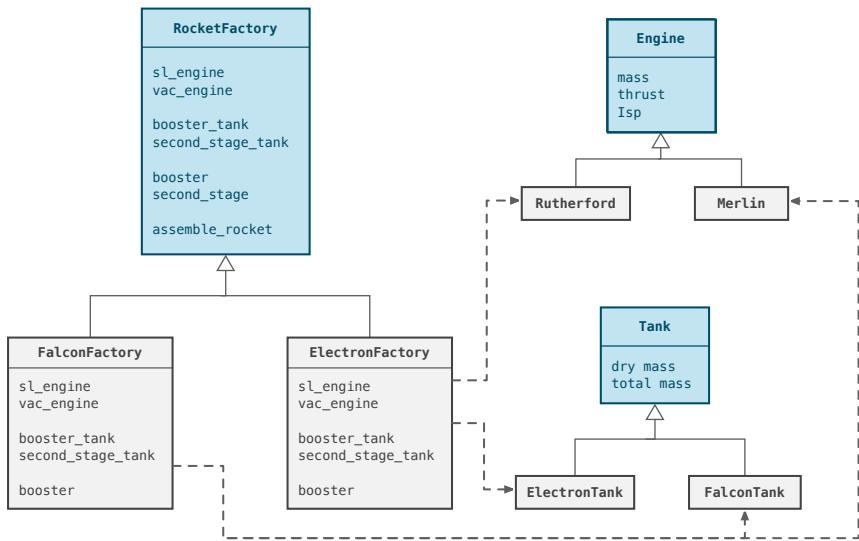


Figure 55: UML diagram of Abstract Factory pattern.

```

engine:
  name   = Rutherford
  thrust = 25000.0
  Isp    = 311.0

```

```

julia> factory = FalconFactory()

julia> sl_engine(factory)
engine:
  name   = Merlin 1D
  thrust = 845000.0
  Isp    = 282.0

```

The Falcon rocket uses Merlin engines while the Electron rocket uses Rutherford engines.

We can also use the factory to assemble a whole rocket:

```

julia> assembleRocket(factory)
stage 2:
  throttle  = 1.0
  propellant = 92670.0

  tank:
    dry     = 3900.0
    total   = 96570.0

  engine:
    name   = Kestrel 2

```

```

thrust = 31000.0
Isp    = 311.0

stage 1:
throttle  = 1.0
propellant = 395700.0

tank:
dry     = 23100.0
total   = 418800.0

engine:
name    = Merlin 1D
thrust  = 7.605e6
Isp    = 282.0

```

Our example of implementing this will *not* use different types for Falcon and Electron engines and tanks. The reason is simply that we don't need it in this case.

## Defining Factory Interface

Let us start by showing how we would typically define an interface. First we define the abstract type which other concrete types must subtype.

```
"Produces rocket parts for a particular type of rocket"
abstract type RocketFactory end
```

Next we define the functions which subtypes must add methods to. Notice the importance of documentation. We must document what kind of arguments are expected for the methods we add. The function definition itself contains no such information. It could not contain it since Julia allows adding methods with any number of arguments and type for their arguments.

```
"""
    sl_engine(factory::RocketFactory) -> Engine
Produce a sea level rocket engine.
Concrete factories must implement this method.
"""

function sl_engine end

"""

    vac_engine(factory::RocketFactory) -> Engine
Produce a vacuum level rocket engine.
Concrete factories must implement this method.
"""

function vac_engine end

"""

    booster_tank(factory::RocketFactory) -> Tank
Create a propellant tank for the booster stage.
"
```

```

Concrete factories must implement this method.
"""
function booster_tank end

"""

    second_stage_tank(factory::RocketFactory) -> Tank
Create a propellant tank for the second stage.
Concrete factories must implement this method.
"""
function second_stage_tank end

"""

    booster(factory::RocketFactory) -> Rocket
Have the rocket `factory` produce a booster stage.
Concrete factories must implement this method.
"""
function booster end

```

Finally we are utilizing the template method pattern. While the second stage could be its own type, it will normally be created by combining a tank suitable for the second stage and a single rocket engine suitable for vacuum.

This means that makers of rocket factory types don't have to code the `second_stage` method unless they have special needs. Using the default implementation will work in most cases.

```

"""

    second_stage(factory::RocketFactory) -> Rocket
Have the rocket `factory` produce a second stage.
"""

function second_stage(factory::RocketFactory)
    tank    = second_stage_tank(factory)
    engine  = vac_engine(factory)
    Rocket(nopayload, tank, engine)
end

```

We do the same for the `assemble_rocket` function. Most modern space vehicles will be made up of two stages, so that is what we build by default.

```

function assemble_rocket(factory::RocketFactory)
    SpaceVehicle([booster(factory), second_stage(factory)])
end

```

## Creating an Electron Factory

Let us create a factory for producing Rocket Lab's Electron rocket. This is an innovative rocket using small engines called Rutherford where the turbo pump is powered by lithium-ion batteries, a highly unusual choice.

First we subtype the `RocketFactory`

```
struct ElectronFactory <: RocketFactory end
```

Next we add methods for creating engines and tanks for the Electron rocket.

```
function sl_engine(::ElectronFactory)
    Engine("Rutherford", 25e3, 311, mass = 35)
end

function vac_engine(::ElectronFactory)
    Engine("Rutherford", 26e3, 343, mass = 35)
end

function booster_tank(::ElectronFactory)
    Tank(0.95e3, 10.2e3)
end

function second_stage_tank(::ElectronFactory)
    Tank(0.25e3, 2.3e3)
end

function booster(factory::ElectronFactory)
    tank    = booster_tank(factory)
    engine  = sl_engine(factory)
    cluster = EngineCluster(engine, 9)
    Rocket(nopayload, tank, cluster)
end
```

As you can see in this case we are not using any Electron specific types for any of the parts. But if we were implementing this the way you normally see the abstract factory used, then `sl_engine` and `vac_engine` may have looked something like this:

```
function sl_engine(::ElectronFactory)
    Rutherford(25e3, 311, mass = 35)
end

function vac_engine(::ElectronFactory)
    RutherfordVac(26e3, 343, mass = 35)
end
```

Implementing a `FalconFactory` would require similar code, but how can somebody implementing a new factory know they are doing it right? We could start by defining our new rocket factory and look at the error messages we get when we try to use it.

```
struct FalconFactory <: RocketFactory end
```

Let us create an instance of this factory and attempt to create an engine.

```
julia> sl_engine(factory)
ERROR: MethodError: no method matching sl_engine(::FalconFactory)
Closest candidates are:
    sl_engine(::ElectronFactory)
```

A developer could then lookup documentation to get clues what to do:

```
help?> sl_engine
search: sl_engine

    sl_engine(factory::RocketFactory) -> Engine
```

Produce a sea level rocket engine. Concrete factories must implement this method.

This informs developers that they need to extend the `sl_engine` function with a method for their factory type which return an `Engine` object.

What about trying to assemble a whole rocket?

```
julia> assemble_rocket(factory)
ERROR: MethodError: no method matching booster(::FalconFactory)
Closest candidates are:
  booster(::ElectronFactory)

help?> booster
search: booster booster_tank sideboosters detach_sideboosters!

    booster(factory::RocketFactory) -> Rocket
```

Have the rocket factory produce a booster stage. Concrete factories must implement this

This tells developers that they need to implement the `booster` method for their factory and return a `Rocket` object.

This process can be cumbersome to repeat so ideally you provide an overview in your documentation about all the methods which need to be implemented and which ones are optional.

## The Observer Pattern

This is a famous pattern as it is part of the Model-View-Controller composite pattern. In this pattern we deal with a subject object which is modified and we want one or more observers to be informed of changes to the subject.

This pattern is also sometimes referred to as publish-subscribe. You got a publishing object publishing change events and a subscriber which subscribes to these events.

I don't want to get into every detail of how this is done. This is mainly about giving you some inspiration about how you can take object-oriented thinking and adapt to the world of Julia.

There are many ways of doing this depending on the complexity of your needs. Let us look at a very simple example. Imagine a library/package supplying a `Point` type:

```
mutable struct Point
    x::Int
```

```
y::Int
end
```

Because it is from an external package, you cannot modify this code, however you may still want to be informed about say modifications to the `x` field.

```
import Base: setproperty!

function setproperty!(p::Point, key::Symbol, x)
    if key == :x
        println("x was changed from $(p.x) to $x")
    end
    setfield!(p, key, x)
end
```

With this code we are informed about anything that changes `x` but not something changing `y`.

```
julia> p = Point(3, 4)
Point(3, 4)

julia> p.x = 10
x was changed from 3 to 10
10

julia> p.x += 1
x was changed from 10 to 11
11
```

Notice even when we use `+=` it catches modifications to this field. However changing `y` or just reading the value of `x` doesn't cause any message to be printed.

```
julia> p.y = 6
6

julia> p.x
11
```

How about observing changes to an array, such as elements being changed or added? One problem with implementing methods such as `setproperty!` is that they may already be implemented. For an array `setindex!` is already implemented and we don't want to alter it.

In this case it really helps that Julia code almost always tend to work with abstract interfaces rather than concrete types. For instance a function dealing with strings will take `AbstractString` as input not `String`.

We make a `VectorSubject` to track changes to vector object (1D array). We need to allow this subject to be a drop in replacement for the vector we are observing. Thus we import some functions to give `VectorSubject` a vector interface.

```
import Base: getindex, setindex!, show, iterate, size

mutable struct VectorSubject{T} <: AbstractVector{T}
    collection::AbstractVector{T}
end
```

Don't worry too much about the template argument `T` as that will be discussed more in detail in later chapters. It indicates the type of each element in the collection. `T` is just a placeholder. Any place with a `T` will get the same element type.

We implement these methods as simple forwarding operations to the collection we are wrapping.

```
function getindex(subject::VectorSubject, key)
    getindex(subject.collection, key)
end

function show(io::IO, subject::VectorSubject)
    show(io, subject.collection)
end

function iterate(subject::VectorSubject, state)
    iterate(subject.collection, state)
end

function iterate(subject::VectorSubject)
    iterate(subject.collection)
end

function size(subject::VectorSubject)
    size(subject.collection)
end
```

The interesting case is the `setindex!` method as this is where we add our observation.

```
function setindex!(subject::VectorSubject, value, key)
    println("Changing element at $key to $value")
    setindex!(subject.collection, value, key)
end
```

While this may seem like a lot of code to do observation, these kind of subjects can easily be reused. In fact we can easily defined subject wrappers for multiple types. If I wanted this to work for a dictionary as well I could implement the previous methods using Union types instead:

```
mutable struct DictSubject{Key, Value} <: AbstractDict{Key, Value}
    collection::AbstractDict{Key, Value}
end

function setindex!(subject::Union{VectorSubject, DictSubject}, value, key)
    println("Changing element at $key to $value")
```

```
    setindex!(subject.collection, value, key)
end
```

Below you can see that we can wrap a `VectorSubject` around a regular array and use it much like a normal array. The difference is that when we change the value at an index this gets reported.

```
julia> array = [2, 3]

julia> subject = VectorSubject(array)
2-element VectorSubject{Int64}:
 2
 3

julia> subject[2]
3

julia> subject[2] = 4
Changing element at 2 to 4
4

julia> map(x->x+10, subject)
2-element Array{Int64,1}:
 12
 14
```

## Multiple Observers

In the classic observer pattern we have multiple observers, meaning many subscribers or observers can indicate that they want to observe a change in a subject. In our current solution there is only *one* observer. However it is trivial to modify our solution to support multiple observers.

```
mutable struct VectorSubject{T} <: AbstractVector{T}
    collection::AbstractVector{T}
    observers::Vector
end
```

We have added a field `observers` to contain a list of objects, which want to observe changes. In addition we need a function to allow us to add observers. Notice how we put the observer first. That is to be able to utilize the `do ... end` syntax in Julia.

```
function add_observer!(observer, subject::VectorSubject)
    push!(subject.observers, observer)
end
```

Finally we need a modification of the method we want to catch changes in.

```
function setindex!(subject::Union{VectorSubject, DictSubject}, value, key)
    for observer in subject.observers
        observer(subject.collection[key], value)
    end
```

```
    setindex!(subject.collection, value, key)
end
```

If you want to test this out you could write code like this:

```
array = [2, 3]
subject = VectorSubject(array)

add_observer!(subject) do old, newvalue
    println("Changed from $old to $newvalue")
end

add_observer!(subject) do old, newvalue
    println("Value changed with $(newvalue - old)")
end
```

If you ran this in the Julia REPL, you would get the following results:

```
julia> subject[1] = 3
Changed from 2 to 3
Value changed with 1
3

julia> subject[2] = 4
Changed from 3 to 4
Value changed with 1
4
```

Observers don't have to be functions. They could also be objects utilizing callable objects. Here we create a type `Observer` which can be used as a callable object.

```
struct Observer
    name::String
end

function (observer::Observer)(old, newvalue)
    println("$observer.name observed $old -> $newvalue")
end
```

We can then create an instance of this observer and add it as an observer:

```
julia> observe = Observer("Curiosity")
```

```
Observer("Curiosity")
```

```
julia> add_observer!(observe, subject)
3-element Array{Any,1}:
 #5 (generic function with 1 method)
 #7 (generic function with 1 method)
 Observer("curiosity")
```

```
julia> subject[2] = 33
Changed from 4 to 33
Value changed with 29
```

```
Curiosity observed 4 -> 33
33
```

## Design Patterns in Julia

One of the key objectives of this chapter was to show not only how you can adapt object-oriented thinking to Julia, but also how typical object-oriented pattern thinking is less valuable in Julia. The flexibility of the language makes a lot of software engineering problems straight forward to solve. You don't need to study a large catalog of object-oriented design patterns to figure out how to structure your code.

Instead you get very far in Julia by getting into the habit of targeting abstract interfaces rather than concrete types. For instance if a function `f` can operate on any integer value, don't implement it with the signature `f(x::Int64)` just because you happen to use `Int64` numbers most of the time. Instead write it as `f(x::Integer)`.

Remember there is no performance penalty for this in Julia, unlike most statically typed languages. The Julia JIT will create specialized versions of your function for different types and store the machine code for these specializations in memory. In fact you could just write `f(x)` if you wanted to and there would be no performance difference.

The main reason to specify types in Julia is to communicate what sort of type we expect to the reader of the code. It is also to be able to provide different code, in cases where the type matters for the implementation.

The lack of focus on patterns in the Julia community is not very different from the lack of focus in other dynamic language communities such as Python, Ruby, Lua and JavaScript. Pattern oriented thinking is more dominant among statically typed object-oriented languages such as C++, Objective-C, Java and C#. In strongly and statically typed languages more careful thought has to be put into types and their relations.

Good support for functional programming also helps simplify many challenges. As we saw in the observer example, we only needed to provide something that acted as a function taking two arguments. This could be a function or a callable object.

In classic Java, you would have to call a specifically named method on a specifically named interface. The fact that functions and types are first class objects in Julia makes a lot of problems go away.



# Code Organization

- **Modules.** A way of creating *namespaces* for your types and functions.
- **Namespaces.** What is it and how does it relate to functions, modules and environments?
- **Packages.** Handling packaging, formatting and distribution of modules to other users.
- **Environments.** Dependencies and versioning for packages.
- **Module Paths.** Where does Julia look for modules?
- **Standard Modules.** The modules Julia comes bundled with.

## Motivation

Before getting into the details let us take a birds-eye view and talk about why we need modules, packages and environments. All these concepts are essentially ways of dealing with increasing software complexity.

Before continuing it is worth clarifying some potential misconceptions. The terminology we use in Julia is *different* from many other languages.

- **Modules** are called *namespaces* in C++ and *packages* in Java and Go.
- **Packages** in Julia are not the same as packages in Java, as those are Julia *modules*. In Julia what we mean by package is how something is packaged and organized for distribution. The Java equivalent would be a JAR file<sup>27</sup>. In C/C++ a package would correspond to a static library or DLL<sup>28</sup>. In Objective-C and Swift it would be the same as a Framework<sup>29</sup>.
- **Environments** are called *virtual environments* in Python but are quite similar to what others would call sandboxes or containers. Docker creates something similar to environments.

As the size and complexity of software developed we had to develop new concepts. Initially we used *functions* to manage larger programs. When that was not enough we had to start putting functions into *modules*. Later software projects got even larger and one started to use code from teams in different

---

<sup>27</sup>A JAR (Java ARchive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution.

<sup>28</sup>DLL is a dynamic link library file. Meaning it is a file containing machine code which gets loaded into memory when a program depending on it is run.

<sup>29</sup>Framework can mean many things in software development, but in the Objective-C and Swift world it refers to a directory which contain machine code for a dynamically linked library, related header files and other resources.

locations. A need for a way to distribute and install modules arrived and we developed the concept of *packages*.

Packages quickly needed some way of dealing with versions. One team would make changes to their package causing the code to be incompatible with the code written by another team using that package.

The complexity of dealing with software made up of multiple parts where some parts depended on different versions of the same package eventually forced the creation of environments to bring order to the chaos.

## Modules

What is the purpose of a module?

Modules provide a namespace for your functions and types.

However this answer is not very enlightening, if you only have a vague or non-existent idea of what a namespace actually is. Namespaces as a concept pops up in many different contexts within computer science. A filesystem, for instance, is thought of as a namespace in particular on Unix like systems such as macOS and Linux.

In the beginning a floppy disk or hard drive would contain a flat list of files. However as the number of files grew one would get name collisions, meaning two or more different files where given the same name.

The initial solution to this was taken from the realm of filing cabinets. Files are not dumped into one stack in a physical filing cabinet. Instead files are organized by folders. Each folder provides a namespace for the files within. With folder two files with identical names can exist as long as they are placed in separate folders.

Of course it did not take long before even this structure was found to be too limiting. Eventually operating systems got nested folders. Hence a file could now be references by an arbitrary long file path.

Above you can see some files organized into different directories (folders). Now why am I talking about filesystems, should I not be talking about Julia modules? The reason is that the way we speak about files and folders has many similarities to Julia modules.

Both with files and modules we deal with different types of paths. For a filesystem we have **fully qualified paths** such as:

```
animals/cat/food  
animals/dog/food
```

We could write only the name of a file, but that requires the file to be accessible in the system's search path. A search path is a list of directories maintained by the operating system, which should be searched to locate a file without a fully qualified path.

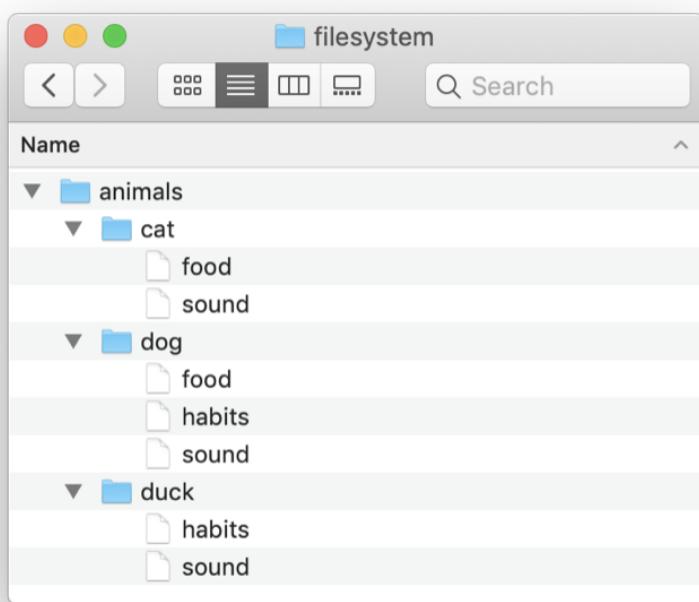


Figure 56: Example of a filesystem with folders in blue and files in white

```
food
```

Both file systems and modules have the concept of a current module or current working directory. You can specify paths relative to this directory. Below is an example of accessing the food file found inside the current working directory.

```
./food
```

This idea extends to more complex **relative paths**. Two dots .. means to enter the parent directory of our current directory. In this case we look into the parent directory for a directory called dog. Basically this gives us a sibling directory. So in our current example, if you are in the duck or cat directory, but want to access the food file in the dog directory you can do this.

```
../dog/food
```

## Functions as namespaces

Namespaces however exist in all sorts of circumstances without us as developers necessarily thinking much about it. Functions also form namespaces. Consider the code example below. In both functions we find the variable named sound. Does this variable refer to the same variable? Of course not. The two variables exist in different namespaces defined by each function.

In programming speak we refer to these kinds of namespaces as *scopes*, and they affect the visibility of variables. In this example sound is in a local scope defined by a function.

```
function dog()
    sound = BarkingSound()
end

function duck()
    sound = QuackingSound()
end
```

However if you had prefixed the variable name with `global`, both references to the variable would have referred to exactly the same variable.

```
function dog()
    global sound = BarkingSound()
end

function duck()
    global sound = QuackingSound()
end
```

## Julia Module Paths vs Filesystem Paths

Let us compare module paths with file paths. When you type a command to execute in a Unix shell, for instance `food`, then the operating system will search through all directory path locations listed in the `PATH` environment variable.

The equivalent for Julia modules is searching through the directory path locations listed in the `LOAD_PATH` global variable (it is not an environment variable. The shell does not know about it).

Filesystem Paths		Julia Module Paths	
animals/cat/food	full qualified path	Animals.Cat.food	absolute module path
animals/dog/food		Animals.Dog.food	
food	must be in search PATH	Dog	in module <code>LOAD_PATH</code>
./food	local directory	.Dog	relative module path
../dog/food	refer to dog food while in cat directory	..Dog	refer to Dog module while in Cat module

Figure 57: Module paths

If some code inside the `Animals` module want to refer to the `Dog` submodule, you have to write `.Dog`. However if code inside the `Cat` module wants to refer to the `Dog` module you would have to write `..Dog`. We will go through some concrete examples of this later, but we are trying to take a birds-eye view at the moment.

## Modularization in Julia

Modularization refer to the idea of organizing code into manageable chunks. It is important to realize that modules is just one of many ways of doing this.

### Functions

Putting code into separate functions is perhaps the simplest and the most important form of modularization in any language. This is often forgotten. Many developers will write what may be called scripts. That means their code is composed of hundreds, if not thousands, of lines of code within a source code file without any further subdivisions.

That is bad from a software engineering perspective, but interestingly it is also bad for *performance* in Julia. This will be counterintuitive for people coming from other languages, especially older developers as historically there was an overhead in calling functions. Calling lots of small functions would thus reduce performance. For Julia it is the *opposite*. Lots of tiny functions improve performance, because it simplifies the job of the Julia just-in-time compiler which is able to produce more optimized code.

Hence my personal preference is to try to keep function sizes below 20 lines of code. With such an expressive language as Julia, that is seldom hard to achieve. A lot of my functions are usually 1-10 lines of code. One line functions are so easy to write in Julia, that you will find yourself using them more frequently than in many other languages.

## Files and Folders

The level above functions in modularizing your Julia code is files and folders. There are many ways in which you can organize your code into files. If you think in more object-oriented terms, you may want to place a type together with the functions that operate on that type. However in Julia since functions are not bundled with their types syntactically the way methods in an object-oriented language are, you can chose other types of organizations if you want to.

For instance you can organize code into functional areas. So that each file may contain functions operating on different types but performing similar operations. Files can further be organized into subfolders, so that each folder contains files implementing related functionality.

## Modules and Submodules

A module would normally containing functionality spread over multiple files. A submodule may be used to represent functionality you have placed in a subfolder. Files and folders are different from modules and submodules in that they don't create any namespace. That is different from e.g. Python where each file represents a module. Julia is more similar to C++ in this case. In C++ a module is literally referred to as a namespace, and it exists independent of files. A C++ namespace can span multiple files like a Julia module.

Modules and submodules gives a perspective on the code for the user of that code. The user does not care about what file the code is placed in. The user only cares about what modules the types and functions are in. However as a developer of the module you care about files and directories because they help you organize your work. The organization which is practical for you as a developer is different from what is practical for the users of your code.

So just like directories in the filesystem, modules can be nested. This allows you to have the same function and type name used multiple times for different things in the same module. You only need to make sure they are placed in separate submodules.

Let us look at some actual examples of modules.

## Rocket Equations Module

In a file called `physics-module.jl` we place the code shown below. We call the module `Fysikk`, which is the Norwegian word for physics.

```
module Fysikk

const global g₀ = 9.80665

exhaust_velocity(Isp) = Isp * g₀
delta_velocity(vᵕ, m₀, mf) = vᵕ * log(m₀/mf)
rocket_thrust(Isp, mass_flow) = g₀ * Isp * mass_flow
mass_flow(thrust, Isp) = thrust / (Isp * g₀)

function burn_length(Δv, m₀, thrust, vᵕ)
```

```

(1 - exp(-Δv/ve))*m0*ve/thrust
end

function propellant_consumption(Δv, m0, ve)
    m0 = m0/exp(Δv/ve)
end

end # Fysikk

```

The beginning of the module starts with the keyword `module` and `end` marks the end of the module. This module contains implementation of some common equations for rocketry.

We can try to use this module by starting Julia with `julia -i physics-module.jl` this has the same effect as if you had started Julia and then copy pasted all the code in the file into the Julia REPL environment.

If we try to use this module in the REPL however we get some problems.

```
julia> using Fysikk
ERROR: ArgumentError: Package Fysikk not found in current path:
- Run `import Pkg; Pkg.add("Fysikk")` to install the Fysikk package.
```

Julia cannot find out package. The reason for that requires understanding how Julia locates packages which we will not cover just yet. The Solution is to use a relative package reference:

```
julia> using .Fysikk
```

Let us try out calling the `exhaust_velocity` function which calculates the velocity of propellant ejected from a rocket engine.

```
julia> ve = exhaust_velocity(282)
ERROR: UndefVarError: exhaust_velocity not defined
```

```
julia> ve = Fysikk.exhaust_velocity(282)
2765.4753
```

We got some hiccups there as well. Using other modules you are used to being able to just write the function name, without prefixing it with the module name. So why is it different in this case?

The reason is that in Julia you have to specify which *symbols* in the module are exported.

```
module Fysikk

export g0,
       exhaust_velocity, delta_velocity,
       rocket_thrust, mass_flow,
       burn_length, propellant_consumption

const global g0 = 9.80665

exhaust_velocity(Isp)           = Isp * g0
```

```

delta_velocity(v_e, m_0, mf)      = v_e*log(m_0/mf)
rocket_thrust(Isp, mass_flow)    = g_0 * Isp * mass_flow
mass_flow(thrust, Isp)           = thrust / (Isp * g_0)

function burn_length(Δv, m_0, thrust, v_e)
    (1 - exp(-Δv/v_e))*m_0*v_e/thrust
end

function propellant_consumption(Δv, m_0, v_e)
    m_0 = m_0/exp(Δv/v_e)
end

end

```

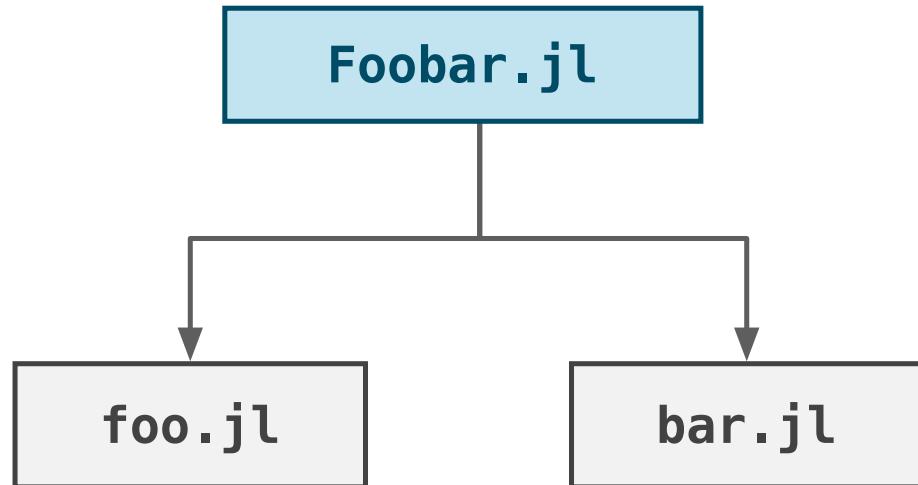
This is different from what a Java, C# or C++ developer would be used to. In those languages you prefix things you want exported with `public`. In Python you don't export explicitly at all. The purpose of `export` in Julia is to basically tag all the functions and types you want to be brought into the current namespace when you write using `Fysikk`.

Unlike languages such as C++ and Java, this is not a hard access control. As you saw in the first example we could write `Fysikk.exhaust_velocity(282)` to access a function which had not been exported.

## Hierarchical Organization of a Module

The `Fysikk` module we made is just a toy module. Real modules will be made up of multiple files. Larger modules will even contain multiple submodules. Conceptually a module will look like the diagram below.

Below is an example of a minimal module `Foobar` which is just made up of two files `foo.jl` and `bar.jl`. Capitalized `Foobar.jl` will define the module of the same name. It will seldom contain any code itself. Instead the code will be distributed across one or more files with lowercase names.



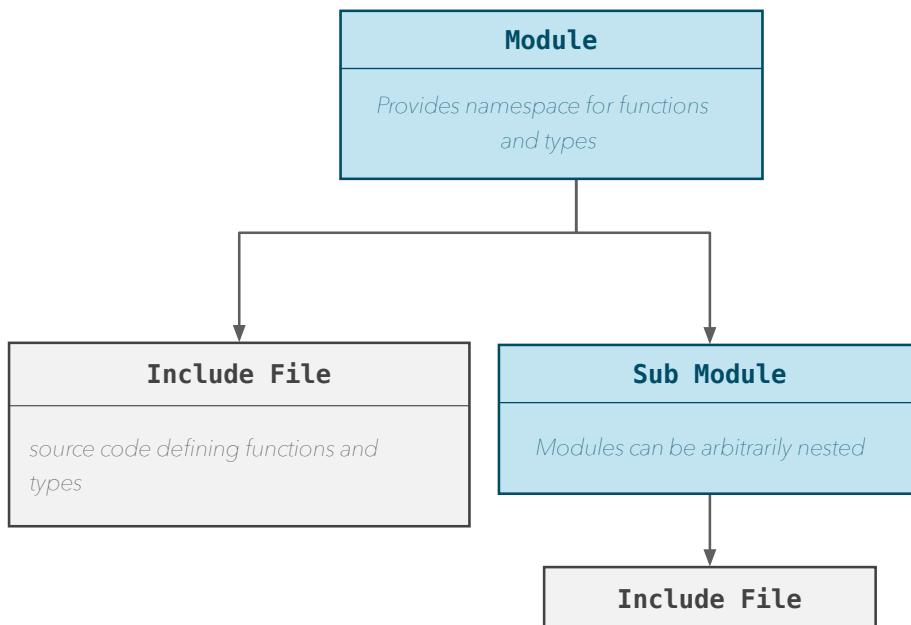


Figure 58: Module hierarchy

A screenshot of a code editor window titled "Foobar.jl — Foobar". The left pane shows the following Julia code:

```

1 module Foobar
2
3     include("foo.jl")
4     include("bar.jl")
5
6 end
7

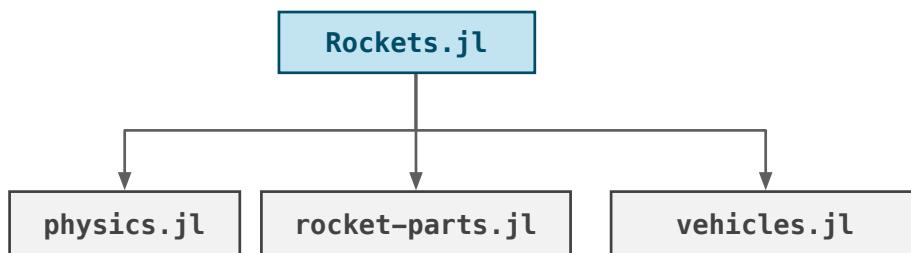
```

The right pane shows a file tree:

- Foobar
  - src
    - bar.jl
    - foo.jl
    - Foobar.jl
  - Project.toml

At the bottom of the window, there are various status icons and tabs.

Foobar is an absolutely minimal and unrealistic module. It does not give a good idea of how a real module would be split into submodules and individual files. So let's look at a more realistic module called Rockets.



```

1  """
2  A module for doing simple Newton style physics related to the movement
3  of bodies. In particular it was made to do simple rocket equations.
4  """
5  module Rockets
6
7  include("geometry/Geometri.jl")
8  include("physics/Fysikk.jl")
9
10 include("transform.jl")
11
12 include("rocket-parts.jl")
13 include("vehicles.jl")
14
15 include("properties.jl")
16 include("actions.jl")
17
18 include("iteration.jl")
19 include("parts-loading.jl")
20
21 include("display.jl")
22 include("plotting.jl")
23 include("simulate.jl")
24
25
26 end # module
27

```

The `Rockets` module contains code we have been working through in various examples to simulate assembling a rocket. It is useful to split this into separate submodules. For instance there is a physics module called `Fysikk` which deals with Newton style equations for calculating the movements of bodies through space. And we have included the rocket equations.

However nothing in the physics module deals with how we have chosen to represent a multi-stage rocket in code. We put it in a separate module because it is generic enough that one could imagine distributing it as an entirely separate module for people who want to do other things than simulate rocket launches.

When doing physics calculations to determine the positions of moving objects, we need types to represent vectors and points. Matrices are used to calculate rotation of objects. Geometry is separate from physics and can be used in contexts not involving physics at all. Hence we put all the geometry files in a separate directory and make them part of a geometry module called `Geometri` (Norwegian for geometry).

Julia does not require us to put files related to a module in a separate directory. Files and directories are completely separate from the modules concept. You as a developer organize it in a way that makes your job of working with the code easier.

Notice we are frequently modularizing based on functionality rather than on types. By that we mean related functionality across multiple types is clustered together in a file.

The geometry library is organized much like an object-oriented program would be organized. The `Circle` type is in a separate file along with functions which perform circle related operations such as:

- Calculating the area of a circle.
- Bounding box of a circle.
- Intersection between circles and other shapes.

```

1  """
2  A module for doing simple Newton style physics related to the movement
3  of bodies. In particular it was made to do simple rocket equations.
4  """
5 module Rockets
6
7  include("geometry/Geometri.jl")
8  include("physics/Fysikk.jl")
9
10 include("transform.jl")
11
12 include("rocket-parts.jl")
13 include("vehicles.jl")
14
15 include("properties.jl")
16 include("actions.jl")
17
18 include("iteration.jl")
19 include("parts-loading.jl")
20
21 include("display.jl")
22 include("plotting.jl")
23 include("simulate.jl")
24
25
26 end # module
27

```

Figure 59: Files included in the `Rockets` module

However the multi-stage rocket is not organized like this. `parts-loading.jl` does not contain any new types. Instead it contains functions related to loading different rocket parts such as propellant tanks and rocket engines.

`display.jl` contains implementation of `show` functions for `Rocket`, `Tank`, `Engine` and other types.

There is no correct way of organizing your Julia code. But this gives you an example of the flexibility Julia offers you in organizing your code in a way that makes sense to you.

For instance the visualization of a whole `SpaceVehicle` involved the visualization of multiple related types. That meant developing multiple `show` functions in parallel. It would have been harder if we had to switch between 4-5 different files to change a `show` function related to all the different types involved.

Likewise when implementing loading of propellant tanks and rocket engines, I jumped between both load functions while doing the development. Putting functions for these separate types in different files would just have slowed down the process.

## Common Misconceptions About Julia Modules

Depending on what your previous programming language background is, there are some aspects of Julia modules which can be confusing, so let us go over the same material we just covered by comparing with other programming languages to highlight how Julia handles modules differently.

## Java Users

To a Java developer it would be natural to define modules using the approach shown below. In every source code file the code is encapsulated in the module the code belongs to.

In the `constants.jl` file you would put this code:

```
module Fysikk

export g₀

const global g₀ = 9.80665

end
```

Then in the `velocities.jl` file you would put:

```
module Fysikk

export exhaust_velocity

exhaust_velocity(Isp) = Isp * g₀

end
```

A Java developer may think this makes the constant and velocity function a part of the `Fysikk` module.

### IMPORTANT

This is **wrong**. You *cannot* repeat a module definition in multiple files in Julia. A module can only be defined **once**. Seen from Julia's perspective you are trying to redefine the `Fysikk` module not add to it. *Don't do this!*

Instead the correct way of doing this in Julia is shown below. In `constants.jl` you put this code:

```
export g

const global g = 9.80665
```

In `velocities.jl` you put this code:

```
export exhaust_velocity

exhaust_velocity(Isp) = Isp * g
```

Neither file *knows* what module they are part of. In fact a file can be part of *many* modules. We define the `Fysikk` module only *once*.

```
module Fysikk

include("constants.jl")
```

```
include("velocities.jl")
end
```

While this system may seem more complex to deal with than the Java style approach, it gives a lot of flexibility to the programmer. For instance you can create an entirely different module for doing physics calculations on the moon, which reuses some of your existing code.

```
module MoonPhysics
    export g

    const global g = 1.62
    include("velocities.jl")

end
```

Because `exhaust_velocity` is implemented by referring to the variable `g` and we changed it to contain the value for the acceleration of gravity on the moon, our `exhaust_velocity` would give a different result. Although if you know common rocket equations, you would immediately see that this is a stupid example, given that the relationship between exhaust velocity and specific impulse (`Isp`) is *always* depended on the `g` on earth.

### C/C++ Users

Superficially the way Julia handles inclusion of files has some resemblance to C/C++, thus it is easy to get the false impression that the same thing happens. Let us clarify this a bit. In Julia `include` is a *function call* which gets evaluated. Thus `include` is being executed at runtime. It causes the evaluation of the code contained within the `include` file. If this code is being evaluated inside the scope of a module, then any function or global defined within, will be part of the namespace that module defines.

```
// constants.h
static const g_0;

// velocities.h
double exhaust_velocity(double Isp);

// Fysikk.h
namespace Fysikk {

#include("constants.h")
#include("velocities.h")

}
```

In C/C++ in contrast, `#include` is something that happens before the program gets compiled. It basically pastes the code into the file. Thus we get a new file where that code exists which later gets compiled by the C/C++ compiler.

We can illustrate how `include` is just a function in Julia with this simple example. Create a file named `hello.jl` with the contents:

```
"hello world"
```

Syntax-wise this is just a Julia string expression. If evaluated it should produce a string object.

```
julia> s = include("hello.jl")
"hello world"
```

```
julia> println(s)
hello world
```

As you can see, `include` is called like any other function and its return value is whatever got last evaluated in the ```hello.jl`'' file. This is very different from a C/C++ `#include` preprocessor directive. You don't run a `#include` like a function call. It does not have a return value, because it is used for substituting text into the C/C++ source code. In fact it is not even part of the C/C++ language.

## Implicit Modules in Julia

Whenever you are using Julia there are a number of modules which are automatically imported into your Julia namespace and available for use without being explicitly imported.

Why should you care about this? Because it helps you understand what is going on in the REPL when you define new modules.

1. **Core** module contains what we may consider as being ``built into'' Julia. Julia simply would not work without this functionality.
2. **Base** is basic functionality which you want to use most of the time such as arrays, dictionaries, IO, time etc.
3. **Main** is an empty module. It is the top-level and current module when Julia runs.

This helps you understand why we had to write `using .Fysikk` previously. Remember writing the following code to use the `Fysikk` module?

```
$ julia -i physics-module.jl
julia> using .Fysikk
```

This is functionally equivalent to writing:

```
julia> include("physics-module.jl")
julia> using .Fysikk
```

Because the module was created while we were in the `Main` module `Fysikk` ends up being a submodule of `Main` and not a top-level module.

Hence what we wrote could be implicitly seen as having somehow managed to write:

```
module Main

julia> include("physics-module.jl")
```

```
julia> using .Fysikk
```

```
end # module Main
```

Hence this is the analogous to writing a Julia source code file containing:

```
module Main
```

```
module Fysikk
```

```
...
```

```
end
```

```
end # module Main
```

This is why you need to write `using .Fysikk` and not `using Fysikk`. By using the `-i` switch or `include` we turn `Fysikk` into a submodule of `Main` and since all code you write in the REPL run in the context of `Main` you have to use a dot-prefix to indicate that you are referring to a local module.

This obviously seems cumbersome. Many people mistakenly think this is how modules are loaded in Julia, that you first have to include the file containing the module and then run `using Foobar` to load the `Foobar` module or whatever your module is named.

When you write `using Foobar`, Julia has an advance system to locate the module and load it. However understanding this system requires getting into how packages and environments work.

## Module Paths and Nested Modules

We can look at nesting of modules and module paths more in detail by looking at a variation of our `Rockets` module. Below you see skeleton code defining the top-level module as well as nested submodules.

Imagine that in the `Rockets` module we have functions which need to access `Physics` and `Motion` module functionality.

```
module Rockets
    module Physics
        module Motion
            ...
        end
    end

    module Geometry
        using ..Physics.Motion
        ...
    end

    using .Physics
end
```

If you are in e.g. the Julia REPL environment or in another Julia package and want to access these modules you would write these module paths:

```
using Rockets
using Rockets.Physics
using Rockets.Physics.Motion
```

## Importing Modules

Thus far you have seen different ways of importing a module in Julia, but we have not said much about the principle differences. E.g. what is the difference between `using Foobar` and `import Foobar`?

The Julia standard manual gives a full overview but this can easily be overwhelming and in practice you will only use a tiny subset of all the possible forms of import.

Besides all forms of import manipulate only two aspects of your Julia runtime environment:

- What functions and types are **brought into scope**, or put in another way: which functions, types and global variables become part of the global namespace.
- What functions are made **available for extension**. Remember if you want to add methods to a function defined in another module, you have to import that module in a particular way.

Julia ends up being more complex than what you may be used to from languages such as Python when dealing with modules. The reason for this is that in Python you don't have to deal with the issue of extending existing functions by adding methods. Extensions in Python involve extending a base class by adding a sub-class. That is done in such an explicit fashion that there is no need to protect against accidentally extending a class you did not intend to extend.

Python modules are also much simpler in terms of what is exported. There is no explicit listing of exported symbols. In Python everything is imported from a module when you use it or only the functions and types you explicitly list as imported.

Let us talk through some different ways of importing by looking at this example module:

```
module Foobar

    export foo, bar

    foo() = "foo"
    bar() = "bar"
    qux() = "qux"

end
```

We are once again using common nonsense words such as ``foo'', ``bar'' and ``qux'' to easily distinguish them from the important keywords and syntax used.

### Using Foobar

You can think of `using` as stating that you want to actually use functions and types inside the `Foobar` module, rather than extending them.

```
using Foobar
```

This will bring `foo` and `bar` into scope because they are exported. However I can still access `qux` by writing a fully qualified path `Foobar.qux`.

```
julia> foo()
"foo"
```

```
julia> qux()
ERROR: UndefVarError: qux not defined
```

```
julia> Foobar.qux()
"qux"
```

The Python equivalent would be `from Foobar import *`, except it is not exactly the same since in Python you don't list symbols to export.

---

```
using Foobar: foo, qux
```

We only bring the `foo` and `qux` functions into scope. One problem with this approach, which is why I don't recommend it, is that it makes no distinction between exported and non-exported functions.

```
julia> using Foobar: foo, qux
```

```
julia> foo()
"foo"
```

```
julia> qux()
"qux"
```

Unless the developer reads the import statement properly, he or she may not realize when seeing a call to `qux` that it was meant to be a private or non-exported function.

Also note how it is quite impractical as you cannot refer to exported function `bar` with a fully qualified path using this method, as the `Foobar` symbol has not been put into scope.

```
julia> Foobar.bar()
ERROR: UndefVarError: Foobar not defined
```

This approach mimics very closely the common way for Python developers to write their code. It is the Python equivalent of writing `from Foobar import foo, qux`.

In the Python community this is strongly recommended, because importing the same symbol from two different modules in Python will cause the last imported

symbol definition to overwrite all prior definitions without any warning or error.

In Julia this is not a problem. If you import `foo` from two different modules `A` and `B` which export `foo` then Julia will force you to write `A.foo()` or `B.foo()`. Just writing `foo()` would cause a runtime exception.

```
WARNING: both A and B export "foo"; uses of it in module Main must be qualified
ERROR: UndefVarError: foo not defined
```

When importing modules with `using` you can still extend functions with methods but it requires fully qualified names.

```
using Foobar
```

```
"Repeat foo `n` times"
Foobar.foo(n::Integer) = "foo"^n
```

Some Julia developers prefer this approach as it makes it clear where the function being extended originates. This makes sense from an object-oriented thinking. You would want to list the class you subclass. However often thinking that you are purposefully extending an existing function is misleading. Often you are merely trying to avoid a name collision.

### Import Foobar

The Julia `import` statement does not know anything about exported symbols. None of the variations of `import` takes exported symbols into account.

In this regard `import` is closer in behavior to that of Python's `import` statement.

However the primary way of using `import` in this book is for extending functions defined in other modules such as arithmetic operators defined in the `Base` module.

```
import Foobar: foo, qux
```

This is how we import `foo` and `qux` so they can be extended. However it also puts `foo` and `qux` into scope, so you can easily call the functions.

```
import Foobar
```

This is a form I seldom use, but which is useful if you use a package where you want to always write fully qualified names such as `Foobar.foo()` and `Foobar.bar()`.

When is this useful? When dealing with special packages using familiar sounding function names this is useful. For instance the `Pkg` module has an `add` method. It is easy to mistake that for a function adding an element to a collection.

Since these operations are conceptually so different, we want to use fully qualified names for calling such functions, such as `Pkg.add("Statistics")` otherwise the code can become hard to read.

## Standard Library

While `Core` and `Base` are always loaded in Julia, Julia is shipped with a larger collection of modules as part of its standard library. You don't have to download anything extra from the internet e.g. to be able to do statistics or socket communications, both are provided by modules bundled with Julia such as `Statistics` and `Sockets`.

The official Julia manual gives a full overview and documentation of the whole standard library and the modules it contains.

We will not cover all of that here but instead talk about where you can find functionality you will frequently need.

- **Dates**. Dealing with date formats, time intervals and converting between time given as seconds since e.g. 1970 to a days, hours and minutes is common in programming.
- **Random** for generating random numbers. We have previously used the `rand()` function.
- **DelimitedFiles** is useful for dealing with CSV (comma separated values) files or files where values are separated by other characters such as tabs. However this module is bare bones. Julia developers will usually use external libraries such as `CSV.jl` and `DataFrames.jl` to deal with with `.csv` files.
- **Statistics** for doing basic statistics like finding the mean or median, computing covariance etc.
- **Logging** Contains macros for logging. The benefit of macros is that similar to the C/C++ preprocessor macros, if they are turned off, there is no performance penalty of having logging code spread around your code.
- **Test** Julia comes with a unit testing library used by basically every Julia package distributed online. If you write a module you want to share with others and collaborate on, you better incorporate unit-tests.

## Packages and Environments

Packages is about *how* you bundle up and distribute modules and how to deal with dependencies and versioning. Modules exist at the language level. There is specific Julia syntax to define something as belonging to a module and expressing what module to use. This is similar to how C++ has language constructs for defining namespaces and Java has language constructs for defining Java packages. To not get confused, keep in mind that a Java package in Julia terminology is a module.

However in all these different languages, how modules are delivered and included into your project is outside the language. The C++ language does not say anything about how dynamic or static libraries should be linked. Nor does it say anything about how dependencies and versions are expressed. That is handled by other tools. The compiler writers define how their implementation

of C++ creates a shared library. How the different pieces are managed is handled by a build system or package management tool such as make, CMake<sup>30</sup>, Scons<sup>31</sup> etc.

Likewise Java as a language does not know about JAR files or tools such as Ant<sup>32</sup> and Maven<sup>33</sup> used for building a project and/or fetching JAR files representing packages.

Thus when you write `using Foobar` in your Julia program, this module could in principle be provided by a large number of different packages existing in different locations and having different versions. Modules is a language level construct. They don't know anything about versions. Packages know their versions and dependencies. Packages contain source code which defines modules.

So the package `Foobar` will contain code defining a module named `Foobar`. But there may be many such packages. Hence you need to use the package manager to define what specific package and what version of that package you want to load when `using Foobar` is executed in your code.

It is the *environment* you code lives in which determines what `using Foobar` means in term of package version and location.

## A Simple Package

To make this clearer let us create a tiny Julia package. We are going to call it `Foobar`. In your Julia REPL you have to hit the `]` key to go into package mode.

```
(v1.5) pkg> generate Foobar
Generating project Foobar:
  Foobar/Project.toml
  Foobar/src/Foobar.jl
```

This will create the follow directory structure on your hard drive:

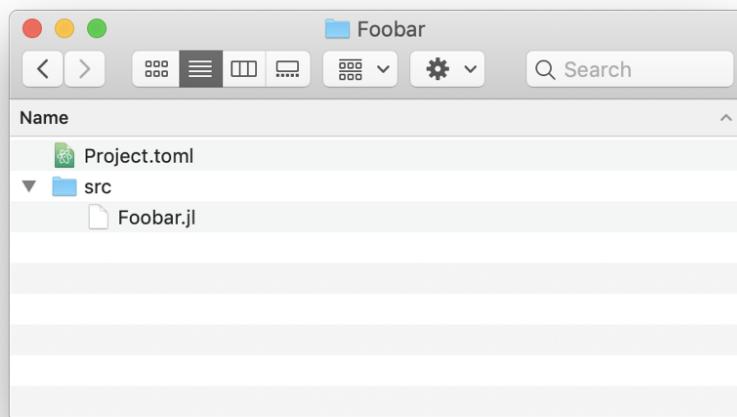
---

<sup>30</sup>A build system primarily for C++ projects. Handles dependencies between source code files, so a compiler can figure out the correct order of compilation.

<sup>31</sup>A popular build system, known for using Python to describe its dependency rules.

<sup>32</sup>A build system written in Java and primarily used to build Java projects.

<sup>33</sup>Unlike ANT, Maven is a build system that provides dependency management and many other things. In this regard it is more similar to Julia's package manager.



We get a mostly empty module definition in `src/Foobar.jl`, with the exception of a function which prints ``Hello World!''

```
module Foobar

greet() = print("Hello World!")

end # module
```

#### **IMPORTANT**

What is important to notice here is that packages have a particular structure which must be observed if you try to create this structure by hand. Your source code should go under the `src` subdirectory. A module containing all the functionality of the package must be placed in `src` where the package, module name and filename has to be identical.

A description of the package goes into `Project.toml`. In my case the contents of this package would be:

```
name = "Foobar"
uuid = "f4ee2b68-ab46-4ff7-bcff-df8c3988e1c7"
authors = ["Erik Engheim <erik@sixty-north.com>"]
version = "0.1.0"
```

This describes who made the package and what the current version of the package is. The package is actually uniquely identified by its `uuid` number and not the name `Foobar`. Many packages could in principle be named `Foobar`. However the name is important because that corresponds to the module name, which is what you use in your code. In your source code you will never use a UUID. UUIDs are of interest to the package system, not at the language level.

**NOTE Universally unique identifier UUID**

A UUID is a 128 bit number usually written in hexadecimal form such as f4ee2b68-ab46-4fff-bcff-df8c3988e1c7. The basic idea is to have a way of giving a component of some sort such as a Julia package a unique identifier without relying on a central registry. UUIDs are randomly generated, but because the number is so large the chance of generating the same number twice is minuscule. This way Bob and Bernie can create a Julia package on their respective computer with a unique identifier without needing to communicate with each other.

Not all languages are like that. E.g. in the Go programming language, when you import a package you actually refer to it by its location in the source code. However this is so that Go can avoid having files like `Project.toml`. The Go creators wanted everything in the source code files.

## Locating and Using Our Package

Before we can actually use the package we need Julia to be able to find it. Previously we simply loaded the file containing the package with something like `julia -i Foobar.jl`, however that is cumbersome.

When writing using `Foobar` Julia looks in a number of locations to find this package. You are able to modify what locations this is, by editing the `~/.julia/config/startup.jl` file. This file is always run before your Julia REPL starts. You can put any Julia code in this file, which you want to run before your REPL starts. If the file does not exist you could add it yourself:

```
$ cd ~
$ mkdir -p .julia/config
$ touch .julia/config/startup.jl
```

This is the contents of my `startup.jl` file:

```
$ cat ~/.julia/config/startup.jl
push!(LOAD_PATH, "$(homedir())/Development/Julia")
```

Which means I am adding the path "`~/Development/Julia`" to global Julia array called `LOAD_PATH`. Julia's module load system will look through the directories listed in this array.

Once you have configured this you can load your custom module. If you don't want to restart Julia, just write `push!(LOAD_PATH, "$(homedir())/Development/Julia")` directly in the Julia REPL.

```
julia> using Foobar
julia> Foobar.greet()
Hello World!
```

## Adding Package Dependencies

Let us modify our `Foobar` package to use some other packages. Keep in mind I will often use the word module and package interchangeable, since the package `Foobar` provides a module named `Foobar`.

```
module Foobar

export greet

using Dates

greet() = println("what a sunny ", dayname(today()))

end
```

We have made a change to our package which requires using the `Dates` module. If we restart the Julia REPL to load our package with the new code changes, Julia will warn us about this fact.

```
julia> using Foobar
[ Info: Precompiling Foobar [f4ee2b68-ab46-4ff7-bcff-df8c3988e1c7]
└ Warning: Package Foobar does not have Dates in its dependencies:
  - If you have Foobar checked out for development and have
    added Dates as a dependency but haven't updated your primary
    environment's manifest file, try `Pkg.resolve()`.

  - Otherwise you may need to report an issue with Foobar
└ Loading Dates into Foobar from project dependency, future warnings for Foobar are suppressed.
```

```
julia> greet()
what a sunny Monday
```

While we are able to still run `greet()` this would have failed if `Dates` had not been somewhere else on the system. Thus we need to add `Dates` to our `Foobar` package. The package manager has a command called `add` for doing this. However it always add package dependencies to the currently active package.

In this context a package work as an environment. So what we do first is to make our `Foobar` package the current active environment.

```
(v1.5) pkg> activate Foobar
```

And your prompt will change to:

```
(Foobar) pkg>
```

Any package commands involving adding, removing or installing packages will apply to the current active environment. Essentially a package is its own little sandbox or virtual environment.

```
(Foobar) pkg> add Dates
Updating registry at `~/.julia/registries/General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
```

```
Updating `~/Development/Julia/Foobar/Project.toml`
[ade2ca70] + Dates
Updating `~/Development/Julia/Foobar/Manifest.toml`
[ade2ca70] + Dates
[de0858da] + Printf
[4ec0a83e] + Unicode
```

This adds the `Dates` package to the `Foobar` environment. Since the `Foobar` environment is also a package, that means the `Foobar` package now has `Dates` added as a dependency. In fact a whole bunch of other packages got added because `Dates` depends on the `Printf` and `Unicode` packages.

This causes the contents of the `Project.toml` file to change. It gets a new section called `deps`:

```
name = "Foobar"
uuid = "94035000-e9dc-11e9-3ec0-43acaba33257"
authors = ["Erik Engheim <erik@sixty-north.com>"]
version = "0.1.0"

[deps]
Dates = "ade2ca70-3891-5945-98fb-dc099432e06a"
```

This section lists *explicit* dependencies of the `Foobar` package. To deal with *indirect* dependencies, a new file `Manifest.toml` gets added to your project directory (directory containing your package).

```
[[Dates]]
deps = ["Printf"]
uuid = "ade2ca70-3891-5945-98fb-dc099432e06a"

[[Printf]]
deps = ["Unicode"]
uuid = "de0858da-6303-5e67-8744-51eddeeb8d7"

[[Unicode]]
uuid = "4ec0a83e-493e-50e2-b9ac-8f72acf5a8f5"
```

`Manifest.toml` describes a dependency graph. It states e.g. that `Dates` depends on `Printf` which in turn depends on `Unicode`. To avoid name clash the UUID of each package is also recorded. I was able to add `Dates` by just writing `add Dates` because the Julia package registry I use doesn't have any other packages named `Dates`. However if there had been other packages named `Dates` I would instead have had to write:

```
(Foobar) pkg> add Dates=ade2ca70-3891-5945-98fb-dc099432e06a
```

This would have uniquely identified my `Dates` package because no other `Dates` package would have this UUID.

To show the difference between direct and indirect dependencies, let us add `Printf` package directly.

```
(Foobar) pkg> add Printf
Updating registry at `~/.julia/registries/General`
```

```
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
  Updating `~/Development/Julia/Foobar/Project.toml`
    [de0858da] + Printf
  Updating `~/Development/Julia/Foobar/Manifest.toml`
    [no changes]
```

After this you can see that the `Project.toml` file has been updated.

```
name = "Foobar"
uuid = "94035000-e9dc-11e9-3ec0-43acaba33257"
authors = ["Erik Engeheim <erik@sixty-north.com>"]
version = "0.1.0"

[deps]
Dates = "ade2ca70-3891-5945-98fb-dc099432e06a"
Printf = "de0858da-6303-5e67-8744-51eddeeb8d7"
```

The `Manifest.toml` file however remains unchanged. However you are not required to look at these `.toml` files to get the status of your project dependencies. In package mode you can perform a number of useful operations such as checking current status.

```
(Foobar) pkg> status
Project Foobar v0.1.0
  Status `~/Development/Julia/Foobar/Project.toml`
  [ade2ca70] Dates
  [de0858da] Printf
```

## Environments

Let us explore further what an environment is and how it works. In our little `Foobar` package example we added the `Dates` and `Printf` packages.

A naive approach to adding dependent packages would be to store the code and resources of dependent packages inside the project directory. Many software systems do that.

As an old C++ developer this is what I have been used to in the past. Large number of third party libraries in the form of dynamically linked libraries, static libraries and include files are bundled together with your source code and all of it glued together with elaborate and complicated build scripts.

If you create more projects using the same libraries, this easily gets complicated. You end up duplicating the same libraries multiple times.

The Julia developers got a chance to think fresh about this before Julia 1.0 got released. Their decision was to put all downloaded or installed packages into a central depot on your hard drive. This depot is organized so that different versions of the same library can live side by side.

Each software project you have on your computer defines its own environment, its own little sandbox. Each of these environments point into this same depot.

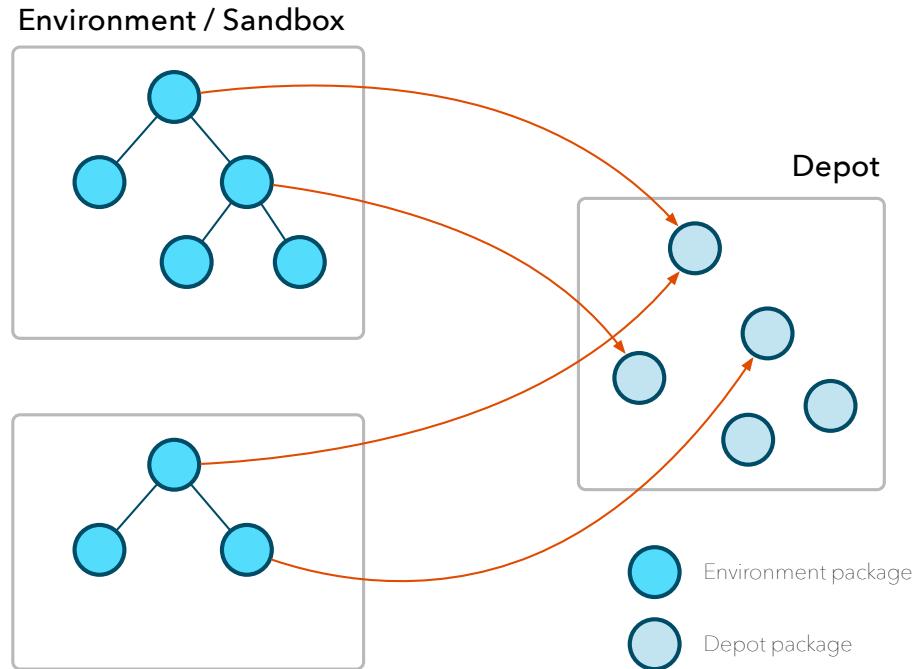


Figure 60: Link between environments and depot

At runtime Julia creates data structures in memory representing the depot and environments. `roots` is essentially a dictionary where you can use package names as key and get out the corresponding UUID. By reading packages stored in the depot, Julia constructs a `paths` data structure which allows you to use UUIDs as keys to lookup paths where the corresponding package is stored. Each package is stored using the same layout as you get when you generate a package with the package manager.

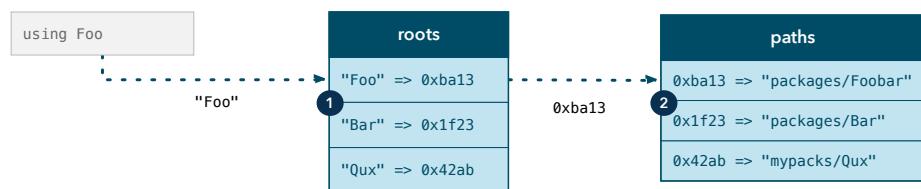


Figure 61: Mapping package names to UUIDs and UUIDs to depot paths

## Different Types of Environments

In Julia modules, packages and environments are overlapping concepts or aspects of the same thing. The package `Foobar` both defines a module `Foobar` and an environment `Foobar`. However not every environment corresponds to a package.

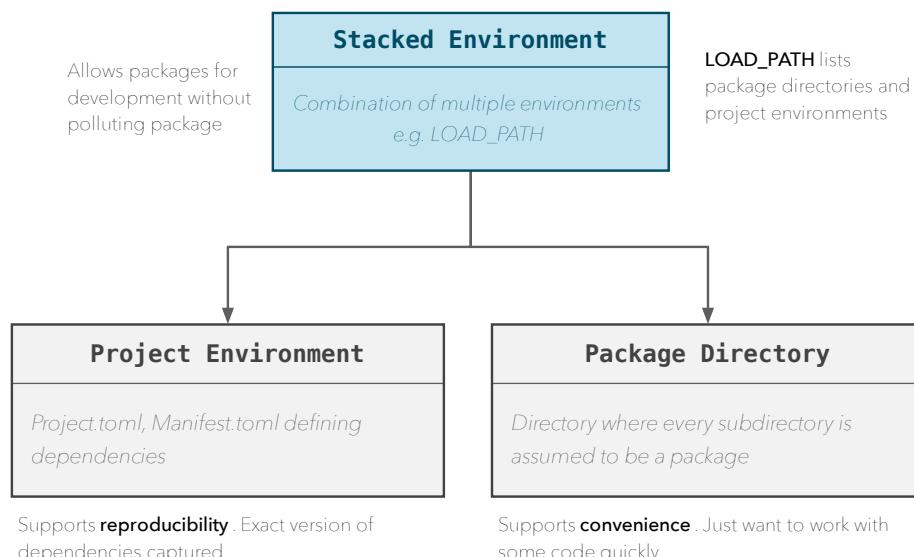
For instance when you start the Julia REPL you are in the `Main` module but

there is no corresponding `Main` package. If you are using Julia 1.5, then the initial active environment will be the `v1.5` environment. However there is no `v1.5` package.

Julia has three different kinds of environments:

- **Project Environment** An environment defined by a directory containing a `Project.toml` and potentially a `Manifest.toml` file defining what packages exist within the environment. This is how we have created packages.
- **Package Directory** This is a directory containing subdirectories containing packages. Don't mistake this for the directory containing the package itself.
- **Environment stack** is an environment created by combining other environments. The Julia `LOAD_PATH` global variable covered earlier, defines such a stacked environment.

Usually you will use all of these environments in combination when you are developing packages.



The `Foobar` package we made represented a *project environment*. The code you write in that package relates to this environment. However when you try to load and use the `Foobar` package in the REPL, you are not in the `Foobar` environment.

Instead when you are in the REPL you are in the environment stack defined by the `LOAD_PATH` global array. Thus when writing `using Foobar` or `import Foobar` Julia will look at environments listed in `LOAD_PATH` and use whatever matching definition it hits first.

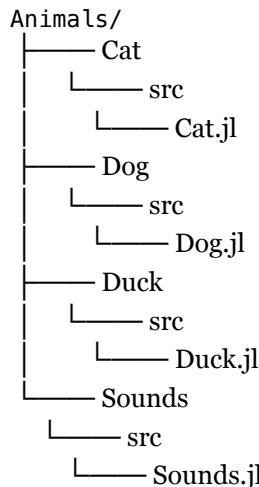
Thus an earlier environment can shadow a later environment if they have packages with the same name. It is quite similar to how the `PATH` environment variable works in an operating system. If you type e.g. `unzip` at the command line,

the shell will look for the first directory path listed in PATH which contains an executable file named `unzip`.

Usually you will put a *package directory* in the LOAD\_PATH so you can easily load your own packages.

## Package Directory

Let us make a simple package directory to better explain what it is. Below is a package directory called `Animals` containing the packages `Cat`, `Dog`, `Duck` and `Sounds`.



Notice how packages must follow this particular file and directory organization, otherwise Julia cannot recognize a subdirectory in `Animals` as a package. Observe how we don't need to have a `Project.toml` or `Manifest.toml` file in each of these packages.

The primary reason this option exists is backwards compatibility with the old Julia package system as well as convenience.

Each package just prints a greeting related to the animal name given to each module.

```

# Cat/src/Cat.jl
module Cat

greet() = println("meow")

end

# Dog/src/Dog.jl
module Dog

greet() = println("woof")

end

```

```
# Duck/src/Duck.jl
module Duck

greet() = println("quack")

end
```

We use all these separate packages in the Sounds package to print the sound each animal makes.

```
# Sounds/src/Sounds.jl
module Sounds

using Cat
using Dog
using Duck

function makesounds()
    Cat.greet()
    Dog.greet()
    Duck.greet()
end

end
```

To be able to use the packages we add `Animals` to `LOAD_PATH`, and then we are ready to run `makesounds()`.

```
$ cd Animals

$ julia

julia> push!(LOAD_PATH, pwd())

julia> using Sounds

julia> Sounds.makesounds()
meow
woof
quak
```

Notice we are able to run functions in `Sounds` without actually having added `Cat`, `Dog` and `Duck` as dependencies in the package. The reason for this is that without the `Project.toml` file `Sounds` is not an environment. Instead `Animals` is a package directory which is part of the environment stack defined by `LOAD_PATH`.

However as soon as you create a `Project.toml` file in `Sounds` it will become an environment and complain if you try to run it without explicitly adding dependencies to `Cat`, `Dog` and `Duck`.

Also you cannot add a package which does not have a `Project.toml` file. So if we turn both `Sounds` and `Cat` into project environments (adding `Project.toml`

file with info about package).

Secondly you cannot add local directories in Julia with the package add command. Instead you have to use the develop command:

```
(v1.5) pkg> activate Sounds
(Sounds) pkg> develop --local Cat
```

develop is slightly different in operation. When you add a package you fix it to a particular version. However when adding Cat using develop, Sounds will pickup all changes you make to Cat without requiring any explicit update. If you have imported Sounds you may want to restart the Julia REPL to import it over again and see the effects of having added Cat to its dependencies.

```
julia> push!(LOAD_PATH, pwd()) # if not already in path

julia> using Sounds
[ Info: Precompiling Sounds [aa8bf3b8-8de8-11e4-007f-e5bcc90f344c]
└ Warning: Package Sounds does not have Dog in its dependencies:
    - If you have Sounds checked out for development and have
      added Dog as a dependency but haven't updated your primary
      environment's manifest file, try `Pkg.resolve()` .
    - Otherwise you may need to report an issue with Sounds
└ Loading Dog into Sounds from project dependency, future warnings for Sounds are suppressed.
```

As you can see, once Sounds is its own environment, it will not tolerate that Dog has not been added explicitly.

## Environment stack

Let us explore how our LOAD\_PATH environment stack is setup.

```
julia> LOAD_PATH
4-element Array{String,1}:
 "@"
 "@v#.#"
 "@stdlib"
 "~/Development/Julia"
```

The last path has been added in my `startup.jl` file so I have access to packages I am personally developing. However if you look at all the previous paths they are weird looking. You can get a lot of information about what these mean by using the online help system:

```
help?> LOAD_PATH
```

One useful function mentioned is the `Base.load_path()` which turns all these paths into real readable paths.

```
julia> Base.load_path()
3-element Array{String,1}:
 ~/.julia/environments/v1.5/Project.toml" "/Applications/Development/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5"
```

```
"~/Development/Julia"
```

The "@" entry means the current active environment and will not always get listed. But we can if we want get it included by making an environment active:

```
(v1.5) pkg> activate Sounds
Activating environment at `~/Development/Animals/Sounds/Project.toml`
```

```
julia> Base.load_path()
4-element Array{String,1}:
"~/Development/Animals/Sounds/Project.toml"
"/Users/erikengheim/.julia/environments/v1.5/Project.toml"
"/Applications/Development/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5"
"~/Development/Julia"
```

When I am in the Julia REPL this environment stack will always be used. It defines my environment. So if I write `using Cat`, Julia will start looking for a match from the beginning of the `LOAD_PATH`. First we will look in the `Sounds` project environment for match. If a `Cat` package is referenced in its `Project.toml` we are done.

Otherwise Julia will continue and look in the `v1.5` project environment. This environment comes with the Julia 1.5 installation. Every Julia installation creates a different environment, so you can have different packages referenced when using different Julia versions.

Next Julia will search the the ".../julia/stdlib/v1.5"" directory which points to the location of the standard library bundled with Julia v1.5.

```
$ ls /Applications/Development/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5
```

Base64	LinearAlgebra	Serialization
CRC32c	Logging	SharedArrays
Dates	Markdown	Sockets
DelimitedFiles	Mmap	SparseArrays
Distributed	Pkg	Statistics
FileWatching	Printf	SuiteSparse
Future	Profile	Test
InteractiveUtils	REPL	UUIDs
LibGit2	Random	Unicode
Libdl	SHA	

You can see this lists all the modules bundled with Julia, which you can use without doing any separate downloads from the internet.

## Practical Julia Workflows

We have covered a lot about how the Julia package system works. While this may help you grasp what is going on, it is not always easy to use this information to come up with a practical way of working with Julia.

That is why in this section we will take the previous information and describe a practical Julia workflow.

The way I have shown you how to work previously by adding code to individual files and them include them or use the `-i` switch when launching Julia is not how I write most Julia code. Apart from small chunks of code it is not a practical way of working.

Here is a summary of how I work:

1. I have package directory added to my `config.jl` file where I place all my Julia projects.
2. Almost all my Julia development starts by generating a package in this package directory.
3. I extensively use the REPL when developing. In the REPL I typically import the `OhMyREPL` and `Revise` packages before importing the package I am currently developing.
4. If my work is split over several packages, I add dependencies using `develop --local DependentPackage`.

Let us walk through these points in more detail.

You want some directory where your packages are included in the `LOAD_PATH`, so you don't need a two-step process each time you want to import one of the packages you are developing.

My primary reason for always putting code inside a package is that that allows you to use the `Revise` package when developing. `Revise` will monitor changes to packages imported after it was imported. It will update the REPL to reflect changes to the source code of monitored packages.

`Revise` is just used for development so you should not add it to your package but rather to the default environment. Whenever you start Julia you will automatically be in the default v1.5 environment. However you can switch to it automatically with by writing `activate` in package manager mode with no argument.

Let us make a simple package `Monitored` to demonstrate how `Revise` works.

```
(v1.5) pkg> add Revise
  Updating registry at `~/.julia/registries/General` 
  Updating git-repo `https://github.com/JuliaRegistries/General.git` 
  Resolving package versions...

(v1.5) pkg> generate Monitored
Generating project Monitored:
  Monitored/Project.toml
  Monitored/src/Monitored.jl
```

We just make a minor modification to this package source code:

```
module Monitored
export greet

greet() = print("Hello Earth!")

end # module
```

Make sure the package was generated in a package directory in your LOAD\_PATH. Now we can import Revise and Monitored to check out how Revise tracks source code changes.

```
julia> using Revise; using Monitored

julia> greet()
Hello Earth!
```

Without restarting the REPL change the implementation of greet() and save the file.

```
greet() = print("Hello Mars!")
```

Jump back to the REPL and notice the change in the result:

```
julia> greet()
Hello Mars!
```

You did not have to issue any commands in the REPL to pickup this code change. While Revise helps Julia development a lot, it has some important limits which are important to be aware of.

One of the core principles in Julia which gives it, its performance advantage is that types are not allowed to change. This means that you are allowed to add new types to your Monitored package without restarting your REPL, however you cannot modify a type if you have already created a value of that type.

Other than that you are free to modify functions and add new functions.

## Purpose of Different Environment Types

Why do we need the complexity and potential confusion of three different environment types?

**Project environments** exists so you can distribute your packages to other people and they can then install that package with all the correct versions of dependent packages. Normally when you add a package its dependencies are downloaded and installed as well.

However if you clone a Git<sup>34</sup> repository of a package you want to contribute, you don't go through the normal Julia package manager machinery.

In this case you would have to make your package active and tell Julia to download all dependencies mentioned in the `Manifest.toml` file. Say you want to contribute to a package called Fudgeball made by Timmy located on Github.

```
$ git clone git@github.com:timmy/Fudgeball.jl.git Fudgeball
$ julia
julia> ]
(v1.5) pkg> activate Fudgeball
(Fudgeball) pkg> instantiate
```

---

<sup>34</sup>Git is a widely used version control system. You use it to keep track of modifications to your source code.

When you issue the `instantiate` command while in a project environment it will cause dependent packages to be downloaded and installed.

So one way of thinking of project environments is that they offer *reproducibility*.

**Package directories** and **Stacked environments** in contrast are really about aiding your development process. In a stacked environment you can put all sorts of package which you use in your REPL sessions but which the package you develop don't rely on. Examples of this would be packages like `Revise` and `Debugger`. You don't want to add those to your project environment (environment of a package you develop).

## Distributing Your Packages

Once you have made your own package you may want to distribute it to the wider Julia community. Julia packages are usually made available on Github<sup>35</sup> by creating a Git<sup>36</sup> repository with the same name as your package and the suffix `.jl`. So if your package was named `Foobar` then it should be stored in a Github repository named `Foobar.jl`.

Github while commonly used in the Julia community is not a requirement. Any publicly hosted git repository works. So you could also use Bitbucket if you prefer.

One of my packages `PList` is on Github and you could install it using the package manager in the Julia REPL:

```
pkg> add https://github.com/ordovician/PLists.jl
```

This is the simple approach if you just want to make your package available quickly with minimum hassle. Remember to check in the `Project.toml` and `Manifest.toml` files into your repository, otherwise users of your package will not be able to obtain the correct dependencies.

If you all this talk of Git, repositories and checking in is unknown to you, then we will do quick Git crash course. If you are familiar with Git, then you could just skip it.

## Git Crash Course

When writing a lot of code over longer time, there is a high probability that you make a mistake and want to go back to a previous version of your code. Or perhaps your code doesn't work anymore but you were absolutely certain it worked last week.

Version control is the magic solution to these problems. A version control system keeps tracks of previous versions of your source code. It keeps track of the history of your code. How that code changed over time.

---

<sup>35</sup>Github is one of the most popular web sites for hosting Git repositories. In fact the Julia source code itself is on Github.

<sup>36</sup>Git is a widely used version control system. You use it to keep track of modifications to your source code.

The topic is too big to cover in detail in this book. There are many version control systems such as SVN, Perforce and Mercurial. However we will focus on Git, because it is the most widely used version control system today. More importantly the Julia community has built its package system on top of Git.

Git itself is a large topic and so I strongly advice you to read a book covering it in more detail. The Pro Git book by Scott Chacon and Ben Straub is a Git book I have had good experience with.

Here is a simple example of creating a Package `Foobar` and putting it under version control. First we generate the package:

```
[Development] $ julia
(@v1.5) pkg> generate Foobar
Generating project Foobar:
  Foobar/Project.toml
  Foobar/src/Foobar.jl
```

```
(@v1.5) pkg>
```

Next we jump into the package directory we generated and initialize a Git repository. For Git, the repository is a hidden directory named `.git` where all the history of your files get stored.

```
$ cd Foobar
$ git init
Initialized empty Git repository in /Users/erik/Development/Foobar/.git/
```

After initialization the repository doesn't contain anything. We need to add files to it.

```
$ ls
Project.toml src
$ git add .
$ git status
On branch master
```

```
No commits yet
```

```
Changes to be committed:
(use "git rm --cached <file>..." to unstage)

  new file:   Project.toml
  new file:   src/Foobar.jl
```

Just adding files, doesn't mean they are ``committed'' or stored in the Git repository history. They are just what we call ``staged.'' The `git commit` command will take all files which are staged and record an entry in the repository history.

To remember what kind of code change you made for that particular commit, you provide a message with the `-m` switch.

```
$ git commit -m "My first commit"
[master (root-commit) 0fa8990] My first commit
```

```
2 files changed, 9 insertions(+)
create mode 100644 Project.toml
create mode 100644 src/Foobar.jl
```

Every time you have made useful code changes which you want to save to history in a commit, you need to first perform a `git add` on every file you want included. If you want to add every file in your current directory, you can just write:

```
$ git add .
```

At any point you can view the history of all your commits to see what code changes you have done over time. Here is an example of the top entries in the history of my Rockets Github repository which is the basis for the rocket code examples you see in this book.

```
$ git log
commit 44f9a366894e910578e5e3ac0fc85c3a6e0de4d8 (HEAD -> refs/heads/master, refs/remot
Author: Erik Engheim
Date:   Sun Sep 20 23:12:57 2020 +0200
```

Added the abstract factory pattern

```
commit ad6991dc60b36bfcd8d25f8a9c966754b59a379d
Author: Erik Engheim
Date:   Fri Sep 18 12:54:17 2020 +0200
```

Added data about Electron rocket

Obtained details from these website: <https://www.spacelaunchreport.com/electron.html>

Of course keeping your code in a local Git repository on your local hard drive doesn't help other people accessing your code. For that you want to put your Git repository on a Git repository hosting service such as Github or Bitbucket.

Normally you would use the web interface on these services to add a new Git repository. Once you have done that, you add a reference to this remote repository to your local Git repository like this:

```
$ git remote add origin git@github.com:ordovician/LittleManComputer.jl.git
```

In this case I am creating an alias called `origin` in my local Git repository configuration, which refers to a repository on Github called `LittleManComputer.jl`. I can then use this alias to easily push the code I have locally on my computer to Github:

```
$ git push origin master
```

This command tells git to push and synchronize the code I have on my `master` branch to the remote repository referred to by the `origin` alias. We will not get into details here about what branches are. Your code can exist in many different branches, but as a default it exists in a branch called `master` and when you start playing around with git you don't need to deal with any other branches.

Okay this was a very basic coverage, so if you don't know Git, I highly advice you to read up on it. Git is a big part of Julia development and plays a crucial

role in facilitating collaboration between different developers and using other people's code.

## Julia Registries

When you use the Julia package manager to add a package, it will look for the package in one of the recorded package registries. When you install Julia, it will be setup to point to the Julia General Registry. If for some odd reason it was not added you could add it like this:

```
pkg> registry add https://github.com/JuliaRegistries/General
  Cloning registry from "https://github.com/JuliaRegistries/General"
    Added registry `General` to `~/.julia/registries/General`
```

A registry is basically a Git repository, which contains information about where all the packages registered with this registry are located on the internet. For instance information about my `LittleManComputer` package is stored under `L/LittleManComputer`. It contains the following files describing the package:

```
Compat.toml
Deps.toml
Package.toml
Versions.toml
```

If you peek inside the `Package.toml` file you will see the following information:

```
name = "LittleManComputer"
uuid = "c742fd3c-88f6-4004-ba45-01ef0bf0104f"
repo = "https://github.com/ordovician/LittleManComputer.jl.git"
```

This tells us the name of the package, its unique identifier the UUID as well as where the package is located on the internet.

You could add this package to your environment using either of the two commands below:

```
pkg> add https://github.com/ordovician/LittleManComputer.jl
pkg> add LittleManComputer
```

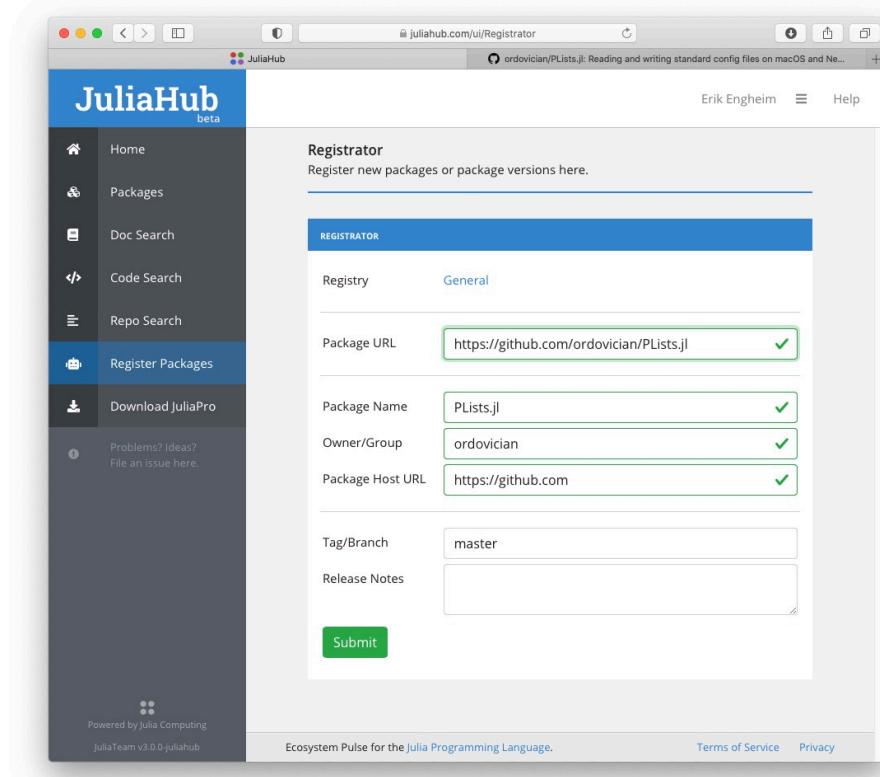
Ideally however you want to use the last command. But for this to work, you need to register your package in the Julia General Registry. This means adding an entry in this Github repository describing your package.

Since you most likely don't have write access to this repository you need to make this change and create a pull request. This means you have cloned the registry and made a change which adds your package. A pull request, is a request sent to the maintainers of a Github repository you don't have write access to, to add your code changes. We call it a *pull* request because they have to pull the changes from your copy of their repository.

In practice however you don't manually do this. Instead you automate this process using JuliaHub.

## JuliaHub

JuliaHub is your starting location for exploring Julia packages. You can search through the code of all registered packages, or for particular documentation. It also offers a great way of simplifying the registration of your packages.



The screenshot shows how I would register my `PLists` package with the Julia General Registry. You simply paste in the URL of your package and the rest of the information will be filled out. You just have to add the release notes.

However before you do this you should read the guidelines on the home page for the Julia General Registry: <https://github.com/JuliaRegistries/General>.

You can easily screw this up, so one of the best ways to get a lot of these details correct is to not use the regular `generate` command in the Julia package manager but instead use a Julia package called `PkgTemplates` to generate the scaffolding for your package. It adds bunch of details omitted by `generate` which is very barebones.

## Generate Package with `PkgTemplates`

With `PkgTemplates` you can setup a template for how to create your packages. Below we are setting up a template for packages that are meant to require Julia v1.4 as the minimum version. You will not get your package accepted to the

Julia General Registry if you don't provide information about minimum Julia version.

```
(@v1.5) pkg> add PkgTemplate

julia> using PkgTemplates
julia> t = Template(julia=v"1.4", dir=".")
```

Notice I also specify where I want my Julia packages to be generated with `dir=".."`. This places the generated package in my current directory. If I did not specify directory it would default to `"~/.julia/dev"` which I personally find impractical, as this is a hidden directory. Should your Julia installation get messed up and you want to start from scratch, it is tempting to erase the whole `~/.julia`, but this can have catastrophic results if you got the habit of putting your personal projects in there.

The `t` object produced from calling `Template` is callable object, which we can use to generate our package:

```
julia> t("Foobar")
[ Info: Running prehooks
[ Info: Running hooks
  Activating environment at `~/Development/Foobar/Project.toml`
    Updating registry at `~/.julia/registries/General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
No Changes to `~/Development/Foobar/Project.toml`
No Changes to `~/Development/Foobar/Manifest.toml`
  Activating environment at `~/.julia/environments/v1.5/Project.toml`
[ Info: Running posthooks
[ Info: New package is at ~/Development/Foobar
```

If we look at the contents of the generate package, you will notice that we get more files than what the simple `generate` command of the package manager gives you:

```
$ tree Foobar
Foobar
├── LICENSE
├── Manifest.toml
├── Project.toml
└── README.md
├── src
│   └── Foobar.jl
└── test
    └── runtests.jl
```

There are a number of important things to notice here:

- `LICENSE` file. It will default to an MIT license. If you don't provide a license, people will be reluctant to use your package.
- `test/runtests.jl` is a file where you can place test code. If your package is active in the package manager, then this file is run with the `test`

command.

The `Project.toml` file now contains more details than before:

```
$ cat Project.toml
name = "Foobar"
uuid = "95edb779-3e72-4e22-9236-d8c03be856c7"
authors = ["Erik Engheim <erik.engheim@earth.com> and contributors"]
version = "0.1.0"

[compat]
julia = "1.4"

[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Test"]
```

The `[compat]` section is important as your package will not get accepted without it. `[extras]` and `[targets]` are used to include packages which should only be made available when running tests.

Under `[extras]` you can list packages you don't want to add as dependencies to your package. These packages can then be listed under `[targets]`. The `test` target is the target for when tests are run. In this example you see it includes the `Test` package previously defined under `[extras]`. This makes it possible to write `using Test` in the `test/runtests.jl` file.

This way you avoid adding a dependency to the `Test` in your regular Julia package.

Please note however that this is the old way of specifying dependencies for the test environment. Starting with Julia 1.2 you can add your own special `test/Project.toml` file. The package specified here will be used when the test script `test/runtests.jl` is run.

This works like any other environment. If you want to add dependencies to this environment, you have to activate the `test` directory.

```
(HelloWorld) pkg> activate ./test
[ Info: activating environment at `~/Development/Foobar/test/Project.toml`.

(test) pkg> add Test
Resolving package versions...
Updating `~/Development/Foobar/test/Project.toml`
[8dfed614] + Test
Updating `~/Development/Foobar/test/Manifest.toml`
[...]
```

This adds the `Test` module, which allows you to write `using Test` in the `test/runtests.jl` script. `Test` contains a testing framework. It allows you to run functions in your module and test if it gives expected output.

---

A package made with PkgTemplates should be easy to register with JuliaHub. Just make sure you read the official guidelines, especially on naming your package and how to use version numbers correctly.



# Input and Output

- **Basic IO Functions.** Most Basic functions for doing input and output.
- **Loading Rocket Engines.** Show how we can use IO functions to load definitions of rocket engines from a file.
- **String representation** of different Julia objects.
- **Asynchronous Code.** Input and output often has to be done in an asynchronous fashion, especially network code due to high latency.
- **Networking.** Client and server setup. Reading and writing to TCP/IP sockets.

## Working with Files

When building our various spacecrafts, we have thus far hardcoded the different properties of the various rocket engines and propellant tanks. For instance to create a merlin rocket engine we have written:

```
julia> merlin = Engine(845e3, 282, 470)
SingleEngine(845000.0, 282.0, 470.0)
```

This is impractical. We want to have easy access to a whole catalog of rocket engines and propellant tanks to choose from. We want a table, like the one below, where one can easily add and remove lines representing different rocket engines.

	name	company	mass	thrust	throttle	Isp
	RS-25	Aerojet Rocketdyne	3.527	1860.0	0.8	366
	BE-3PM	Blue Origin	0.25	490.0	0.18	310
	LV-T30	Kerbal	1.25	205.16	0.0	265
	LV-T45	Kerbal	1.5	167.97	0.0	250
	48-7S	Kerbal	0.1	16.88	0.0	270
	Merlin 1D	SpaceX	0.47	845.0	0.7	282
	RD-180	NPO Energomash	5.48	3830.0	0.47	311
	Rutherford	Rocket Lab	0.035	18.0	0.3	303

This is a CSV file, which you can easily edit with a multitude of software packages. If you open it in a text editor it will look like this:

```
name,company,mass,thrust,throttle,Isp
```

```
RS-25,Aerojet Rocketdyne,3.527,1860,0.8,366
BE-3PM ,Blue Origin,0.25,490,0.18,310
LV-T30,Kerbal,1.25,205.16,0,265
LV-T45 ,Kerbal,1.5,167.97,0,250
48-7S,Kerbal,0.1,16.88,0,270
Merlin 1D,SpaceX,0.47,845,0.7,282
RD-180,NPO Energomash,5.48,3830,0.47,311
Rutherford,Rocket Lab,0.035,18,0.3,303
```

We want to be able to load data from these files and construct different engine and tank objects and store them in a dictionary. That way we can easily grab a Merlin 1D or RD-180 rocket engine as needed.

```
julia> engines = load_engines()
Dict{String,SingleEngine} with 8 entries:
  "Rutherford" => Engine(18000.0, 303.0, 35.0)
  "RD-180"      => Engine(3.83e6, 311.0, 5480.0)
  "LV-T30"      => Engine(205160.0, 265.0, 1250.0)
  "BE-3PM "     => Engine(490000.0, 310.0, 250.0)
  "LV-T45 "     => Engine(167970.0, 250.0, 1500.0)
  "Merlin 1D"   => Engine(845000.0, 282.0, 470.0)
  "RS-25"       => Engine(1.86e6, 366.0, 3527.0)
  "48-7S"       => Engine(16880.0, 270.0, 100.0)

julia> engines["Merlin 1D"]
Engine(
    thrust = 845000.0,
    Isp    = 282.0,
    mass   = 470.0
)
```

Want more engines? Just add more lines to the CSV file. Let us go over the details of how this can be achieved.

## Basic IO Functions

We start simple and look at the basics of opening a file and reading strings, number or lines from it. Once we got all the basics covered we can start building a `load_engines()` function and a `load_tanks()` function.

Opening a file is done with the `open` function which will return an `I0Stream` object, which is a subtype of `I0`.

```
julia> io = open("rocket-engines.csv")
I0Stream(<file rocket-engines.csv>)
```

`I0` is an abstract type which allows us to deal with reading and writing to files, buffers, external processes and pipes using the same API (application programming interface).

**NOTE**

When developers speak of an **API** what they mean is usually a set of functions performing related tasks. The *Julia IO API* is the set of functions which allow you to work with IO subtypes.

Let us play a bit with the `IO` object we obtained from the `open` call. This reads a single line from the currently open file:

```
julia> readline(io)
"name,company,mass,thrust,throttle,Isp"

julia> readline(io)
"RS-25,Aerojet Rocketdyne,3.527,1860,0.8,366"
```

Repeated calls give us successive lines in the file. We are not forced to read whole lines. Reading a single character is also possible:

```
julia> read(io, Char)
'B': ASCII/Unicode U+0042

julia> read(io, Char)
'E': ASCII/Unicode U+0045
```

`read` takes as second argument, the type you want to read. What type should you specify? That depends on how data was originally stored in the file.

If it is just a bunch of text, it does not make sense to call `read(io, Int64)`. In that case four consecutive ASCII characters would be interpreted as an integer number.

However if you got an UTF-8 formatted text file, you can read the whole file, and store it in a string object with a `read(io, String)` function call.

Once you are done reading from a file, you need to close the `IOStream` object referring to the file.

```
julia> close(io)
```

If you don't you would be leaking limited resources. Also if file writing is buffered you risk not completing the physical writing on disk.

## Open Modes

To avoid accidentally overwriting an important file, the `open` function will only return an `IOStream` object limited to reading operations. Any attempted at write operations would fail.

However there is an optional second argument, which allows you to provide the *mode* in which the file should be opened. This can be done either with a text string or a named arguments.

The simplest case is to open a file for reading. Since this is done frequently, you can omit the mode specification.

```
stream = open("filename", "r")
stream = open("filename")
```

But if you want to open a file for writing, you have to be explicit and provide a second argument:

```
stream = open("filename", "w")
stream = open("filename", write = true)
```

There are a myriad of combinations here, so I have collected all the possibilities in this table.

Mode	Description	Keywords
r	read	
w	write, create, truncate	write = true
a	read, write, append	append = true
r+	read, write	read = true, write = true
w+	read, write, create, truncate	truncate = true, read = true
a+	read, write, create, append	append = true, read = true

Details about the whole IO API can be read in the Julia reference manual or you can look it up in the REPL. If you ask for help on the `open` function in the REPL you get a complete overview over all the different open modes and different ways of calling `open`.

```
help?> open
search: open isopen propertynames CompositeException operm hasproperty getproperty

open(filename::AbstractString; keywords...) -> IOStream
```

Open a file in a mode specified by five boolean keyword arguments:

We don't cover every aspect of Julia's IO API here because we are not trying to be a reference manual but teach you how you can combine these functions to create practical solutions.

## Loading Rocket Engines

Now that we have explored the basics, it is time to look at how we can combine these functions to load rocket engine data from a CSV file. Below is an implementation of the `load_engines()` function.

```
function load_engines(path::AbstractString)
    rocket_engines = Dict{String, SingleEngine}()

    lines = readlines(path)
    for line in lines[2:end]
        parts = split(line, ',')
        name, company = parts[1:2]
        mass, thrust, throttle, Isp =
```

```

        parse.(Float64, parts[3:end])

    engine = Engine(thrust * thrust_multiplier,
                    Isp,
                    mass * mass_multiplier)
    rocket_engines[name] = engine
end

rocket_engines
end

function load_engines()
    load_engines("data/rocket-engines.csv")
end

```

Let us go through the different parts of the function and explain it more in detail by playing with the different parts of the code in the Julia REPL environment.

First we read in the data from the file and store every line as an element in an array.

```
julia> lines = readlines("rocket-engines.csv")
9-element Array{String,1}:
 "name,company,mass,thrust,throttle,Isp"
 "RS-25,Aerojet Rocketdyne,3.527,1860,0.8,366"
 "BE-3PM ,Blue Origin,0.25,490,0.18,310"
 "LV-T30,Kerbal,1.25,205.16,0,265"
 "LV-T45 ,Kerbal,1.5,167.97,0,250"
 "48-7S,Kerbal,0.1,16.88,0,270"
 "Merlin 1D,SpaceX,0.47,845,0.7,282"
 "RD-180,NPO Energomash,5.48,3830,0.47,311"
 "Rutherford,Rocket Lab,0.035,18,0.3,303"
```

The for-loop will pick every line in succession. Let me grab a random line to show you how each line is processed in the for-loop.

```
julia> line = lines[2]
"RS-25,Aerojet Rocketdyne,3.527,1860,0.8,366"
```

First the line is split into multiple parts. We use the `split` function to separate the line into multiple pieces. The separator is given as the second argument. Here we separate using a comma, but it could have been any string or character.

```
julia> parts = split(line, ',')
6-element Array{SubString{String},1}:
 "RS-25"
 "Aerojet Rocketdyne"
 "3.527"
 "1860"
 "0.8"
 "366"
```

We use destructuring to conveniently extract the array elements we want.

```
julia> name, company = parts[1:2]
2-element Array{SubString{String},1}:
 "RS-25"
 "Aerojet Rocketdyne"
```

We want `mass`, `thrust`, `throttle` and `Isp` as floating-point numbers, so we first get them as an array of strings, and then parse each string utilizing broadcasting (adding a dot after `parse`).

```
julia> mass, thrust, throttle, Isp = parse.(Float64, parts[3:end])
4-element Array{Float64,1}:
 3.527
 1860.0
 0.8
 366.0
```

This allows us to finally construct a rocket Engine object. Here we are not providing number of engines and hence a `SingleEngine` object rather than an `EngineCluster` object is created.

```
julia> engine = Engine(thrust, Isp, mass)
SingleEngine(1860.0, 366.0, 3.527)
```

## Different ways of using IO functions

One of the strengths of Julia is how composable functions are. The *IO API* shows this very well. To demonstrate how, I will show you several different code examples achieving exactly the same thing.

Earlier we used the `open` function to acquire an `IOStream` object. Look at this first code example. Do you see any problems with this approach?

```
io = open("rocket-engines.csv")
lines = readlines(io)
close(io)
```

The main problem is that you have to remember to call `close` on the `IO` object. If you don't, you start leaking limited resources. That is easy to forget, so Julia provides higher level functions to avoid this. `open` has a method which takes a function object `f` as first argument. The argument passed to `f` is the `IOStream` object which was created from the ``normal'' `open` method. After this function object is executed, Julia will close the stream for you.

```
lines = open(io -> readlines(io), "rocket-engines.csv")
```

Since the function object is given as first argument, the do-end form is available to us, so we could also have written:

```
lines = open("rocket-engines.csv") do io
    readlines(io)
end
```

But can we simplify this even more? Why provide an anonymous function, which takes an `IO` object as argument, when that is what `readlines` already

does? You can just pass `readlines` directly as the function argument.

```
lines = open(readlines, "rocket-engines.csv")
```

## Create Engines By Reading One Line at a Time

Let us put together what we have learned and create an alternative approach to reading rocket engine data from a CSV file. In our first example we read all the lines in one go. Then we iterated over an array containing these lines.

However in this example we read one line at a time from the file and create an engine as we go.

```
engines = open("rocket-engines.csv") do io
    readline(io)
    map(eachline(io)) do line
        parts = split(line, ',')
        mass, thrust, _, Isp =
            parse.(Float64, parts[3:end])
        Engine(thrust, Isp, mass)
    end
end
```

Why would we want to do that? For small files it is fine to read the whole thing into an array. However if you work with really large files, you would be wasting a lot of memory if you read the whole thing into memory before processing it. If you read one line at a time, you can release the memory allocated for that line before reading the next line.

This is also a demonstration of a more functional approach. Our original solution was more imperative as we mutated a data structure on every iteration. In this case we map each line to an `Engine` object, producing an array of `Engine` objects.

## String Representations

A neat thing about Julia is that when you create your own custom Julia types they automatically have a textual representation in the REPL environment. Hence as you create and modify objects, you can quickly see in the REPL environment, how they are changed.

However the default display of a Julia object is not always what you want. More complex data types have more complex internal organization you may not want to show to the user.

Other times you may be working in a different development environment which has opportunities for more than just a textual representation, perhaps it can draw graphics? For instance we want a graph to be shown as a drawing, rather than as a bunch of letters or symbols.

Julia has a very sophisticated and flexible system for representing data in different ways. We are going to cover how this system works. The easiest way to do that, is to work through an example.

## Pretty Objects

Consider our space rocket. We utilized named function arguments to create a sort of DSL (domain specific language) for describing a `SpaceVehicle`. This is very easy to read.

```
SpaceVehicle(
    Rocket(
        tank = Tank(
            dry_mass = 23.1e3,
            total_mass = 418.8e3
        ),
        engine = Engine(
            thrust = 845e3,
            Isp = 282,
            mass = 470,
            count = 9
        )
    ),
    Rocket(
        payload = SpaceProbe(22e3),
        tank = Tank(
            dry_mass = 4e3,
            total_mass = 111.5e3
        ),
        engine = Engine(
            thrust = 31e3,
            Isp = 311,
            mass = 52
        )
    )
)
```

However in the Julia REPL it is shown as:

```
SpaceVehicle(Rocket(Rocket(NoPayload(), Tank(23100.0, 418800.0, 395700.0), SingleEngi
```

Which is not very readable. Can we make it look better?

## Show

To customize how a space vehicle is shown, we have to add methods to the `show` function. `show` is what is called by the REPL to show an object. First, how do we change the rocket engine and propellant tank representations? There is already a default representation which gives us this:

```
julia> merlin = Engine(845e3, 282, 470)
SingleEngine(845000.0, 282.0, 470.0)
```

```
julia> tank = Tank(4e3, 111.5e3)
Tank(4000.0, 111500.0, 107500.0)
```

But what we actually want, is for it to look like this:

```
julia> merlin = Engine(845e3, 282, 470)
Engine(
    thrust = 845000.0
    Isp     = 282.0
    mass    = 470.0
)

julia> tank = Tank(4e3, 111.5e3)
Tank(
    dry_mass   = 4000.0
    propellant = 107500.0
    total_mass = 111500.0
)
```

To do that, we register two new methods for `show`, handling the `Engine` and `Tank` type.

```
const tab = "    "

function show(io::IO, engine::Engine)
    println(io, "Engine(")
    println(io, tab, "thrust = ",
            thrust(engine), ",")
    println(io, tab, "Isp    = ",
            Isp(engine), ",")
    println(io, tab, "mass   = ",
            mass(engine))
    print(io, ")")
end

function show(io::IO, tank::Tank)
    println(io, "Tank(")
    println(io, tab, "dry_mass   = ",
            tank.dry_mass, ",")
    println(io, tab, "propellant = ",
            tank.propellant, ",")
    println(io, tab, "total_mass = ",
            tank.total_mass)
    print(io, ")")
end
```

The REPL will call `show` and provide an `IO` object representing standard output (`stdout`). Functions such as `print` and `println` default to writing to `stdout` but you can provide your own `IO` object to write to, which is what we do here.

## IO Type Hierarchy

At this point it is useful to actually talk about how `IO` and `IOStream` objects are related. `IO` is an abstract datatype forming an elaborate type hierarchy.

The point of this type hierarchy is that it allows us to write functions such as `show`, which can be used in combination with many different input and output

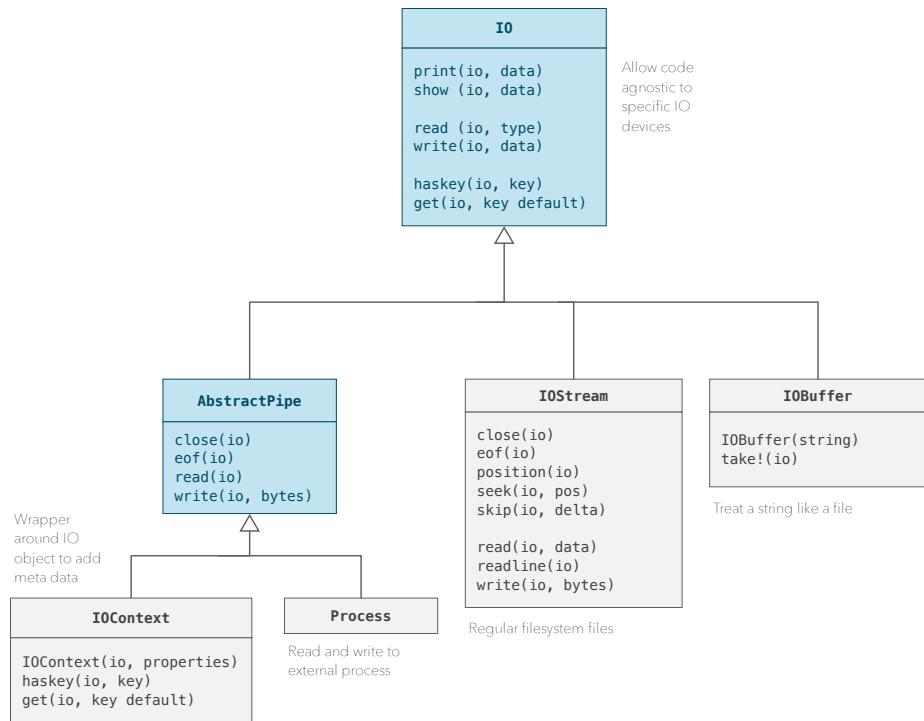


Figure 62: Type hierarchy of numbers in Julia

devices or concepts. I can use `show` whether I am writing to a string, standard output, a file, a network socket, a string buffer or even a unix pipe. The latter case allows us to send information from one process to another.

It also shows the beauty of Julia's multiple dispatch system. In an object-oriented language `show` would have had to be a method belonging to say a `Showable` interface. Objects which desired to show themselves would have to implement this `show` method.

This poses a problem: What if I got the `Engine` and `Tank` types from another package? Somebody else made this package. Perhaps they never implemented a `show` method. In an object-oriented language, the only way for me to change the visualization would have been to subclass `Engine` and `Tank` and add a specific `show` method implementation. However for this to work I would somehow have to be able to replace every instantiation of `Engine` or `Tank` with my own subtypes.

#### NOTE

There are a number of exceptions to this general case. In Objective-C and Swift you can add methods to existing classes through what is called categories or class extensions. Python would also allow you to add methods to existing classes, but there is no special syntax for it, nor does it seem to be common

practice.

The advantage of multiple dispatch in this case is that you can add a custom visualization to a data type you did *not* make yourself. You don't need access to the source code for either the data types you want visualized or the IO subtypes.

Nor are you required to do a lot of upfront design work. Somebody working in an object-oriented language may have to decide on a Showable interface ahead of time for a whole collection of classes to implement. The creators of Julia's show system did not have to think of this ahead of time. They could have designed the IO hierarchy and numerous Julia types first, then later add the show system without needing any kind of refactoring<sup>37</sup>.

## Recursive Show

We are not quite done with the string representation of the space vehicle. If you look at the full definition of our space vehicle, different parts are indented with whitespace for readability. Let us incorporate whitespace indentation into our show function.

```
SpaceVehicle(
    Rocket(
        Tank(
            dry_mass    = 4000.0,
            propellant = 107500.0,
            total_mass = 111500.0,
        ),
        Engine(
            thrust     = 845000.0,
            Isp       = 282.0,
            mass      = 470.0
        )
)
```

We will use a variable named `depth` to keep track of the indentation level. Exponents are quite useful in this case. As you can see in this example they allow us to easily repeat a character or string multiple times, to form a longer string.

```
julia> 'A'^3
"AAA"

julia> "AB"^3
"ABABAB"

julia> " " ^3
"   "

julia> " " ^5
```

---

<sup>37</sup>Refactoring code means to change the structure of the code without changing its behavior. After a refactoring the code should do the exact same thing as before.

```
"    "
```

We can combine this with depth to create an arbitrary indentation level. Here it is used to define how the Rocket type should be displayed.

```
const tab = "    "

function show(io::IO, r::Rocket, depth::Integer = 0)
    println(io, tab^depth, "Rocket(")
    depth += 1

    show(io, r.tank, depth)
    println(io, ",")
    show(io, r.engine, depth)

    if r.payload != nopayload
        println(io, ",")
        show(io, r.payload, depth)
    end
    println(io)

    depth -= 1
    print(io, tab^depth, ")")
end
```

Notice how depth is incremented before passing it on to the show function for the Tank and Engine types. Their show methods need to be modified to incorporate depth of indentation.

```
function show(io::IO, engine::Engine, depth::Integer = 0)
    println(io, tab^depth, "Engine(")
    depth += 1
    println(io, tab^depth,
            "thrust = ",
            thrust(engine), ",")
    println(io, tab^depth,
            "Isp    = ",
            Isp(engine), ",")
    println(io, tab^depth,
            "mass   = ",
            mass(engine))
    depth -= 1
    print(io, tab^depth, ")")
end

function show(io::IO, tank::Tank, depth::Integer = 0)
    println(io, tab^depth, "Tank(")
    depth += 1
    println(io, tab^depth,
            "dry_mass = ",
            tank.dry_mass, ",")
    println(io, tab^depth,
```

```

        "propellant = ",
        tank.propellant, ",")
println(io, tab^depth,
        "total_mass = ",
        tank.total_mass, ",")
depth -= 1
print(io, tab^depth, ")")
end

```

To put the cherry on top we, we define `show` for the `SpaceVehicle` type:

```

function show(io::IO, ship::SpaceVehicle)
    println(io, "SpaceVehicle(")
    show(io, ship.active_stage, 1)
    println(io)
    println(io, ")")
end

```

## What if I want a string instead of printing?

`show` has obvious utility for printing stuff to standard output, files, pipes etc. However often you want to get the representation as an `String` object, which you can pass around. How do you do that? That is when we use the Julia `repr` function. Here are some examples to clarify how it differs from `show`.

```

julia> tank = Tank(4e3, 111.5e3);

julia> show(tank)
Tank(
    dry_mass    = 4000.0,
    propellant = 107500.0,
    total_mass = 111500.0,
)

julia> repr(tank)
"Tank(\n    dry_mass    = 4000.0,\n    propellant = 107500.0,\n    total_mass = 111500.0,\n)"

julia> repr(1)
"1"

julia> repr(zeros(3))
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.3333333333333314829616256247390992939472198486328125"

julia> A = [1 2; 3 4];

julia> repr(A)
"[1 2; 3 4]"

```

`repr` can also take a MIME type as an extra argument. We already touched

upon this in the Strings chapter. MIME types allows us to get different text representations of the same type. For instance an array could be represented as something that fits in a CSV file or as an HTML table.

```
julia> repr(MIME("text/plain"), A)
"2×2 Array{Int64,2}:\n 1  2\n 3  4"
```

An alternative to `repr` is to use the `sprint` function. You can provide it to a function which writes to an `I0` object. For instance `println` can write to an `I0` object. But what if we want to collect the output of `println` or `show` into a string instead? Then we can use the `sprint` function. It takes a function as first argument. `sprint` will pass an `I0` object to this function as well as all other arguments.

```
julia> sprint(println, "hello world")
"hello world\n"
```

```
julia> sprint(show, A)
"[1 2; 3 4]"
```

```
julia> sprint(show, zeros(3))
"[0.0, 0.0, 0.0]"
```

If you don't want to return a `String` object but actually want to print to screen, you can use the `display` function

```
julia> display(MIME("text/plain"), A)
2×2 Array{Int64,2}:
 1  2
 3  4
```

## Registering Representations for Different MIME Types

Most Julia types only have a plain text representation so trying different MIME types with `repr` or `display` does not give any interesting results. Thus we are going to create our own type `Point`, and demonstrate how to add the ability to represent it in plain text, JSON and XML format.

```
struct Point
    x::Float64
    y::Float64
end
```

To do that we need to add methods to `show`. That is because both `repr` and `display` use `show` to do their work. We can begin with the really simply case where we don't specify a format at all.

```
import Base: show
show(io::IO, p::Point) = print(io, "($p.x, $(p.y))")
```

I like to think of this as the fallback mechanism. It gives us some kind of representation of the type, regardless of what kind of MIME types our development environment supports.

Remember in Strings how we covered creating instances of MIME objects and their corresponding type? `MIME("foo/bar")` is a MIME object while `MIME"foo/bar"` is the corresponding type of that object. That is important to keep in mind when we register different text representations for the `Point` type below.

Here we add a method which will get called each time Julia wants to get a JSON representation of a `Point` object.

```
function show(io::IO, ::MIME"text/json", p::Point)
    print(io, "{x = $(p.x), y = $(p.y)}")
end
```

This is for plain text representation, which will most commonly be used:

```
function show(io::IO, ::MIME"text/plain", p::Point)
    print(io, "Point($(p.x), $(p.y))")
end
```

And finally we have an XML representation:

```
function show(io::IO, ::MIME"text/xml", p::Point)
    println(io, "<point>")
    println(io, "\t<x>$p.x</x>")
    println(io, "\t<y>$p.y</y>")
    println(io, "</point>")
end
```

With these methods, we can now use `repr` and `display` with the registered MIME types.

```
julia> p = Point(10, 3);
julia> display("text/json", p)
{x = 10.0, y = 3.0}

julia> display("text/plain", p)
Point(10.0, 3.0)

julia> repr("text/json", p)
"{x = 10.0, y = 3.0}"

julia> repr("text/plain", p)
"Point(10.0, 3.0)"

julia> display(MIME("text/xml"), p)
<point>
  <x>3.0</x>
  <y>4.0</y>
</point>
```

`display` works by iterating over a list of displays stored by the Julia display system. One of the most common types of these displays is the `TextDisplay`, which is most commonly used with the standard Julia REPL. The Julia standard library contains `display` methods dealing with different types of displays.

Slightly simplified the `display` method called by the normal Julia REPL looks like this.

```
display(disp::TextDisplay, mime::MIME, x) = show(disp.io, mime, x)
```

You can see the text display maintains its own `IO` object, which is used when it invokes `show`.

## Context Dependent Display of Objects

MIME types are not the only way you can configure how an object is displayed. We can also add data to the `IO` object itself. Now why would we ever want to do that? The reason is: How to best display a value depends on context. Showing a value by itself on the Julia REPL is different from showing it in e.g. a large matrix, or as a key in a dictionary.

Here is an example of how we can change the display of our `Point` type depending on context. In an array we might want to show a point as `(2, 3)` while by itself we would prefer the longer form `Point(2, 3)`.

```
function show(io::IO, ::MIME"text/plain", p::Point)
    if get(io, :compact, true)
        print(io, "($p.x, $p.y)")
    else
        print(io, "Point($p.x, $p.y)")
    end
end
```

We interact with the `IO` object as if it was a dictionary. Remember `get` is used with a dictionary when you don't know if a key is present or not. `get(io, :compact, true)` will be true if `io` does not have a key named `:compact`. By convention `io[:compact]` should be true if we want a short and compact representation of an object. So here I am adhering to Julia conventions and not just checking on some arbitrary key.

It is not always obvious when Julia will seek to get a compact representation of an object, so we can fake it ourselves by doing exactly the same as the Julia's standard library.

A regular `IO` object such as `stdout` is wrapped around in an `IOContext` object. Anyone with an object-oriented background may recognize this as using the decorator pattern.

```
julia> io = IOContext(stdout, :compact => false);
```

We can use this wrapper to see the non-compact version of our `Point` type.

```
julia> show(io, MIME("text/plain"), p)
Point(10.0, 3.0)
```

And finally a compact version:

```
julia> io = IOContext(stdout, :compact => true);
```

```
julia> show(io, MIME("text/plain"), p)
```

```
(10.0, 3.0)

julia> repr("text/plain", p, context=io)
"(10.0, 3.0)"
```

## Asynchronous Code

The next input and output topic would naturally have been network communications. However we cannot jump straight to it because network communications present new challenges that do not exist with file IO. *Latency* is considerably higher with network communications. That means we risk ending up with inefficient code, which sits around waiting for a response to a request for some data.

A good example of this is a mobile phone application. Perhaps you want to show an address-list of people, which the user can click on to call or connect to. Data about users could be on another server. We don't want the user interface to remain *unresponsive* while we are waiting for the name and picture of each person in the list.

Instead the common choice is to fetch names first and fill up the list of names. Next we send a request for images. However instead of waiting for a response, we insert an animated spinner where the picture should have been. While waiting for the picture to be retrieved, we want to support user interactivity. Code checking for finger taps should run while the image is downloaded. For instance a user may want to tap information about a particular address-list contact, before all the images have been downloaded.

Consequently we need is a way of handling multiple tasks at the same time which are only partially done, and which may be resumed at any point, once some event has happened, such as more data becoming available on the network socket.

## Coroutines vs Threads

There are many ways of doing this, but the approach pursued here is referred to as *coroutines*. Coroutines may look similar to threads<sup>38</sup> in usage, but there are important differences. Both represent separate tasks which are carried out independent of each other. The microprocessor may do a bit of work on one task, then switch to another do some work on that one, before switching back to the first task and continue where it left off.

The key difference is how this switching is performed. With coroutines, you have to *explicitly give up control* in one task for another task to resume. With threads the operating system will just interrupt a task whether the task wants to or not and pass over control to another task.

---

<sup>38</sup>A process (running program) can have many threads of execution. A thread is a sequence of instructions executed by a CPU core. The OS can switch between different threads and keep track of where a previously executed thread left off before resuming it.

There are advantages and disadvantages to both approaches. Back in the 80s and 90s there was a battle between operating systems using *cooperative* and *preemptive multitasking*. Cooperatives multitasking is based on coroutines while preemptive multitasking work like modern threads. Cooperative multitasking fell out of favor because a badly written program could forget to give up control and hence render all other programs unresponsive.

For a single application however, coroutines are often much easier to work with than threads. The reason is that tasks are usually using shared resources. By shared resources we mean objects or values in memory. With threads you can have a task beginning to modify an object only to be interrupted in the middle of this modification. The object may then be left in a bad or undefined state. The new task which just gained control could try to use this partially modified object and everything turns into a mushroom cloud. For good reason, multithreaded code is hard to debug.

With coroutines you make your life a lot easier. No other task can take over control unless you explicitly allowed it to do so. Thus you can safely modify objects without risking that some other task suddenly jumps in and takes over the CPU before you are done.

How do you know when control is passed to another task? Any function call which is setup to block will hand over control to another task.

A typical example would be operations such as reading and writing to an I/O object. These operations block until they are carried out. For instance if you try to read a line from `stdin`, Julia will block the execution of your program until you have typed something on the keyboard and hit enter. More specifically, the current task is blocked.

Julia keeps a pool of tasks at different state of execution. Whenever one task blocks because e.g. it is waiting for something, Julia will check with its pool of tasks to see if any of them are ready to continue execution. The entity handling this is the scheduler. It tries to make sure all tasks get to run in turn.

## Tasks and Channels

In Julia we have two types which are the foundation for working with coroutines:

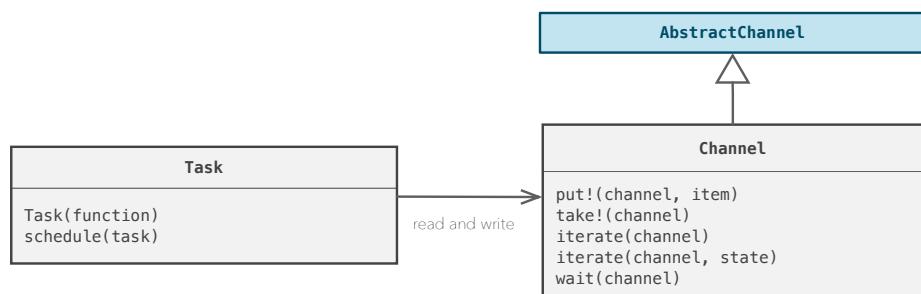


Figure 63: Tasks and Channels

- Task is an abstraction of a task being performed. An object representing a coroutine. It points to a function being run, and keeps track of how far execution has progressed inside this function.
- Channel is a way to communicate between tasks. Channels work like FIFO (First in First Out) queues. You push values into the channel from one task. Another task pulls values out at the other end of the channel.

To get a sense of how this works, we will look at a simple example involving both a task and a channel. We have a function `generator()` which returns a `Channel` object.

```
function generator()
    Channel() do channel
        for i in 0:5
            put!(channel, Char('A' + i))
        end
    end
end
```

You can see there is a loop putting characters in the alphabet into a channel. Using the `take!` function we can fetch individual characters from this `Channel`:

```
julia> g = generator()
Channel{Any}(sz_max:0,sz_curr:1)

julia> take!(g)
'B'

julia> take!(g)
'B'

julia> take!(g)
'C'

julia> take!(g)
'D'

julia> take!(g)
'E'
```

This `Channel` is iterable, thus we can collect values from it as if we were iterating over a collection.

```
julia> g = generator()
Channel{Any}(sz_max:0,sz_curr:1)

julia> collect(g)
6-element Array{Any,1}:
 'A'
 'B'
 'C'
 'D'
 'E'
```

```
'F'
```

Since the splat operator ... relies on iteration, we can use the channel with the splat operator as well.

```
julia> g = generator()
Channel{Any}(sz_max:0,sz_curr:1)

julia> [g...]
6-element Array{Char,1}:
 'A'
 'B'
 'C'
 'D'
 'E'
 'F'
```

Even the `string` function is flexible enough to work with channels. With splat the channel is emptied into an a function call with each character as an argument to `string`, thus constructing a `String` object.

```
julia> g = generator()
Channel{Any}(sz_max:0,sz_curr:1)

julia> string(g...)
"ABCDEF"
```

You may be fooled into thinking that all this code does is:

1. Iterate over the range `0:5`
2. Store the characters `A' to `F' in the `Channel` object.
3. Later pull the values out of the `Channel` object.

However that is *not* what happens. There is only ever 1 value in this `Channel`. Once that value is read with `take!` the loop advances and puts another value in the channel.

You don't have to take my word for it. You can easily prove it yourself by adding a `println` statement:

```
function generator()
    Channel() do channel
        for i in 0:5
            println("putting ",
                    Char('A' + i),
                    " into the channel")
            put!(channel, Char('A' + i))
        end
    end
end
```

If the `Channel` got filled up with all values before `generator()` returned it, we would have expect to see the following printed:

```
julia> g = generator()
```

```

Channel{Any}(sz_max:0,sz_curr:1)

putting A into the channel
putting B into the channel
putting C into the channel
putting D into the channel
putting E into the channel
putting F into the channel

```

However this is *not* what we actually see. Instead we get this interaction:

```

julia> g = generator()
Channel{Any}(sz_max:0,sz_curr:1)

julia> take!(g)
putting A into the channel
'A'

julia> take!(g)
putting B into the channel
'B'

julia> take!(g)
putting C into the channel
'C'

```

Obviously `println` is called every time `take!` is called. This needs a more detailed explanation.

## What is going on under the hood?

Understanding exactly what is going on in this code is hard, because it is at such a high level of abstraction. To show you what is actually going on I will show the more verbose version of the `generator()` function.

```

function generator()
    channel = Channel(2)

    function make_letters()
        for i in 0:5
            put!(channel, Char('A' + i))
        end
    end

    letter_task = Task(make_letters)
    schedule(letter_task)

    channel
end

```

In the initial version we used a `Channel(f)` constructor, taking a function object `f` as its only argument. This constructor did a whole bunch of stuff, which is

spelled out more clearly in the more verbose code example above. Here we construct a `Channel` with capacity 2. That means it can take two items, before `put!` blocks and tries to move control over to another task.

`make_letters()` is a function containing a for-loop producing individual letters from A to F. We turn this function into a task by providing it to the `Task` constructor. That `task` object is responsible for keeping track of how far the execution of `make_letters()` has advanced. Without this function, the task would not know what specifically it is supposed to do. The function is in effect a description of what the task does. A task however needs more than a description. It also needs to keep track of how far the execution has reached, whether it is waiting for something to complete or is ready to run. That is why a task has to be represented by an object of type `Task` and not merely a function object.

`schedule(task)` puts the task in the scheduler queue and begins to run it. The scheduler will at that point contain two separate tasks. One representing the REPL constantly asking the user to type input and another representing the character producer. The scheduler will alternate between suspending and resuming the execution of each of these tasks.

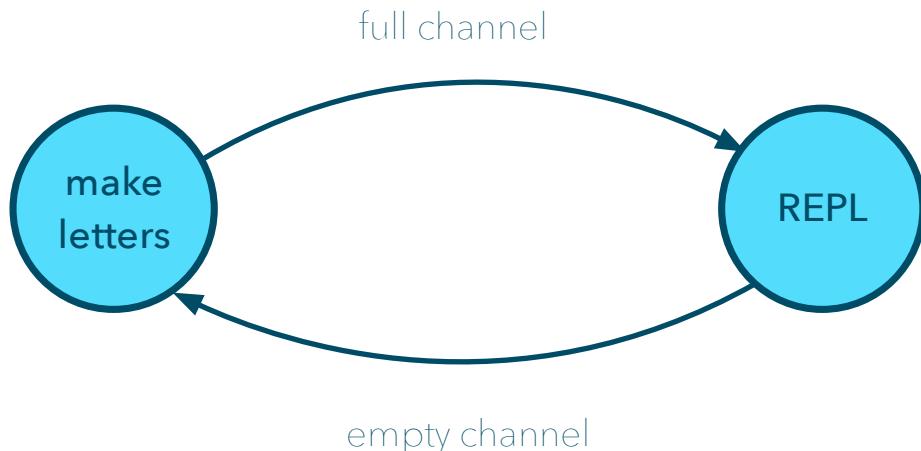
When the `letter_task` task hit the `put!(channel, Char('A' + i))` call it will fill the channel with a character. If the channels is not full it will continue. In the first example our channel had zero capacity, and would thus immediately block on a `put!` call.



Figure 64: A letter producing task blocking

However `channel = Channel(2)` means we made a channel which could hold a max of two characters. Hence when we try to put the third character into the channel, it will block. Blocking causes control to be handed over to the task scheduler, which looks through the scheduling queue, in order, for the next task. The scheduler will pick the first task it finds which is no longer blocking and ready, for execution. In our case that will be the Julia REPL.

When you make a call to `take!(channel)` in the REPL you get the opposite effect. It will complete as long as there are characters in the channel. However if you attempt to take a character from an empty channel, it will cause execution to block and control to be handed over to the task scheduler.



Hence execution will ping pong between `letter_task` and the Julia REPL in response to the channel being full or empty. However this is not the only way for execution to switch between tasks. Many other function calls can block. In this next section we will look at blocking of I/O function calls.

## Networking

Unlike reading from a file, network communications can have a lot of latency. Hence almost any real world networking code will have to deal with concurrency in some fashion.

We are going to explore a simple client-server example. We got a server listening to connections from clients. Any number of clients can connect, and we want to serve all of them. That means that while we are waiting for a client to send us data, we have to be able to process connections from new clients or data just received from another connected client.

### Server Side

In the example below we have made what we call an echo server. The client sends a text say ``foobar'' and the server sends back exactly the same text but uppercased to ``FOOBAR''.

```

using Sockets

@async begin
    server = listen(2001)

    while true
        sock = accept(server)
        @async while isopen(sock)
            line = readline(sock, keep=true)
            write(sock, uppercase(line))
        end
    end

```

```
end  
end
```

`listen(2001)` means to listen on port 2001 for connections from clients. We get back a `server` object which we instruct to accept connections from clients. This call will block until a client actually connects. At this point a socket object representing that connection is returned.

#### NOTE

What is the purpose of a **port number**? Your computer is uniquely identified on the network by an *IP address*. Normally people connect computers on the internet using hostnames such as `www.nytimes.com` but these have similar relationship to an IP address as an address-book or phone book has to phone numbers.

When you call someone you tap their name on your smartphone, but stored with that name is their actual phone number, which is what the phone network uses.

When communicating with a computer, the IP address is not enough. You may want to communicate with a variety of different programs or services running on that computer. To distinguish between all of these we use *port numbers*.

For instance your web browser wants to communicate with a web server. That is done through port 80. While an FTP client for downloading files would communicate with port 21.

Sockets like files are `I0` objects in Julia. To the underlying operating system they can be quite different. However on a Unix-like system such as Linux or macOS both files and sockets are represented as file descriptors<sup>39</sup> by the operating system. That means most of the same functions that work for files work with sockets. However as a Julia user you can just think of them as `I0` objects which you can read and write to, like any other `I0` object.

Once the socket object is returned from `accept`, we try to read from the socket using `readline`, then we uppercase that line and write it back to the client.

## Client Side

Let us look at what is going on at the client side of this communication. Say you have started running your Julia server in another terminal window and you have fired up a Julia REPL session to communicate with this server. We start by making a connection to port 2001 on *localhost* (the hostname identifying the computer you are currently using):

```
julia> clientside = connect(2001)
```

---

<sup>39</sup>A file descriptor is an abstract handle used to access a file or other IO resources such as a pipe or network socket. They are usually represented by positive integers.

`connect` returns a socket you can use for communications. We store the socket in the `clientside` variable. Or if we are to be pedantic, we bind the name `clientside` to the socket object.

Next we want to write some code, which reads data from the server. However a problem in doing that is that calling `readline` will block until we get a response. That puts us in a catch-22 situation, because then we are not able to write to the socket first. Remember the server only sends a response when we write to it. Hence we wrap the code for reading from the server in an asynchronous task.

```
julia>
@async while isopen(clientside)
    line = readline(clientside, keep=true)
    write(stdout, line)
end
```

This task will run immediately but as soon as it hits our `readline` it will block and return control to the REPL. That allows us to write a `println` which sends a message to the server.

```
julia> println(clientside, "Hello World")
HELLO WORLD
```

We immediately get ``HELLO WORLD'' as a response making it look as if that is what `println` wrote to the console. However the relation is more intricate than that. ``Hello World,'' was sent to the server, which sent back an uppercase version.

Our client async task, currently blocking waiting for a server response, will then be brought back to life. It writes the response from the server to `stdout` with `write(stdout, line)`. The latter is what produces ``HELLO WORLD''.

## Breakdown of Sever Code

The `@async` macro, while very convenient, hides a lot of the details. To understand how the server works, it helps write out the code without using any macros. Without `@async` we need to explicitly create two functions `echo_server` and `writer`.

`echo_server` is the function which waits for client connections and spawns a task for each connected client. Each client gets a `writer_task` based on the `writer` function. The `writer_task` task reads data from the client and responds with text received in uppercase.

```
function echo_server()
    server = listen(2001)
    while true
        sock = accept(server)
        function writer()
            while isopen(sock)
                line = readline(sock, keep=true)
                write(sock, uppercase(line))
            end
        end
    end
end
```

```

        end
writer_task = Task(writer)
schedule(writer_task)
end
end

server_task = Task(echo_server)
schedule(server_task)

```

At the top level we create a task called `server_task` which manages the execution of the `echo_server` function as a coroutine. The task is immediately scheduled, which causes the code to run until it hits `sock = accept(server)` which blocks and causes control to be handed over to the task scheduler. It will pass control over to the Julia REPL task.

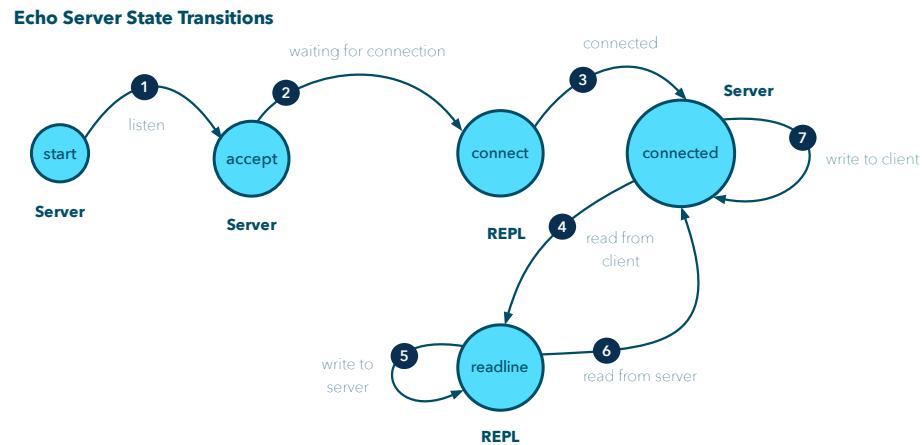


Figure 65: Echo server state transitions

Keeping straight how this interaction works between four different tasks can be difficult. So we will go through the details more thorough. Let us first list the four tasks involved:

- `server_task` listens for connections.
- `writer` is the server's way of dealing with communications with a client.
- `@async` client task listening to data from the server, and writing out server response to the user on `stdout`.
- Julia REPL environment, which is where we actually run the code to make the initial connection as well as where we execute code to send data to the server.

The task switching between all of these can be complex, which is why we have the diagram above to give an overview:

1. `sock = accept(server)` the sever is in a state of listening for client connections.
2. Since this blocks, we transition over to the REPL task.
3. `clientside = connect(2001)` we begin connection to the server. Now the `server_task` is able to resume.

4. `readline(sock, keep=true)` when the server tries to read from socket, it will block and control is once again returned to the REPL.
5. `println(clientside,"Hello World")` at the client we write ``Hello world'' to the server.
6. `line = readline(clientside, keep=true)` Next we try to read a response from the server. But this will block, as the server had no chance to write a response, since the client task has been in control the whole time. This blocking causes control to be handed over to the task scheduler which finds that one of the server tasks `writer_task` can now resume because it was waiting for data from the client.
7. `write(sock, uppercase(line))` the server task writes as upper cased message back to client. Next it will loop around and try to look for next message from client. This will cause it to block, and control is passed on to the REPL @async task.

## Networking API

Now that we have gone through an example, let us summarize what functions are used most frequently in the Julia networking API.

### Server

Functions specific to the server-side. Listening for connections on a port and accepting connections (creating socket for communicating with client).

```
server = listen(portnumber)
socket = accept(server)
```

### Client

Connect to a server at given port number.

```
socket = connect(portnumber)
```

### Common

Used by both client and server to check connection status and closing connection.

```
isopen(io)
close(io)
```

### Output

Writing to a socket.

```
write(io, data)
print(io, number, string, date, ...)
println(io, number, string, date, ...)
```

**Input**

Reading from a socket.

```
data = read(io, type)
readline(io)
readlines(io)
```

# Shell Scripting

- **Directory Navigation** using Julia functions.
- **Filesystem Operations** such as copying, moving or finding files.
- **Navigate** inside files using `seek`, `mark` and `reset`.
- **Unix Pipes**. Working with external processes in similar fashion to Bash<sup>40</sup> and other shell environments.

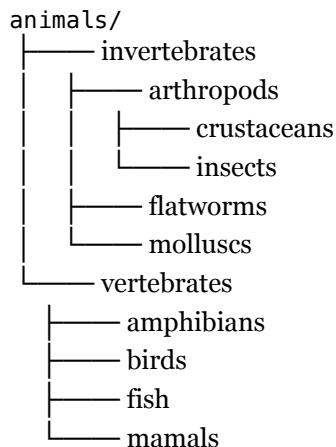
In the previous chapter we dealt with the IO system in general and focused on functions which work on any IO device.

However in this chapter emphasis is on file and directory operations you commonly do from a Unix shell script. Julia happens to be a good language for shell scripting. Tasks done in a Unix shell script can equally well be done with Julia.

To replicate shell functionality, we need to learn more about how we navigate the filesystem, pipe data between running programs (processes) and search inside files with Julia.

## Working with Files and Directories

Before looking into the Julia APIs it can be useful have a short introduction to Unix shell tools. We will use this directory hierarchy to practice filesystem operations:



<sup>40</sup>The Bourne Again Shell, which is a play on the name *Bourne Shell*.

It shows a taxonomy of animals, grouped into various subgroups. You can create this hierarchy yourself either using a graphical file manager or the command line.

Once done, you can use the Unix command line tools to go into the `amphibians` directory and create an empty file called `frog`:

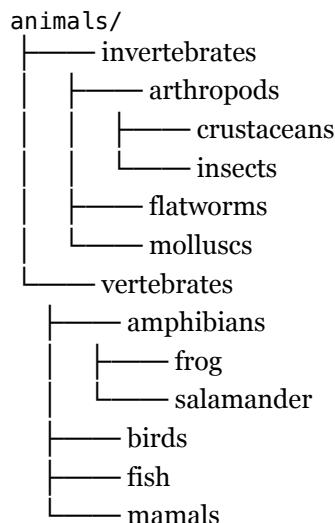
```
$ cd animals/vertebrates/amphibians
$ touch frog
$ cd ..
$ cd ..
$ cd ..
```

We can do exactly the same from the Julia REPL:

```
julia> cd("animals/vertebrates/amphibians")
julia> touch("salamander")
"salamander"

julia> cd("..")
julia> cd("..")
julia> cd("..")
```

After this the `animals` directory will look like this:



However it is cumbersome to use `cd(..)` every time to return to our starting directory. Fortunately there is a variant of `cd` which takes a function `f` as first argument. Function `f` is called after you have switched directory. After `f` has completed, then `cd` will jump back to the original location in the filesystem.

We can demonstrate how this works with the `pwd` function which returns the current working directory. That is the directory affected by your directory and file commands.

```
julia> pwd()
"~"
```

```
julia> cd(pwd, "animals/vertebrates/amphibians")
"~/animals/vertebrates/amphibians"
```

```
julia> pwd()
"~"
```

You can see in this example, that when `cd` calls `pwd` we are in the `animals/vertebrates/amphibians` location, but afterwards we are back to our home directory `~`. Please note I have edited the output of `pwd` for clarity. You will likely see a full path and not `~`.

This example is not very useful, so let us pair `cd` with a more useful function, such as `readdir`. This is the Julia equivalent of the Unix shell command `ls`:

```
julia> readdir("animals")
2-element Array{String,1}:
 "invertebrates"
 "vertebrates"
```

```
julia> readdir("animals/vertebrates/amphibians")
2-element Array{String,1}:
 "frog"
 "salamander"
```

If we combine it with `cd` you can get a better sense of how useful it is to take a function as an argument.

```
julia> cd(readdir, "animals/vertebrates/amphibians")
2-element Array{String,1}:
 "frog"
 "salamander"
```

Remember whenever the first argument is a function we can use the do-end form instead. This makes it easy for us to add more files using the `touch` function.

```
cd("animals/vertebrates/mamals") do
    touch("cow")
    touch("human")
end
```

This can be done more succinct by using the `foreach` function. `foreach` will apply a function on every element in a collection.

```
cd("animals/vertebrates/birds") do
    foreach(touch, ["crow", "seagull", "mockingjay"])
end
```

We don't have any insects yet. They belong under arthropods. If we don't know if that directory already exists we can use:

```
mkpath("animals/invertebrates/arthropods/insects")
```

To add crabs e.g. we need the crustaceans group. To avoid writing the same paths out multiple times one can store it in a variable, and use `joinpath` to

construct new paths.

```
julia> arthropods = "animals/invertebrates/arthropods/"
"animals/invertebrates/arthropods/"

julia> crustaceans = joinpath(arthropods, "crustaceans")
"animals/invertebrates/arthropods/crustaceans"
```

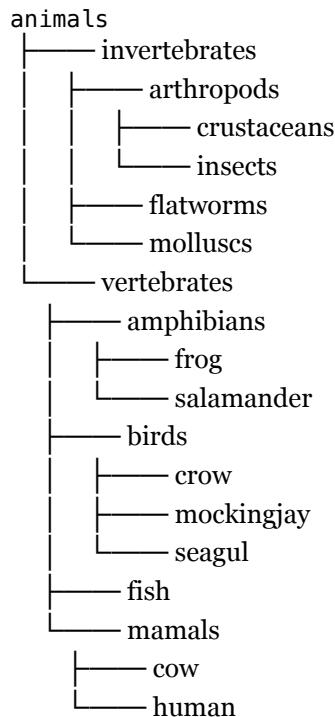
By keeping the path in a variable we can easily reuse it in different circumstances.

```
mkpath(crustaceans)
cd(crustaceans) do
    touch("bever")
end
```

Oops they are not supposed to be there. How to delete?

```
cd(crustaceans) do
    rm("bever")
end
```

After all these file and directory manipulations, we should have a hierarchy of files and directories looking like this:



## Working with Paths

This hierarchy is useful to demonstrate how various functions for working with directory paths behave in Julia. `basename` gives the last entry in a path:

```
julia> basename("animals/vertebrates/mamals/human")
"human"
```

With `dirname` we can get the directory part of the path. For instance, what directory the file `human` is inside:

```
julia> mammals = dirname("animals/vertebrates/mamals/human")
"animals/vertebrates/mamals"
```

As seen before we can join a directory path with a file to create a file path:

```
julia> joinpath(mammals, "human")
"animals/vertebrates/mamals/human"
```

Julia has various function to get the absolute path, relative path and home directory:

```
julia> abspath("animals")
"/Users/erikengheim/animals"
```

```
julia> relpath("animals/vertebrates/../invertebrates")
"animals/invertebrates"
```

```
julia> abspath(homedir())
"/Users/erikengheim"
```

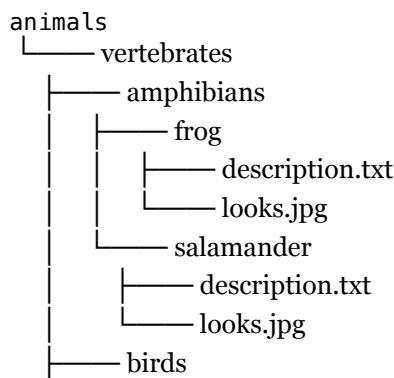
## Reorganizing Assets Example

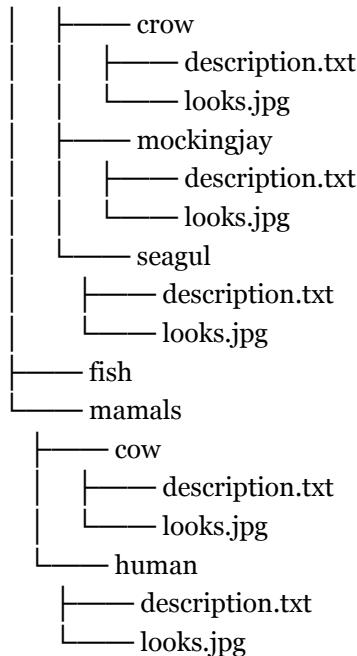
Say we want to make an application where users can click through lists over animals and look at pictures of the animals with a description below. Our current storage structure isn't setup for this. We just have a single file to describe each animal. But really what we want is an image file and a description file.

But what if we already have a large library containing data about animals, not structured this way? How do we reorganize it?

We could do it by hand. Move directories and files around using a file manager. But we are programmers, and we can get the computer to do all the repetitive work of reorganizing files for us. So let us look at a program that can do this.

We want to create a directory structure similar to the one seen below:





Our solution is to split this into two separate problems:

- `replace_animal(animal)` function replaces an animal file with a directory containing a `description.txt` file, describing the animal and a `looks.jpg` image file, showing the animal.
- `visitfiles(fn, root)` which traverses the directory hierarchy looking for files. Each time a file is found the `fn` function is applied to that file.

Thus `visitfiles` will find the files to change, and `replace_animal` will perform the actual transformation.

```

function visitfiles(fn, root::AbstractString)
    if isfile(root)
        fn(root)
        return
    end

    cd(root) do
        for file in readdir()
            visitfiles(fn, file)
        end
    end
end

```

This is a *recursive* function, meaning it calls itself. Recursive functions tend to greatly simplify processing of hierarchical data structures. You will find it used for binary trees, linked lists, graphs and many other data structures. This is how the function works:

1. Check if `root` is a file. If it is, perform our animal replacement. To keep things flexible we avoid hardcoding that. Instead a function `fn` is taken

as an argument, allowing the caller to specify what should happen to each file.

2. If root is not a file, we assume it is a directory and we enter that directory with `cd(root)`.
3. With `readdir()` we get a list of directory entries, which we process in turn.
4. For each entry we want to check if it is a file or a directory to step into. However `visitfiles` already does that, so we can just call it over again. Hence we get a recursion.

We can run this in the REPL to check whether it finds the correct files. If you program on a Mac, you would get the same problem as I have popping up, which is a bunch of `.DS_Store` files.

```
julia> visitfiles("animals") do file
           println(joinpath(pwd(), file))
       end
~/animals/.DS_Store
~/animals/invertebrates/.DS_Store
~/animals/invertebrates/arthropods/.DS_Store
~/animals/vertebrates/.DS_Store
~/animals/vertebrates/amphibians/frog
~/animals/vertebrates/amphibians/salamander
~/animals/vertebrates/birds/crow
~/animals/vertebrates/birds/mockngjay
~/animals/vertebrates/birds/seagul
~/animals/vertebrates/mamals/cow
~/animals/vertebrates/mamals/human
```

Fortunately `visitfiles` provides a convenient way to get rid of these files:

```
visitfiles("animals") do file
    if file == ".DS_Store"
        rm(file)
    end
end
```

---

Next we create a function for replacing animal files.

```
function replace_animal(animal::AbstractString)
    rm(animal)
    mkdir(animal)
    cd(animal) do
        foreach(touch, ["description.txt", "looks.jpg"])
    end
end
```

With any function manipulating the filesystem, it is useful to backup your files first or simply print out the actions which *would* have been performed, rather than actually performing them. In this case the latter is not practical since we actually *need* to create a directory to enter. Here is a walkthrough:

1. We use `rm(animal)` to remove each animal file.
2. `mkdir(animal)` is used to create a directory with a name identical to the file removed.
3. With `cd(animal)` we enter this directory and use `touch` to create the description and image files.

In a more realistic implementation we would likely have copied this image file from somewhere else.

We can do a small-scale test of the function first, to see if it works as expected:

```
julia> touch("foobar")
"foobar"

julia> replace_animal("foobar")

shell> ls foobar
description.txt looks.jpg
```

When one is confident it works, you can visit all the files and perform a replace:

```
julia> visitfiles(replace_animal, "animals")
```

## Navigate Inside Files

Files unlike a lot of other IO devices, have the notion of a position inside the file. You can move to specific positions within the file and record the current position. Later you can revert to a previously recorded position. This is done with the functions:

- `seekstart(io)` moves to the beginning of the file.
- `seekend(io)` move to the end of the file.
- `seek(io, pos)` move position `pos` in file.
- `mark(io)` records current position in file.
- `reset(io)` to position previously marked.

When are these functions useful? Remember how we created rocket engines and tanks by reading CSV files? In this case we processed every line and every line produced an object. In such cases, seeking through a file and marking positions has little value.

However in other cases you work with larger files where there are only particular parts you are interested in or the data isn't clearly structured by lines. For instance when parsing a source code file, a statement doesn't necessarily limit itself to a line.

Let use the construction of the book you are reading as an example. It was originally written in markdown, but there are many flavors of markdown and you may have to switch from one type of markdown to another. In this case most of the text can be preserved but there are particular syntactic structures you want to change.

For instance in Pandoc or Github style markdown, inline math equation are written as:

```
$y = 10x + b$
```

While in Markua style markdown, inline math equations would be written as:

```
`y = 10x + b`$
```

Converting this kind of text can be tricky, because you have to distinguish inline math which uses a single \$ and math blocks which use double dollar signs \$\$, like this:

```
$$y = 10x + b$$
```

The function below takes the name of a file, opens that file, and search for inline math equations, replace them and write the result back to file.

```
function relace_inline_math(filename)
    out = IOBuffer()
    open(filename) do io
        while !eof(io)
            s = readuntil(io, '$')
            write(out, s)

            if s[end] == '`'
                write(out, '$')
                continue
            end

            mark(io)
            s = readuntil(io, '$')

            if isempty(s)
                reset(io)
                s = readuntil(io, raw"$$")
                write(out, '$')
                write(out, s)
                write(out, raw"$$")
                continue
            end

            write(out, '`')
            write(out, s)
            write(out, raw"`$")
        end
    end
    seekstart(out)
    s = read(out, String)
    open(filename, "w") do io
        write(io, s)
    end
end
```

Generally I would recommend against writing functions of this length. However one has to consider complexity. Most lines have very low complexity. The

size is also partially an outcome of the imperative nature of the code: State is repeatedly mutated. More functional oriented code tends to be easier and more natural to write as short functions.

Anyway, let us talk through this function. The bulk of the code is made up of the `while !eof(io)` loop which keeps reading from the file. The loop ends when we have reached EOF (End Of File).

## Simplify Loops with Continue

One trick, which is common to use in big loops like this, is to use the `continue` statement. By using this we can avoid deep nesting of if-else-statements inside loops. When Julia hits a `continue` statement inside a loop it will jump to the beginning of the loop.

The first case, where we use this strategy, is when we try to figure out if we have encountered Markua styled inlined math:

```
if s[end] == ``
    write(out, '$')
    continue
end
```

Say we read this line of text in the markdown document:

```
`y = 10x + b`$
```

In this case we would have:

```
s = "`y = 10x + b`"
```

As you can see checking the last character of `s` would verify whether this was the case. In this case we are lucky because we don't need any further processing. There is no need to continue running the rest of the code, we can just skip to the beginning of the loop again, hence the use of `continue`.

## Readuntil

Most of the code is built around skipping through the text until the next interesting part using the `readuntil` function.

It will be tricky to understand this code without a clarification of how exactly this function works. We will go through some simple examples to demonstrate how it. The `IOBuffer` type gives a practical solution to simulating the interactions with a file. It allows us to treat a simple text string as if it was the contents of a file.

```
julia> buf = IOBuffer("two + four = six");
julia> readuntil(buf, '+')
"two "
julia> readuntil(buf, '+')
" four = six"
```

```
julia> eof(buf)
true

julia> readuntil(buf, '+')
""
```

Notice that after `readuntil` has reached the end of the IO stream object (EOF), it will just keep returning empty strings.

Also observe that the character +, which we read until, gets swallowed but *not included* in the string returned. We *can* include it if we want to.

We use `seekstart` to move to the start of the stream, so we can repeat the reading, this time with `keep=true`, to retain the character we are reading until.

```
julia> seekstart(buf);

julia> readuntil(buf, '+', keep=true)
"two +"

julia> readuntil(buf, '+', keep=true)
" four = six"
```

In our `relace_inline_math` function you can see that we read until the dollar sign without keeping it in the returned string `s`.

```
s = readuntil(io, '$')
```

The reason for this is that we are replacing expressions enclosed with dollar symbols with backticks. Thus we don't need to save the dollar symbols.

## Mark and Reset

After checking if we have encountered an inline math expression which has already been converted, we want to make sure we have not hit a math block enclosed with double dollar signs `$$`.

To deal with this we save the current position in the IO stream, before reading until the next dollar sign. If the string returned is empty, it must mean a dollar sign immediately followed and we are dealing with a math block.

```
mark(io)
s = readuntil(io, '$')

if isempty(s)
    reset(io)
    s = readuntil(io, raw"$$")
    write(out, '$')
    write(out, s)
    write(out, raw"$$")
    continue
end
```

We deal with math blocks by locating the end of the block using `readuntil(io, raw"$$")`. Other than that we write to our output stream exactly what we read. Afterwards we are done and can jump to beginning of loop with `continue`.

## Seekstart

Our output is an `IOBuffer` named `out` which we keep writing our transformed text to. When done processing we want to write the contents of this `IOBuffer` to the same file as the one we read from. That is accomplished with this code segment:

```
seekstart(out)
s = read(out, String)
open(filename, "w") do io
    write(io, s)
end
```

Notice that we use `seekstart(out)` before calling `read`. That is because at this point we are at the end of the IO stream object. Any attempt at reading from it would produce an empty string. There is nothing at the end. We need to move to the beginning and read from there.

## Reading and Writing From External Processes

IO objects are not limited to files or sockets, they can also be `stdin` and `stdout` of external processes. Sometimes a Unix command offers important functionality not present in Julia. Thus we need a way of sending information back and forth between Julia and external processes.

Say we want to search for particular files among our animal directories. E.g. a list of all the `.jpg` files. This could be done with the Unix `find` command:

```
$ find animals -type f -name "*.jpg"
animals/vertebrates/mamals/human/looks.jpg
animals/vertebrates/mamals/cow/looks.jpg
animals/vertebrates/amphibians/frog/looks.jpg
animals/vertebrates/amphibians/salamander/looks.jpg
animals/vertebrates/birds/mockjay/looks.jpg
animals/vertebrates/birds/seagul/looks.jpg
animals/vertebrates/birds/crow/looks.jpg
```

But how can this functionality be utilized from Julia instead of reimplementing the functionality from scratch? Let me begin by showing you a code example.

```
function findfiles(start, glob)
    readlines(`find $start -type f -name $glob`)
end
```

This shows the function in action:

```
julia> files = findfiles("animals", "*.jpg");
julia> files[1:3]
```

```

3-element Array{String,1}:
"animals/vertebrates/mamals/human/looks.jpg"
"animals/vertebrates/mamals/cow/looks.jpg"
"animals/vertebrates/amphibians/frog/looks.jpg"

julia> files = findfiles("animals", "*.txt");

julia> files[end-3:end]
4-element Array{String,1}:
"animals/vertebrates/amphibians/salamander/description.txt"
"animals/vertebrates/birds/mockjay/description.txt"
"animals/vertebrates/birds/seagul/description.txt"
"animals/vertebrates/birds/crow/description.txt"

```

Notice we are able to read the output, from the process, as if it was a regular file. Using `readlines` we even get an array of strings, we can easily slice and dice.

Let us look at a simple example to better understand how this works:

```

julia> dir = "animals"
"animals"

julia> cmd = `ls $dir` 
`ls animals`

julia> typeof(cmd)
Cmd

```

Notice this kind of looks like string interpolation. The value of the variable `dir` gets interpolated with `$dir`. However the backticks cause a `Cmd` object rather than a `String` object to be created.

A `Cmd` object can be opened and read from just like a regular file:

```

julia> io = open(cmd)
Process(`ls animals`, ProcessRunning)

julia> typeof(io)
Base.Process

julia> readline(io)
"invertebrates"

julia> read(io, Char)
'v'

julia> read(io, Char)
'e'

julia> read(io, Char)
'r'

julia> read(io, String)

```

```
"tebrates\n"
julia> close(io)
```

The `io` object returned when we open a `Cmd` object is of type `Process`. Remember the IO type hierarchy we showed before. It shows that `Process` is a concrete type at the bottom of this hierarchy.

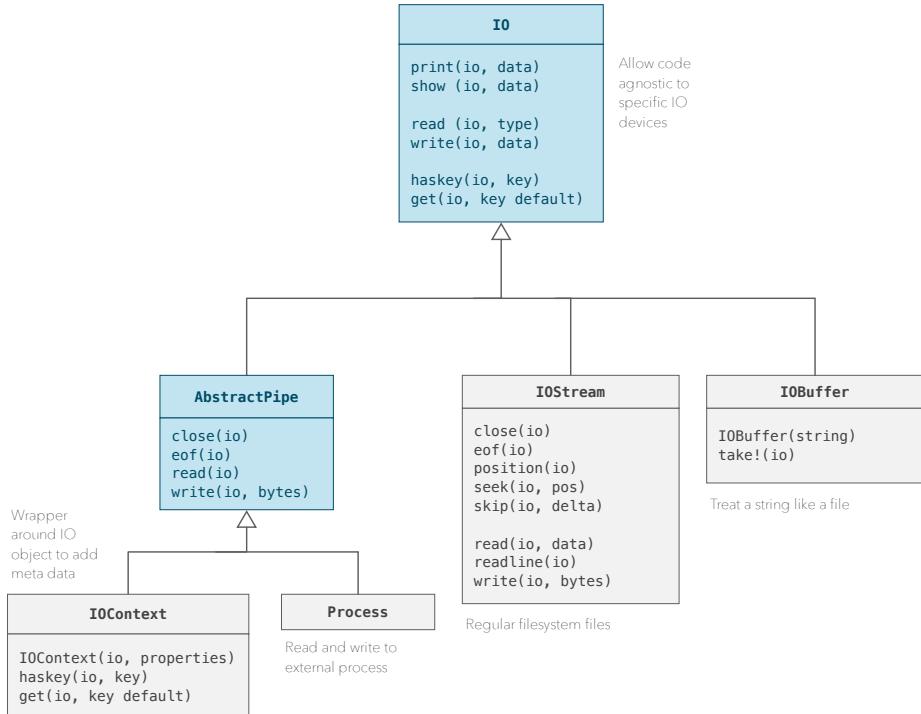


Figure 66: Type hierarchy for IO types.

## Safety of Calling External Commands

If you come from other programming languages such as Python, Ruby or Perl you may have learned that using backticks to run shell commands is a big no-no, however in Julia this is not the case. Let me explain why.

In e.g. Perl and Ruby using backticks will cause a shell process to be spawned, which will interpret the text inside the backticks as if you wrote it in the shell. Hackers can exploit this, by throwing in special characters that the shell interpret in a particular way the writer of the original code had not intended.

However in Julia, what you write between the backticks does not get passed to a Unix shell. Instead Julia has its own parser, that parses this code and creates a `Cmd` object. This makes Julia nicer for shell scripting. You never have to quote variables, and you can easily use ranges and arrays. Let us look at some examples:

```
julia> elements = [3, 8, 10, "hello", false];
```

```
julia> run(`echo $elements`);  
3 8 10 hello false
```

```
julia> range = 1:2:12  
1:2:11
```

```
julia> run(`echo $range`);  
1 3 5 7 9 11
```

In this case we are using `run` instead of `readline`. This is basically the same as running the shell command and getting the output sent to stdout (your terminal window).

## Pipes

In a Unix shell we have an awesome concept called pipes. These allow you to pipe the output of one command into the input of another command. Here is a demonstration of this concept:

```
$ ls animals/vertebrates  
amphibians birds fish mammals  
  
$ ls animals/vertebrates | sort -r  
mammals  
fish  
birds  
amphibians
```

The `ls` command sends a list of filesystem entries to stdout. The `sort` command will take everything you write on the keyboard (stdin) and send it sorted to stdout (console).

However by using the pipe symbol `|` we connect the stdout of `ls` to the stdin of `sort -r`. The `sort` command has no idea that its input is coming from another command.

Pipes gave a lot of flexibility to early Unix systems. Small programs doing a single thing could be chained together using pipes to create new functionality.

We can create these sort of pipes between Julia `Cmd` objects as well:

```
julia> dir = "animals/vertebrates"  
"animals/vertebrates"  
  
julia> pipe = pipeline(`ls $dir`, `sort -r`)  
pipeline(`ls animals/vertebrates`, stdout=`sort -r`)  
  
julia> io = open(pipe);  
  
julia> readline(io)  
"mammals"
```

```
julia> readline(io)
"fish"

julia> readline(io)
"birds"

julia> close(io)
```

In fact we can chain together any number of commands:

```
julia> ls = `ls $dir`
`ls animals/vertebrates`

julia> rsort = `sort -r`;

julia> upper = `tr a-z A-Z`;

julia> run(pipeline(ls, rsort, upper));
MAMALS
FISH
BIRDS
AMPHIBIANS
```

## Environment Variables

Another important part of working with the Unix shell is *environment variables*. These are accessible through a special global dictionary called ENV.

```
julia> ENV["JULIA_EDITOR"]
"mate"

julia> ENV["SHELL"]
"/usr/local/bin/fish"

julia> ENV["TERM"]
"xterm-256color"

julia> ENV["LANG"]
"en_US.UTF-8"
```

Environment variables can be useful in many contexts, not just when working with the shell. For instance the text editor, TextMate which I typically use for programming, has a plugin-system based around:

- Stdin and stdout redirection.
- Environment variables.

A plugin-script basically reads input from stdin and writes output to stdout. In addition information can be conveyed from TextMate to the plugin-script through environment variables. Scripts launched by TextMate will see these environment variables. You don't see them in your regular shell.

This is based on the Unix behavior of how running programs (processes) inherit their environment from their parent process (the processes that spawned them). Here are some of the environment variables used by TextMate:

- TM\_CURRENT\_LINE Current line text. The line the caret is on.
- TM\_CURRENT\_WORD Word at location of caret.
- TM\_SELECTED\_TEXT Currently selected text.
- TM\_LINE\_INDEX Position of caret on line.
- TM\_LINE\_NUMBER Line number at caret position.

To use Julia code in your plugin, you have turn the source code file into an executable script.

## Turning Scripts Into Executables

Unix-like operating systems, such as Linux and macOS, allow you to make a file containing source code executable. You do this by putting a line in the *beginning* of the source code which informs the operating system which interpreter should run the script:

```
#!/usr/local/bin/julia

row = ENV["TM_LINE_NUMBER"]
col = ENV["TM_LINE_INDEX"]
println(row, ", ", col)
```

The first line has to start with a hashbang `#!`, next comes the location of the interpreter to run the script. Because the hash symbol `#` marks the beginning of comments in most script languages including Julia, this line is ignored by the interpreter executing the code.

### **NOTE** Interpreters and Julia

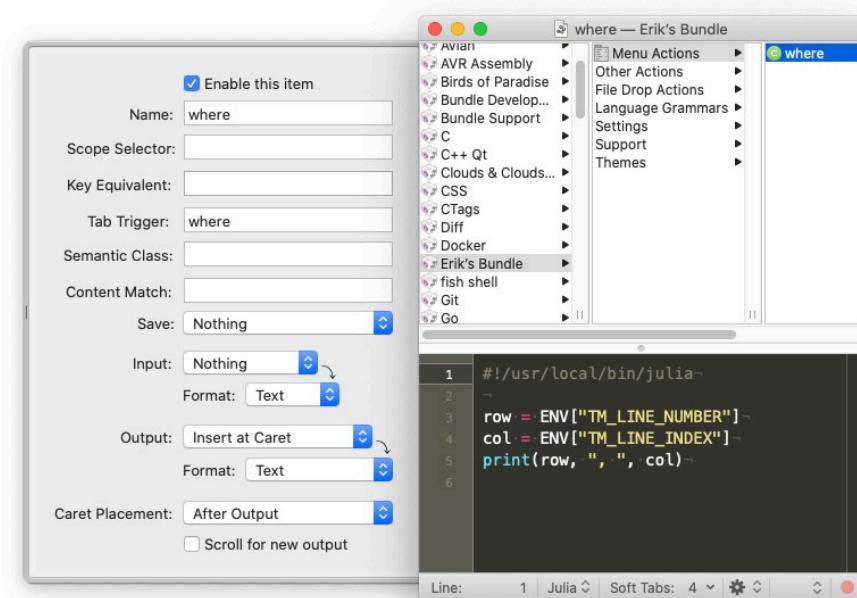
While we refer to Julia as an interpreter here, that is technically incorrect, since Julia is a just-in-time compiled. However running source code directly has traditionally been done by interpreters such as Sh, Bash and Perl. So the hashbang `#!` was made for interpreters.

Another variant commonly used when you don't want to hardcode the location of the interpreter is to use the `env` command:

```
/usr/bin/env julia
```

In this case the OS will use the Julia executable which can be located using the `PATH` environment variable.

Our example script is not very useful, other than to demonstrate how a plugin-system can work. The picture below shows the plugin editor for TextMate, where the code for the plugin has been added.



In TextMate a plugin is referred to as a bundle. As you can see we have made a plugin called ``Erik's Bundle," which is made up of various command you can execute.

A command called `where` has been added. The drawer sticking out on the left describes how the Julia script implementing `where` is called. We could have specified a key combination but in this case, we are using a ``Tab trigger.'' The trigger is specified as ```where`.'' This means you type ```where`'' inside an editor window and hit tab to trigger the command.

This will cause our script to run. TextMate will fill into `TM_LINE_NUMBER` what line number the caret is on. In `TM_LINE_INDEX` it will put the column it is on that line. Then it will execute the script.

The script writes this info to stdout. We have configured the TextMate command to output what it writes to stdout to the text editor window where the caret is located.

The beauty of this system for writing plugins is that you can write your plugins in any language you please. Julia did not exist when TextMate got created, but that doesn't matter. The other benefit is that you don't have to have access to TextMate to test the script. Recreating the environment of TextMate is easy.

Let us do that. Put the code into a file called `where.jl` or whatever you prefer. Next you need to allow this file to be executed, but toggling the execution privilege:

```
$ chmod +x where.jl
```

Before running the script we want to simulate TextMate by setting the `TM_LINE_NUMBER` and `TM_LINE_INDEX`. Let us pretend the caret is on line 8 and

column 3:

```
$ export TM_LINE_NUMBER=8
$ export TM_LINE_INDEX=3
```

How you set environment variables will differ depending on the shell you use. The example above is from the Bash<sup>41</sup> shell as that is widely used. If you used Fish<sup>42</sup> shell instead it would be:

```
$ set -x TM_LINE_NUMBER 8
$ set -x TM_LINE_INDEX 3
```

We can then run the script from the command line and see what output it gives.

```
$ ./where.jl
8, 3
```

Now you may wonder why I picked TextMate as an example, given that it is not a widely used text editor and only works on macOS. It is very simple: Most other text editors have their plugin-system tied to specific programming languages.

## Command Line Arguments

Shell commands can usually take a number of arguments:

```
find animals -type f -name "*.jpg"
```

In this example `animals`, `-type`, `f`, `-name` and `"*.jpg"` are the command line arguments. If you want to create a shell command by writing a Julia script you need a way of obtaining these arguments.

This is done in a very similar way to how we obtained environment variables. Instead of a dictionary we have a global variable named `ARGS`, containing all the arguments.

Here is a simple demonstration of replicating the Unix `cat` command:

```
#!/usr/bin/env julia
for file in ARGS
    s = read(file, String)
    print(s)
end
```

We loop over each element in the `ARGS` array which should contain a file name. Then we open the file and print its output.

Say we put this code inside a file called `cat.jl`, we have to give it execute permission:

```
$ chmod +x cat.jl
```

To test the command I made two files `foo.txt` and `bar.txt`, with a single line in each. You can test with whatever files you like.

---

<sup>41</sup>The Bourne Again Shell, which is a play on the name *Bourne Shell*.

<sup>42</sup>Fish is a less known user-friendly shell.

```
$ ./cat.jl foo.txt
foo text

$ ./cat.jl bar.txt
bar text

$ ./cat.jl foo.txt bar.txt
foo text
bar text
```

The reason why you need to put a `./` in front of the file to execute it is because, we have not placed it in a location stored in the `PATH` environment variable. If you put `cat.jl` in for instance `/usr/local/bin` or another location which the OS typically search for executable files then you would not need the `./` prefix.

## Why Use Julia Instead of Bash?

Shell environments such as Bash, Zsh, Fish and Sh where made for interacting with files and processes. So why would you use Julia instead?

These tools are fine for very short scripts, but as soon as you go much beyond 5-6 lines it will almost always be easier with a proper programming language such as Julia.

For instance if we look at this example of string manipulation in Bash it is not immediately obvious how string manipulation works:

```
s="Hello World"
echo ${s/World/Mars}
echo ${s:3}
```

If we write the same in Julia it is far more obvious what is being done:

```
s = "Hello World"
println(replace(s, "World" => "Mars"))
println(s[4:end])
```

Using Julia you get access to superior handling of arrays, and the ability to do set operations, and write proper functions.

The major downside is that Julia is not installed on every operating system. However Julia programs can be ahead-of-time compiled for easier distribution. We will not cover that in this book as that is a more advanced topic. Anyone interested should explore the `PackageCompiler` package.

# Parametric Types

- **Motivation.** Why does Julia have parametric types at all?
- **Parametric types** are templates for making concrete types.
- **Type parameters** control what kind of types are made from parametric types.
- **Memory fragmentation** causes and how it motivates usage of parametric types?
- **Type constraints** to limit the types which can be used in specific contexts.

You may already be familiar with parametric types if you have used Java, C++, C# or Haskell. Parametric types goes under many different names. In Java they are referred to as generics, while in C++ they would be referred to as template classes.

Julia parametric types is not a new or novel concept in programming. Parametric types have existed for a long time in different programming languages. What is novel however is the use of parametric types in a dynamic language. Parametric types of a dynamic language is a rarity. The concept usually makes no sense in the context of a dynamic languages. However Julia is a language with some unusual traits.

## Why Use Parametric Types?

To explain the point of parametric types, it helps to work through a simple example. Here we are defining a parameterized 2D vector type.

```
struct Vector2D{T}
    x::T
    y::T
end
```

Previously we would have defined `x` and `y` as being of type `Float64` for instance. However in this case they are of type `T`, which is a placeholder for an actual type. No concrete type named `T` exists. `T` is what we call a *type parameter*. Just like functions have parameters, parametric types have *type parameters*. Let me give you a motivation for why you may want that. In Julia we could imagine making bunch of different 2D vector types like this:

```
struct IntVector2D
```

```

x::Int
y::Int
end

struct FloatVector2D
    x::Float64
    y::Float64
end

# Calculate magnitude of vector
norm(v::IntVector2D) = sqrt(v.x^2 + v.y^2)
norm(v::FloatVector2D) = sqrt(v.x^2 + v.y^2)

```

This would be cumbersome and require a lot of boilerplate repetitive code. We are basically repeating the same code over and over again because we are using different types for `x` and `y`. The Python, Ruby or JavaScript developer however would object and say this is completely unnecessary. If you want flexibility in the type for `x` and `y` just leave out the type annotation like this:

```

struct Vector2D
    x
    y
end

```

This is of course possible, but we don't want `x` and `y` to be just any type. They should be some kind of number. Given that you already know about Julia abstract types you probably have an obvious solution to this problem. Just use a type annotation with `Number` like this:

```

struct Vector2D
    x::Number
    y::Number
end

```

This is an entirely reasonable solution when performance does not matter. However Julia was built specifically to support high performance computing. That means the language needs to be able to express code and types in ways which can be easily turned into efficient machine code and which has an optimal layout in memory.

## Boxing of Values

To understand why annotating `x` and `y` as abstract types is bad for performance, we need to understand what boxing is and how it affects performance.

In almost every dynamic language in use, every single value is boxed. What that means is that the actual data is placed inside a sort of generic box. More specifically it means the value gets bundled with meta data which describes the value. Most important is a tag describing the type of the value. Frequently there are other fields to handle memory management. If memory was managed by reference counting, then a reference counting field (`refcount`) would have been included.

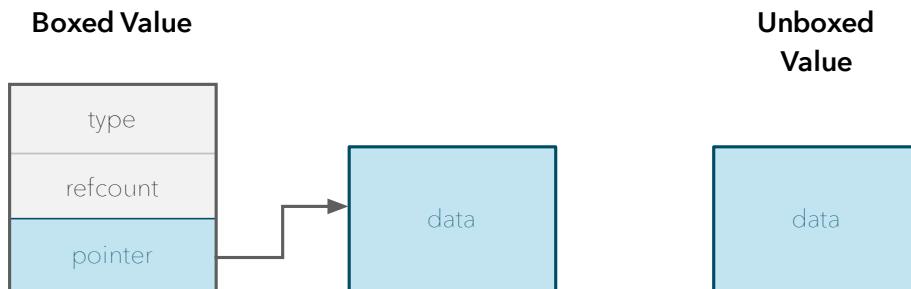


Figure 67: Boxing and unboxing of values

**INFO Reference Counting and Garbage Collection**

In most modern programming languages, memory is managed automatically. Julia is no exception. You don't have to explicitly allocate memory for objects and explicitly free memory as in languages such as C/C++. The system that handles allocating memory and freeing it when it is no longer used is called a *garbage collector*. One popular approach to this is *reference counting*. With reference counting each object keeps track of how many references to each object exists.

If you have an array of millions of boxed objects you get a performance problem for several reasons:

1. The values cannot be stored in contiguous memory<sup>43</sup> because without knowing the exact type, the memory requirements can vary for each object. That means every element in the array must contain type information and a pointer to the actual value somewhere else in memory.
2. If a just-in-time compiler (JIT) is to generate machine code for processing this large array, there is no efficient way of doing it. Every time an element is processed the type has to be looked up and appropriate code to execute has to be selected.

The first case causes memory fragmentation which is bad for CPU cache. Now we are not going to get into cache in detail, but it is something that is useful for every programmer to have a rudimentary knowledge of. The main memory of a modern computer is very slow. If the microprocessor says ``give me the byte at memory address 0x4213," then the CPU will have to wait quite some time for that byte to actually arrive. How long?

## CPU Cache

It takes about 100 nano seconds ( $100 \times 10^{-9}$  seconds) to fetch data from main memory. While a typical CPU instruction can execute in less than 1 nano second. That means while the CPU is waiting for data to arrive from main memory it can

---

<sup>43</sup>Items stored in *contiguous memory*, is stored adjacent to each other with no gaps.

do more than 100 instructions. This potentially creates a huge bottleneck for performance.

Thus to avoid that the microprocessor sits idle, there is a much smaller, but much faster memory sitting inside the CPU which tends to be filled with the data currently being used and the code currently being run. This memory is called the cache. Fetching something from cache will typically take 0.5 nano seconds. Thus we can get the next instruction before we have finished executing the current one.

But I still have not explained why contiguous memory is an important consideration for CPU cache and performance.

Here is the kicker: There is no difference in time fetching 1 byte and fetching 64 consecutive bytes from main memory (cache line). Thus if data you are working on is placed contagiously in memory, asking for one byte, will also give you the 63 next bytes you want to work on. However if memory is fragmented and data is spread all over the place this assumption falls apart.

Boxing tends to make memory fragmentation worse. However if we create composite types only made up of concrete types, then Julia knows exactly how much space each object takes.

A 64-bit floating point number takes 8 bytes ( $64/8 = 8$ ). Hence a `Vector2D{Float64}` would require take 16 bytes. If I wrote `Array{Vector2D{Float64}}` in Julia to create an array of 1 million 2D vectors, Julia would be able to figure out that this requires 16 million bytes of memory. Julia could allocate that in one operation and put all the 2D vectors in contiguous memory. It would know that at every 16 byte interval it can find the next 2D vector object.

The reason Julia can be absolutely certain of this is that Julia types are not allowed to change and there can be no subtypes of a concrete type. If you say the `x` is a `Float64`, then its space requirements is set in stone. The requirements are known and they cannot change. However if `x` was a `Number` then there would be no way of knowing the exact space requirements. Finding out the requirements once would not help either as the developer would be free to change the concrete type stored at any time later.

## Type Parameter Constraints

In this definition of `Vector2D` the type parameter `T` is entirely unconstrained when performing vector operations such as the dot product and cross product.

```
struct Vector2D{T}
    x::T
    y::T
end

dot(u::Vector2D, v::Vector2D) = u.x*v.x + u.y*v.y
cross(u::Vector2D, v::Vector2D) = u.x*v.y - u.y*v.x
```

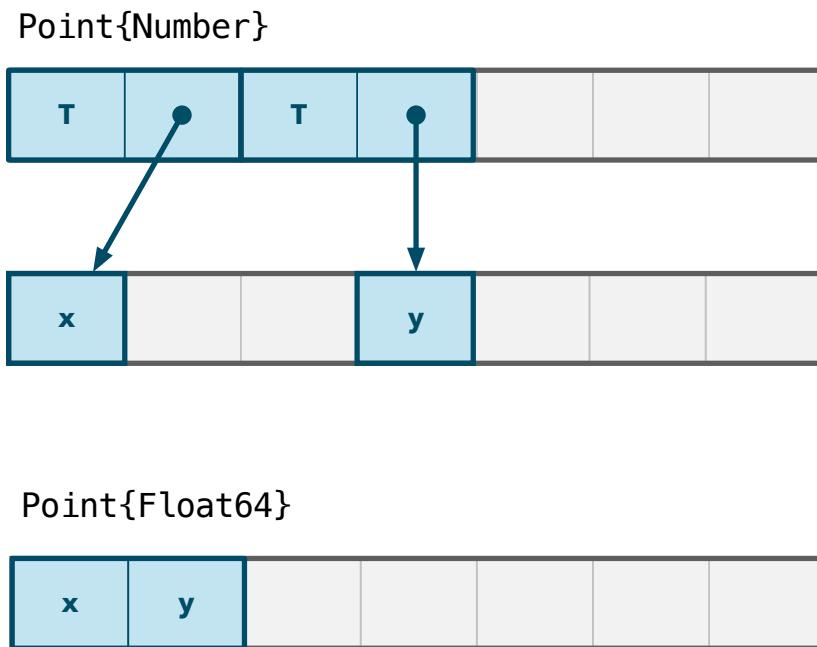


Figure 68: For `Point{Number}` `x` and `y` coordinates must be boxed values. However for `Point{Float64}` boxing is not needed.

What does that mean, and why is that important? Let us look at some problems with unconstrained type parameters.

```
julia> u = Vector2D{Char}('A', 'B')
Vector2D{Char}('A', 'B')

julia> v = Vector2D{Int8}(2, 3)
Vector2D{Int64}(2, 3)

julia> dot(u, v)
ERROR: MethodError: no method matching *(::Char, ::Int64)
```

We were able to create a 2D vector of characters which really should not be possible. That we did something wrong is not caught before we try to run `dot(u, v)` and even then the problem is caught late. It is not upon making the call itself but when Julia attempts to perform `u.x * v.x` inside the implementation of `dot(u, v)`. Multiplying a character with a number is not possible.

If we replace the `u` vector with a number based one you can see it works fine. `u` and `v` does not even need to use the same number type as they respective type parameter.

```
julia> u = Vector2D{Float64}(3.0, 1.0)
Vector2D{Float64}(3.0, 1.0)
```

```
julia> dot(u, v)
9.0
```

But we work with a dynamic language so we cannot catch problems using a compiler before it is run apart from using a linter. However that does not mean we don't care about *when* problems are caught. When running a program and something goes wrong, it is much easier to figure out the problem if we catch the problem at the source as early as possible.

Hence we want Julia to cause an exception when an attempt is made to create a 2D vector of characters. We do this by introducing a constraint on the type parameter.

```
struct Vector2D{T<:Number}
    x::T
    y::T
end
```

With this change attempting to create a 2D array of characters will throw a `TypeError` exception.

```
julia> u = Vector2D{Char}('A', 'B')
ERROR: TypeError: in Vector2D, in T, expected T<:Number, got Type{Char}
```

Constraints can be provided in many different ways. We can also put constraints on functions.

In this definition I am saying that the type parameter to the `u` and `v` arguments have to be a subtype of `Number`.

```
function dot(u::Vector2D{<:Number}, v::Vector2D{<:Number})
    u.x*v.x + u.y*v.y
end

function cross(u::Vector2D{<:Number}, v::Vector2D{<:Number})
    u.x*v.y - u.y*v.x
end
```

In our case this is superfluous because you cannot make `Vector2D` objects where the type parameter is something different from a number.

Given that we know the type parameter is a number, this may make more sense:

```
function dot(u::Vector2D{T}, v::Vector2D{T}) where T
    u.x*v.x + u.y*v.y
end

function cross(u::Vector2D{T}, v::Vector2D{T}) where T
    u.x*v.y - u.y*v.x
end
```

In this case we are saying that the type parameter of `u` has to be the same as the one for `v`. We can demonstrate this in the REPL.

```
julia> dot(Vector2D{Int8}(2, 4), Vector2D{Int16}(3, 1))
ERROR: MethodError: no method matching dot(::Vector2D{Int8}, ::Vector2D{Int16})
```

```
Closest candidates are:
  dot(::Vector2D{T}, ::Vector2D{T}) where T
```

```
julia> dot(Vector2D{Int8}(2, 4), Vector2D{Int8}(3, 1))
10
```

Please note that we have written the type parameter out explicitly. However you seldom have to do that due to Julia's ability to do type inference (the ability to guess a type).

#### **NOTE Julia Type Inference**

Many statically typed languages use type inference. In this case the compiler will infer the type and report any problem at compile time, before the program is run. However this is quite different from what Julia does as compilation in Julia happens at runtime. This means type inference also happens at runtime.

Let me show some examples of equivalent expressions, where we rely on type inference:

```
julia> xs = Float64[1.0, 4.0, 2.0]
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

```
julia> xs = [1.0, 4.0, 2.0]
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

```
julia> eltype(xs)
Float64
```

```
julia> Vector2D{Int64}(1, 2)
Vector2D{Int64}(1, 2)
```

```
julia> Vector2D(1, 2)
Vector2D{Int64}(1, 2)
```

```
julia> Vector2D(3.0, 2.0)
Vector2D{Float64}(3.0, 2.0)
```

```
julia> Vector2D{Int8}(32, 64)
Vector2D{Int8}(32, 64)
```

```
julia> Vector2D(Int8(32), Int8(64))
Vector2D{Int8}(32, 64)
```

Let us do some more cases. Below we are not just saying that  $u$  and  $v$  vectors have to use the same type parameter  $T$ , but we *also* add the constraint that the type parameter  $T$  has to be a subtype of `Number`.

```
function dot(u::Vector2D{T}, v::Vector2D{T}) where T <: Number
    u.x*v.x + u.y*v.y
end

function cross(u::Vector2D{T}, v::Vector2D{T}) where T <: Number
    u.x*v.y - u.y*v.x
end
```

This is useful if we allowed non-number based 2D vectors to be made. In this case we have to be specific about what type of vectors you are allowed to perform dot and cross product on.

If you don't want to have any constraints, you could write that as:

```
function dot(u::Vector2D{T}, v::Vector2D{S}) where {T, S}
    u.x*v.x + u.y*v.y
end

function cross(u::Vector2D{T}, v::Vector2D{S}) where {T, S}
    u.x*v.y - u.y*v.x
end
```

In this case we are saying that  $u$  uses a different  $T$  type parameter from  $v$  which used  $S$ . No restriction on what type  $T$  and  $S$  can be is made. Writing out type parameters like this which makes no restriction is seems unnecessary and superfluous which is why Julia allows you to accomplish exactly the same by writing:

```
function dot(u::Vector2D, v::Vector2D)
    u.x*v.x + u.y*v.y
end

function cross(u::Vector2D, v::Vector2D)
    u.x*v.y - u.y*v.x
end
```

If you don't specify a type parameter for a parameterized type, Julia assumes no constraint is placed upon it. Since we require 2D vectors to be number based this is actually a sensible choice in this case.

## Parameterized Ordered Dictionary

Previously we covered Implementing an XML Attribute List in our chapter on Object Collections. We created a collection `OrdDict` storing keys and values ordered by insertion, which could be accessed through a dictionary interface.

```
mutable struct OrdDict <: AbstractDict{Symbol, String}
    items::Vector{Pair{Symbol, String}}
end
```

```
OrdDict()      = OrdDict(Pair{Symbol, String}[])
OrdDict(items...) = OrdDict(collect(items))
```

`OrdDict` is currently limited to dealing with `Symbol` keys and `String` values. We want this collection to be able to handle any kind of key-value pair, and by turning it into a parameterized type, we can do that.

```
mutable struct OrdDict{K,V} <: AbstractDict{K,V}
    items::Vector{Pair{K, V}}
end

OrdDict{K,V}() where {K,V} = OrdDict(Pair{K, V}[])
OrdDict(items...) = OrdDict(collect(items))
```

We parameterize `OrdDict` using two type parameters `K` and `V`. We are using `K` as the placeholder for the type of the key and `V` as the placeholder for the type of the values.

Whenever I write `dict = OrdDict{String, Int}()` then Julia is informed that the `K` type should be replaced with `String` and the `V` type with `Int`. However as we have seen earlier you don't normally have to spell this out, as Julia is good at inferring what the type parameters should be.

```
julia> pairs = ["two" => 2, "four" => 4]
2-element Array{Pair{String,Int64},1}:
 "two" => 2
 "four" => 4

julia> typeof(pairs)
Array{Pair{String,Int64},1}

julia> dict = OrdDict(pairs)
OrdDict{String,Int64} with 2 entries:
 "two"  => 2
 "four" => 4

julia> typeof(dict)
OrdDict{String,Int64}
```

Notice Julia looks at the types of the provided `pairs` array and use that to infer what `K` and `V` should be.

When we create an empty collection we don't have this luxury, as there is no input to infer types from.

## Iteration

Interestingly to support iteration we don't need to change the code we wrote in our first version which was not parameterized. As before we simply forward to the implementation used by the `items` vector.

```
length(a::OrdDict)      = length(a.items)
isempty(a::OrdDict)     = isempty(a.items)
```

```
iterate(a::OrdDict)    = iterate(a.items)
iterate(a::OrdDict, i) = iterate(a.items, i)
```

Why does this work without modification? Because `OrdDict` is shorthand for `OrdDict{K, V}`, where `K` and `V` has no constraints defined. And in fact we don't need to specify any constraints in any of these method implementations. Why not? Because the keys and values can be anything.

We can easily verify that the dictionary behaves as expected.

```
julia> item = OrdDict(:row=> "1", :column=> "2")
OrdDict(Pair{Symbol, String}[:row=>"1", :column=>"2"])

julia> for (k,v) in item
           println("key: ", k, " value: ", v)
       end

key: row value: 1
key: column value: 2

julia> collect(item)
2-element Array{Any,1}:
 :row => "1"
 :column => "2"

julia> map(first, item)
2-element Array{Symbol,1}:
 :row
 :column

julia> map(last, item)
2-element Array{String,1}:
 "1"
 "2"
```

## Lookup Values by Key and Index

For key lookups we need to slightly change the original code. In this case the type parameter names `K` and `V` are needed, because the key we are searching for has to be of the same type as the keys in our dictionary.

```
function getindex(a::OrdDict{K, V}, key::K) where {K, V}
    for item in a.items
        if first(item) == key
            return last(item)
        end
    end
    throw(KeyError(key))
end
```

Index based lookup does not require any code changes, because the implementation does not rely on knowing the type of the key or value.

```
function getindex(a::OrdDict, index::Integer)
    a.items[index]
end
```

When implementing `setindex!` which allows us to set values in our collection based on key, we need to use both type parameters `K` and `V`. So in this case we cannot skip them. You know you need them whenever they get repeated for multiple arguments. E.g. the `K` type parameter is used both for the `a` argument and the key argument.

```
function setindex!(a::OrdDict{K, V}, value::V, key::K) where {K, V}
    for (i, item) in enumerate(a.items)
        if first(item) == key
            a.items[i] = key => value
            return
        end
    end
    push!(a.items, key => value)
end
```

Also if we want to support an array like interface and support pushing key-value pairs to the back of the collection we need to specify both type parameters.

```
function push!(a::OrdDict{K, V}, x::Pair{K, V}) where {K, V}
    push!(a.items, x)
end
```

This enforces that you cannot add a pair to the ordered dictionary, which isn't made up of similarly typed objects. If the key is `Char` and the value is `Int` in the ordered dictionary, then the first value in the `x` pair also has to be a `Char` and the second value has to be an `Int`.

## Checking for Items and Removing Them

For completeness we will add code to check for presence of a key and for deleting an item by key. However by now it should be clear to you how you make these modifications to support parameterized types.

```
function haskey(a::OrdDict{K, V}, key::K) where {K, V}
    for item in a.items
        if first(item) == key
            return true
        end
    end
    false
end

function delete!(a::OrdDict{K, V}, key::K) where {K, V}
    for (i, item) in enumerate(a.items)
        if first(item) == key
            deleteat!(a.items, i)
        end
    end
end
```

```
end
a
end
```

## Pitfalls

In my experience it is easy to use parameterized types wrong. In this section we are going to explore some common misconceptions and mistakes which are easy to make.

We will look at some examples using a `Point` type, specifying a location in a 2D world with `x` and `y` coordinates.

```
struct Point{T}
    x::T
    y::T
end
```

## Subtyping

When thinking about relations between parameterized types you have to keep in mind that the Julia type system was designed to enable high performance. That is why types are immutable and why you cannot create a subtype of a concrete type.

Let us look at a case which may confuse you:

```
julia> Int <: Integer
true

julia> Point{Int} <: Point{Integer}
false
```

Why is `Int` a subtype of `Integer` but `Point{Int}` is not a subtype of `Point{Integer}`? The simple reason is that you cannot create an instance of `Integer`. It is an abstract type. However you can create an instance of `Point{Integer}`. It is a concrete type just like one of our first `Point` versions where `x` and `y` had the type `Number`. Even if `Point` contained fields annotated with an abstract type, did not mean that the `Point` composite type was itself abstract.

This is the simple rule that must always be followed in Julia:

A super type must always be abstract. It cannot be a concrete type which you can instantiate.

A type such as `Point{T}` is abstract as it does not specify a particular type, unless you have defined a type named `T`. Thus we need a way to specify that `T` is a place holder type and not a concrete type. In Julia you do that with `where T`. Below we have some examples demonstrating this.

```
julia> Point{Int} <: Point{T}
ERROR: UndefVarError: T not defined

julia> Point{Int} <: Point{T} where T
true

julia> Point{Int} <: Point{T} where T <: Integer
true

julia> Point{Int} <: Point{<:Integer}
true
```

Let us cover each of these cases:

1. This does not work because Julia does not know whether it should assume `T` is an actual concrete type or a placeholder type. Hence the error message.
2. This works fine because we are telling Julia that `T` is a placeholder type. That also means `Point{T}` is not a concrete type which can be instantiated. Hence it is possible for `Point{Int}` to be a subtype of it. We placed no constraint on `T` which `Int` does not match.
3. Here we added the constraint on `T` that it must be a subtype of `Integer`. That works fine, because `Int` is a subtype of `Integer`.
4. This is a shorthand form of the former. There is no need to use the letter `T` when the constraint only occurs in one place. Remember we have earlier used `T` to specify that two arguments must use the same type parameter.

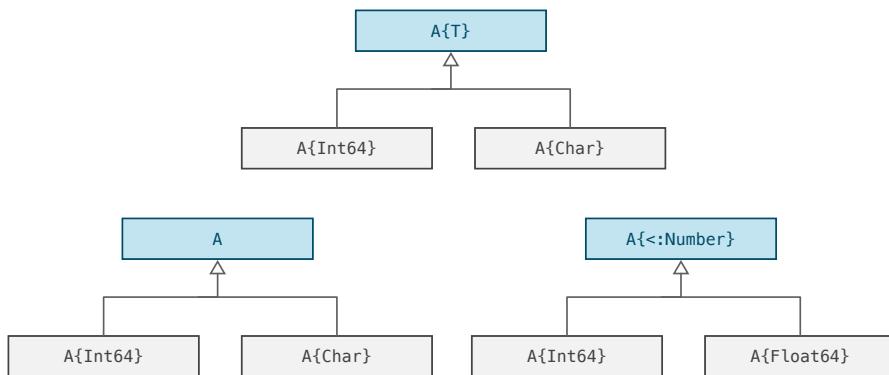


Figure 69: Sub-typing with parameterized types. Think of the blue boxes as supersets containing a number of concrete types matching type parameter constraints.

Let us cover some of these cases a bit more in detail. There is nothing magical about the name `T`. We could have used any name. It is not the name, but the `where` clause which makes it a type placeholder.

We can illustrate this with some simple example:

```
julia> Point{Int} <: Point{Char} where Char
```

```
true
```

This looks weird because we know that `Int` is not a subtype of `Char`. However when adding `where Char`, then `Char` no longer refers to a concrete type. Here is another example illustrating this:

```
julia> Int <: Char
false
```

```
julia> Int <: Char where Char
true
```

## Memory Layout

Why can't concrete types be subtypes of other concrete types? Looking at memory layout of arrays of different types gives us a clue.

Let us do a thought experiment, where we assume the `Point{Int8}` can be a subtype of `Point{Number}`. If this was true, then what would imply that a `Point{Int8}` value could be inserted into a `Vector{Point{Number}}` array. Why? Because being a subtype implies that the subtype is interchangeable with the supertype.

According to this thought experiment, I should then be able to write this code:

```
supa = [Point{Number}(3, 4), Point{Number}(2, 1), Point{Number}(-4, 5)]
supa[1] = Point{Int8}(7, 11)
```

Allowing this wouldn't be smart from a performance perspective. Each `Point{Int8}` value only needs 2 bytes. One byte for the `x` and `y` coordinates each.

But if we were to allow any subtype to be put in there, we need to box it. How else will Julia be able to figure out exactly what the memory layout of each point?

And as we have discussed earlier, if a value is boxed, it needs more memory. We need to store things like the type, memory allocation data etc.

Contrast this with using an array defined specifically to have elements of type `Point{Int8}`. If you know every element is of this type, then you can lift the type info out and store it on the array object itself, rather than on every point element. This saves us a lot of space.

The 3-element array from our example can then allocate with a mere  $3 \times 2 = 6$  bytes.

Ok, to be fair it will take a bit more space since the array has a header consuming 40 bytes. Hence for small arrays this optimization doesn't matter. But as arrays grow larger it will make a big difference.

We can explore memory usage of different types in the REPL:

```
julia> p = Point{Int8}(2, 3)
Point{Int8}(2, 3)
```

```
julia> q = Point{Number}(Int8(2), Int8(3))
Point{Number}(2, 3)

julia> r = Point{Number}(Int64(2), Int64(3))
Point{Number}(2, 3)

julia> sizeof(p)
2

julia> sizeof(q)
16

julia> sizeof(r)
16

julia> varinfo(r"^\$|q|r\$")
  name      size summary
  ____  _____  _____
    p      2 bytes Point{Int8}
    q     18 bytes Point{Number}
    r     32 bytes Point{Number}
```

Do you see that despite the fact that both `q` and `p` are representing a point made up of two 8-bit `x` and `y` coordinates they don't take the same space? `q` consumes a total of 18 bytes while `p` requires just 2 bytes. Why is that?

`sizeof(q)` tells us that `q` the `Point{Number}` type requires 16 bytes excluding the size of any objects it may point at. `r` has exactly the same size, despite holding two 64 bit integers. The reason is that because the type parameter is abstract, we must use boxing. Hence in both `q` and `r` we have two pointers to the actual number values. These pointers each hold a 64 bit memory address, which is why they consume  $2 \times (64/8) = 16$  bytes of memory. Adding the two bytes containing the `x` and `y` for `q` gives a total of 18 bytes.

With `r` we have two 64 bit pointers each pointing to a 64 bit integer. Hence we have a total of four 64 bit values which totals 32 bytes.

Bottom line is that `q` and `p` does not have the same size despite holding the same kind of data. So for instance if we have an array of `Point{Number}`, then each slot in the array takes 16 bytes. Now imagine trying to put a `Point{Int128}` into one of these slots. It takes 32 bytes. That is too big to fit into a 16 byte slot. If you tried to cram it in, it would overwrite the next value.

This is also one of the reasons Julia is not an object-oriented language. Being able to create subtypes of concrete types would kill performance because you would be forced to box values all the time.

## Appropriate Array Type

What we have covered thus far should help you answer this challenge. Why is it bad to define the type of the input to a function like this?

```
addup(xs::Vector{Number}) = reduce(+, xs)
```

Let us try it in the Julia REPL and see what happens.

```
julia> addup([1, 2, 3])
ERROR: MethodError: no method matching addup(::Array{Int64,1})
Closest candidates are:
  addup(::Array{Number,1})
```

Yes that does not work because `Vector{Int64}` is not a subtype of `Vector{Number}`. You could make it work by defining the array as `Vector{Number}` type like this:

```
julia> addup(Number[1, 2, 3])
6
```

But this is not desirable. You don't want to require users of your function to change their input to the very specific array type you are using. Based on what we have covered thus far, it should be clear that the proper solution is to define it as:

```
addup(xs::Vector{<:Number}) = reduce(+, xs)
```

This will make both cases work.

```
julia> addup([1, 2, 3])
6
```

```
julia> addup(Number[1, 2, 3])
6
```

Let us do another common mistake. You are used to writing function taking string arguments as taking arguments of type `AbstractString`.

```
concat(strs::Vector{AbstractString}) = join(strs)
```

This will of course fail, because `Vector{String}` is not a subtype of `Vector{AbstractString}`.

```
julia> concat(["iron", "man"])
ERROR: MethodError: no method matching concat(::Array{String,1})
Closest candidates are:
  concat(::Array{AbstractString,1})
```

We need to define it as taking an abstract type `Vector{<:AbstractString}` as argument.

```
concat(strs::Vector{<:AbstractString}) = join(strs)
```

## Impractical Constraints

Be wary of creating unnecessarily strict constraints on your input.

```
locate(xs::Vector{T}, x::T) where T<:Number = findfirst(==(x), xs)
```

It may seem good to make the elements in the array the same as type as the type of the element you are searching for. But it can easily get you into trouble.

```
julia> locate([5, 8, 4, 1], 4)
3

julia> locate(Int8[5, 8, 4, 1], 4)
ERROR: MethodError: no method matching locate(::Array{Int8,1}, ::Int64)
Closest candidates are:
  locate(::Array{T<:Number,1}, ::T<:Number) where T<:Number
```

It is better to make the element type and type of object searched for more independent from each other.

```
locate(xs::Vector{<:Number}, x::Number) = findfirst==(x), xs)
```

This means the element type of `xs` and the type of `x` does not have to be the same type, but since they are both numbers they can be used together.



# Testing

- **Unit Tests** tests a unit, the smallest piece of code that can be logically separated from the rest of the system.
- **Integration Tests** test to see if multiple units work properly together.
- **Regression Test**. The act of performing all your tests after a modification to make sure everything still works as expected.
- **TDD**. Test Driven Development is a popular approach to testing in object-oriented programming. We contrast it with *REPL Driven Development*.
- **Test Sets** are the fundamental building block of tests in Julia.

Throughout most of this books we have focused on the Julia language itself. How you solve concrete programming problems.

Now we will explore tools and techniques for removing defects in your code and improve the quality of your software. To aid us in this endeavor we will look at an existing Julia project and add tests to it. In subsequent chapters we will add logging and explore running this code through a debugger.

The project is called LittleManComputer.jl, located at:

<https://github.com/ordovician/LittleManComputer.jl>

It is an implementation of a simple computer simulator. Let me describe this simulator conceptually before we jump into the Julia code.

## The Little Man Computer

The Little Man Computer(LMC) is a very simply microprocessor and computer system designed to teach beginners assembly programming<sup>44</sup>. It keeps things simple by operating on decimal numbers, rather than binary numbers. The computer has 100 memory cells, numbered from 0-99.

Each cell can hold a 3-digit decimal number. The stored numbers can represent either:

- Instructions or operations to perform.
- Data to perform operations on.

---

<sup>44</sup>Assembly code has a one to one correspondence with the native machine code instructions of the microprocessor unlike Julia code, where one function could correspond to hundreds of machine code instructions.

The CPU has a single *accumulator* which is used to perform arithmetic and comparison operations. There is also an *input* and output box, where you can read user *input* and write *output* to the user.

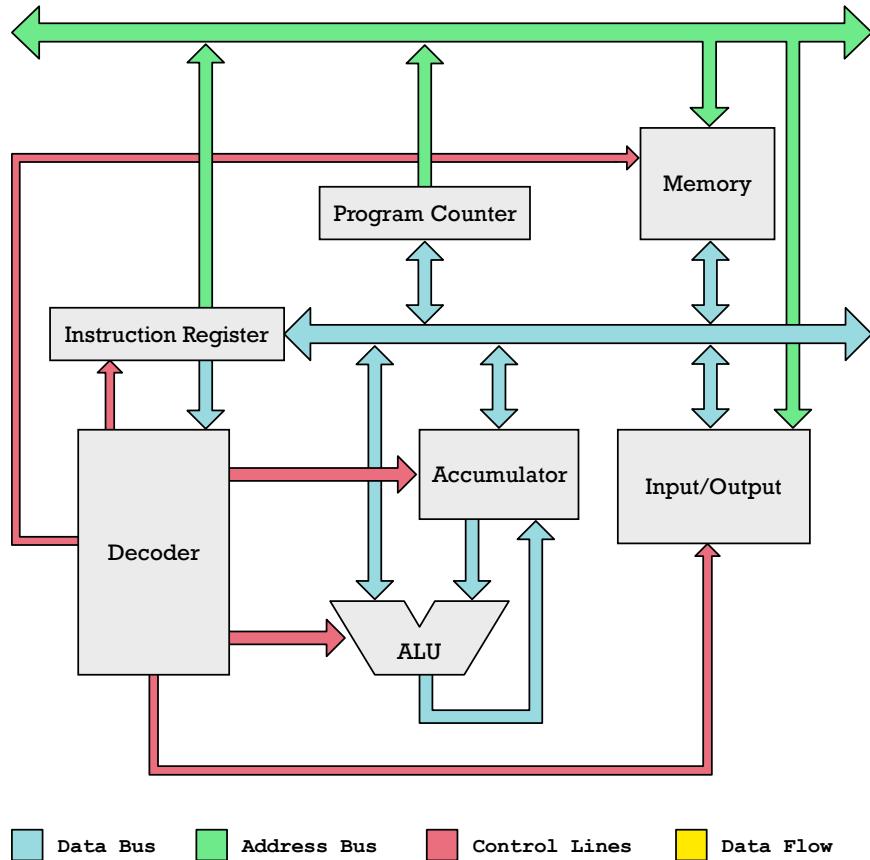


Figure 70: Diagram of the different parts of the Little Man Computer. Data flows along blue arrows. Red lines enable and disable different parts of computer.

Every computer has a microprocessor (CPU) with a fixed instruction set. Unlike Julia where we can add new functions, the microprocessor has a fixed number of unique operations it can perform.

The LMC computer has a minimal set of ten different instructions. A real CPU, such as an Intel x86 microprocessor, has over 1500 different instructions. But there are real CPUs with far less, such as the 6502, popular in home computers back in the 1980s. It has 56 different instructions.

Let us examine the instruction set of the LMC processor. Each instruction is a simple 3-digit number. However to make programming easier we represent these numbers using short abbreviations called *mnemonics*. Here is a list of all the LMC instructions with the mnemonics and their machine code encoding (number representation):

- ADD 1xx **ADD** number at address xx in memory to the accumulator.
- SUB 2xx **SUBtract** number at address xx from the accumulator. Store result in the accumulator.
- STA 3xx **STore** Accumulator content at address xx in memory.
- LDA 5xx **LoAD** Accumulator with contents at address xx in memory.
- BRA 6xx **BRAnch** to location xx in program.
- BRZ 7xx **BRanch** if the accumulator is **Zero**.
- BRP 8xx **BRanch** if the accumulator is **Positive** (greater than zero).
- INP 901 **INPut** number to the accumulator.
- OUT 902 **OUTput** accumulator content.
- HLT 000 **HaLT**/stop the program.

Let me help you interpret this list. Instructions are split into two parts:

- **Opcode**, which is what kind of operation you are performing such as *add*, *subtract* or *branch* (jump to new location in the program). This is the first digit. This of this as similar to a function name.
- **Operands** which specify what you are performing the operations on. Compare this to function arguments.

Let us look at the first two entries in the list ADD and SUB. The first one has the number 1 as opcode, while SUB has 2 as opcode. Next comes the single operand consuming two digits. This is represented by xx. Why two digits? Because the value we are adding to the accumulator is stored in a memory address. Memory cells have addresses from 0 to 99. Hence we need two digits to represent any valid address.

Based on this description, can you guess what the instructions 142 and 231 means?

- 142 means carry out opcode 1 on data found at address 42. Opcode 1 means ADD. Hence we are adding contents of address 42 to the accumulator.
- 231 subtracts the number found in memory cell 31 from the accumulator.

## Example of LMC Programs

Here is a simple example of an LMC program which fetches two numbers and adds them:

```

INP
STA total
INP
ADD total
OUT
HLT
total DAT 0

```

The numbers are fetched from input using the INP instruction. Because there is only one accumulator register, it must make a temporary storage in the memory cell labeled **total**. At **total** we got the DAT 0 directive, which tells the assembler

to initialize the cell in this location to 0. In fact we could have omitted the 0, since this is the default value if none is specified.

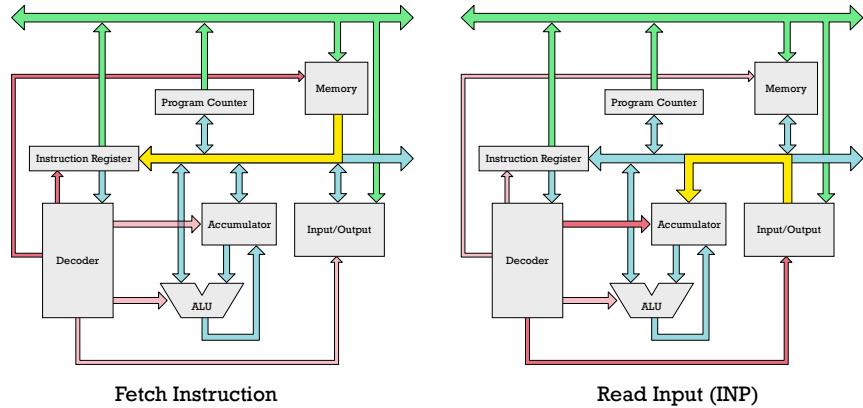


Figure 71: Yellow arrows showing how data flows when fetching an instruction from memory and when reading data from input.

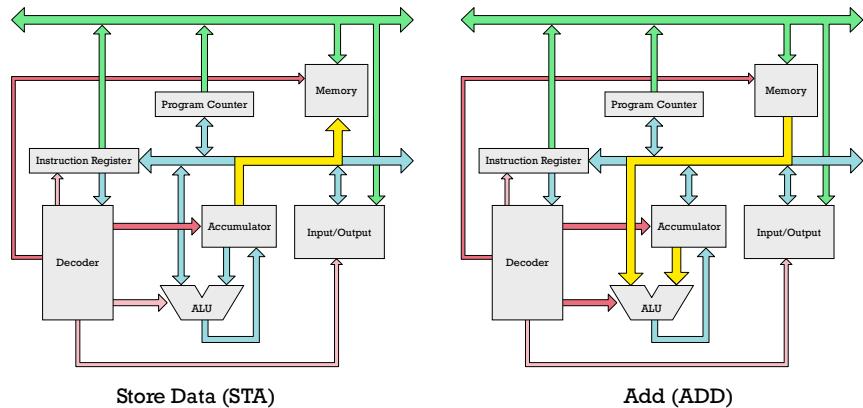
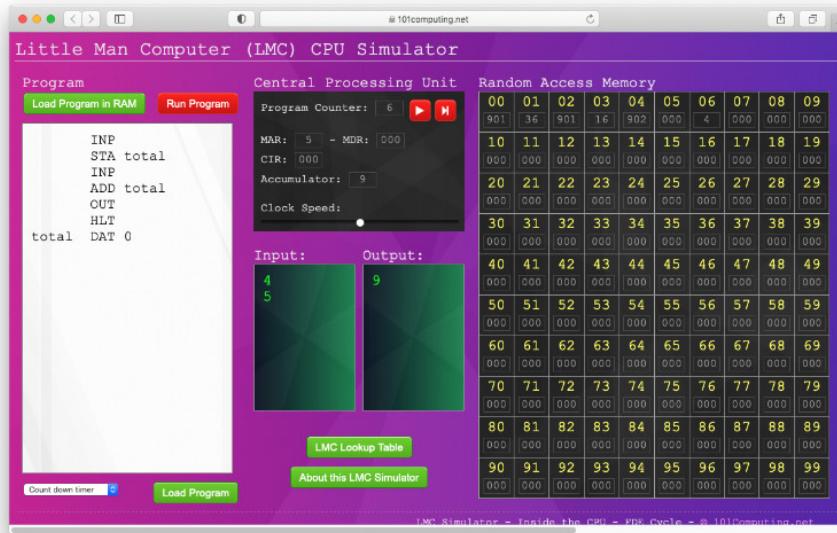


Figure 72: Storing data in accumulator using STA instruction and adding number in memory to accumulator using ADD.

## LMC Simulators

We can load this into an LMC simulator such as the one below at [101computing](http://101computing.net/lmc-simulator/):



The screenshot shows the simulator executing our program example with 4 and 5 as input, producing a 9 as output.

The robowriter simulator, while less flashy, is better for our purposes as it allows us to take an assembled program of machine instructions and load up directly. That way we have a simple way of testing if our assembler works as well as being able to compare our `LittleManComputer.jl` simulator with an existing one.

## API of LittleManComputer Package

`LittleManComputer.jl` has this API for working with LMC assembly code and instructions:

- `program = assemble(src)` takes a `src` file and produce `program`, integer array, consisting of LMC instructions.
- `disassemble(program)` takes as input `program`, an array of instruction and print out the corresponding mnemonics.
- `simulate!(program, inputs)` simulates executing of an LMC program with given inputs.

The example program shown here is already at the path `LittleManComputer/examples/add-two-inputs.lmc` directory. Here is an example of assembling, and simulating this program:

```
$ cd LittleManComputer/examples
```

```
julia> using LittleManComputer
julia> program = assemble("add-two-inputs.lmc")
7-element Array{Int64,1}:
 901
 306
```

```
901
106
902
  0
  0
```

Inputting numeric instructions into the simulator:

```
julia> simulate!(program, [4, 5])
1-element Array{Int64,1}:
 9
```

The simulator returns an array of all the outputs. To better understand what is happening and what code gets executed we can provide a callback:

```
julia> simulate!(program, [4, 5], callback=simcallback)
0:901  Accu: 0 // INP
1:306  Accu: 4 // STA 6
2:901  Accu: 4 // INP
3:106  Accu: 5 // ADD 6
4:902  Accu: 9 // OUT
5:0 Accu: 9 // HLT
1-element Array{Int64,1}:
 9
```

Now that you got some sense of what this package actually does, we can start talking about how to create tests for this package.

## Defining Tests for LittleManComputer

To be able to create tests, the minimal requirement is this structure:

```
test
└── runtests.jl
```

When you run tests from the package manager it will look for the file `test/runtests.jl` to perform the tests. Remember our discussion in the Code Organization chapter? This directory forms its own environment. Why is that handy?

Because this allows us to add particular package dependencies which are only relevant in testing, but not under normal execution of the Little Man Computer. The functionality for performing testing exists in the `Test` package, which we include with these steps:

```
(@v1.5) pkg> activate test
Activating environment at `~/Development/LittleManComputer/test/Project.toml`
```

```
(test) pkg> add Test
```

But this is not necessary because the `LittleManComputer` package already has done this. It has the following test directory structure:

```
test
```

```

    └── Manifest.toml
    └── Project.toml
    └── assem_tests.jl
    └── data
        ├── instructionset.lmc
        └── labels.lmc
    └── disassem_tests.jl
    └── runtests.jl

```

`test/data` contains data we use in our tests. The directory contains two `.lmc` programs. We want to test if they get assembled correctly by the `assemble` function.

`runtests.jl` is used as a wrapper:

```

using LittleManComputer
using Test

@testset "All Tests" begin

    include("assem_tests.jl")
    include("disassem_tests.jl")

end

```

As you can see, the actual test code is stored in the files: `assem_tests.jl` and `disassem_tests.jl`.

The `Test` package provides the `@testset` macro. Look at the disassembler tests. Notice how they are nested:

```

@testset "Disassembler tests" begin

    @testset "Without operands" begin
        @test disassemble(901) == "INP"
        @test disassemble(902) == "OUT"
        @test disassemble(000) == "HLT"
    end

    @testset "With operands" begin
        @test disassemble(105) == "ADD 5"
        @test disassemble(112) == "ADD 12"
        @test disassemble(243) == "SUB 43"
        @test disassemble(399) == "STA 99"
        @test disassemble(510) == "LDA 10"
        @test disassemble(600) == "BRA 0"
        @test disassemble(645) == "BRA 45"
        @test disassemble(782) == "BRZ 82"
    end
end

```

What is the purpose of nesting tests? It is not apparent if you run the tests as they are written:

```
(@v1.5) pkg> activate .
Activating environment at `~/Development/LittleManComputer/Project.toml`

(LittleManComputer) pkg> test
Test Summary: | Pass  Total
All Tests      |  47    47
Testing LittleManComputer tests passed
```

But say we alter some of the tests to make them fail, then we would get this kind of test report (edit for clarity):

```
(LittleManComputer) pkg> test
Test Summary:       | Pass  Fail  Total
All Tests          |  45     2     47
  Assembler tests |  36
  Disassembler tests |  9     2     11
    Without operands |  3
    With operands   |  6     2     8
```

Each line shows a test set which aggregates the results from the test sets it contains. E.g. *Disassembler tests* contains two test sets *Without operands* and *With operands*. Because the latter has 2 failures it means *Disassembler tests* inherits those two failures. Thus the failures bubble up to *All tests* which get 2 failures in total.

This makes it easier to organize our tests to represent functional units of our package. E.g. *assembly* and *disassembly* are two separate functional units, which we want to test separately. These can be further separated into even smaller functional units.

This gives an elegant way of expressing which functional parts are broken.

Let me use a car analogy: You test a car and test fails. Next you ask what part of the car failed and the answer is: ``the engine.'' That is one test set. But we can go further, ``What part of the engine failed?'' Answer: ``The Carburetor,'' which is our test set one level deeper. The Carburetor is further split into even smaller functional units which can be tested separately.

## The Test Macro

Inside a test set we write our individual tests. The tests are basic boolean expressions which must evaluate to `true` for the test to pass.

Let us look at the first test inside the *With operands* test set:

```
@testset "With operands" begin
    @test disassemble(105) == "ADD 5"
```

This tests that when 105 is disassembled it will turn into the ADD 5 instruction. To mark this boolean expression as a test which must be true we put the `@test` macro in front.

We can examine how this works by rewriting the test to deliberately fail:

```
@test disassemble(105) == "SUB 5"
```

Run the tests again, and there will be failure. Instead of the overview, let us focus on the diagnostics information which is produced. Here is a subset of the output which tells us what went wrong:

```
With operands: Test Failed at ~/Development/LittleManComputer/test/disassem_tests.jl:10
  Expression: disassemble(105) == "SUB 5"
  Evaluated: "ADD 5" == "SUB 5"
Stacktrace:
 [1] top-level scope at ~/Development/LittleManComputer/test/disassem_tests.jl:10
```

Notice it tells us the source code line of the failed test. We also see what the expression looks like in the source code as well as what it looks like when evaluated:

```
Evaluated: "ADD 5" == "SUB 5"
```

You can easily test out how this works in the REPL as well:

```
julia> using Test
```

```
julia> x = 5
5
```

```
julia> @test x == 5
Test Passed
```

```
julia> @test x == 6
Test Failed at REPL[98]:1
  Expression: x == 6
  Evaluated: 5 == 6
ERROR: There was an error during testing
```

The `@test` macro is very flexible as it allows us to create tests even inside loops. In many other languages a test must be defined as a function at the top level, which makes it hard to define multiple tests using a loop.

The contents of the `instructionset.lmc` file is this LMC assembly code:

```
ADD 0
SUB 0
STA 0
LDA 0
BRA 0
BRZ 0
BRP 0
INP
OUT
HLT
```

This covers almost all the defined instructions, and we want to check if compilation of this program will produce the expected numerical machine code instruc-

tions.

In the `Instruction set` test set, we assemble this file and then using a for-loop we check if each instruction was assembled correctly:

```
@testset "Instruction set" begin
    instructions = [100, 200, 300, 500, 600,
                    700, 800, 901, 902, 000]
    filepath = joinpath(datadir, "instructionset.lmc")
    program = assemble(filepath)
    for (i, expected) in enumerate(instructions)
        @test program[i] == expected
    end
end
```

Notice how we get the path of the file we are assembling. We get this path by combining "`instructionset.lmc`" with the `datadir` directory path. But where does `datadir` come from? `datadir` is defined as seen below:

```
datadir = joinpath(@__DIR__, "data")
```

This is a common trick in unit tests. `@__DIR__` is a macro which will always equal to the path of the directory containing the currently executed file. If used in the REPL it will give the working directory.

## Testing Terminology

To be able to discuss tests further, it is worthwhile to define some common concepts in testing:

- **Unit Test:** A single function or collection of functions working as one unit.
- **Integration Test:** Are two separately developed modules working properly together?
- **Regression Test:** The act of performing all your tests after a modification to make sure everything still works as intended.

This needs further elaboration to be better understood. I like space rockets and testing of rockets has a fascinating history. During the space race between the USSR and America two very different strategies to testing were followed.

Normally individual functional units would be tested. The turbo pump would be tested separately to make sure it is pumping fuel and oxidizer correctly. Later the whole rocket engine is test fired on a stand.

Next things like fuel tanks, stage separators, navigation computer and many other things get tested separately. Thus we are testing all the functional units separately.

A good reason for test sets to be hierarchical is that units really form a hierarchy. One could say that a rocket engine is a functional unit in a rocket stage, which needs to be tested.

But even a rocket engine could be split into different parts such as the combustion chamber, preburner and turbo pump. These parts can be tested separately. But it doesn't stop there. The nuts and the bolts used to assemble the part, was tested by their makers unless you made them yourself.

We can continue. Nuts and bolts are made of steel which also has to be quality tested. Hence it is easy to see how you get hierarchies and that is the beauty of the Julia *test sets* allowing you to create hierarchies almost arbitrarily deep.

The nuts with corresponding bolts can be tested as individual units to make sure they have the strength and rigidity we desire.

But there is no way for a nut and bolt maker to test their fasteners in every possible context. This is where integration tests come in. You test the nuts and bolts with an assembled machine part. The integration test may show that the bolts don't work together with the other parts. Maybe due to the humidity in the air and the other metals cause the bolts rust in their holes. Maybe the diameter of the holes don't fit the bolts.

At the higher level you may find that the rocket engine and all the other parts work fine alone. Launching the rocket is the full integration test. It shows whether all the different parts can work together to get the rocket into orbit. E.g. it may turn out that when the parts are integrated the rocket develops vibrations which destroys it.

In the space race, the US would perform detailed unit tests of all the functional units of the rocket. This made progress relatively slow even if it built confidence in success towards the final launch.

In the USSR in contrast there was very few unit tests. Integration tests dominated. Rockets would quickly get assembled. The Russian approach was rapid iteration, rather than meticulous unit testing. On the surface the two space programs thus looked very different. In the USSR it was normal for rockets to blow up. Blowing up rockets was their approach to development.

These represent outliers, but you can do combinations. E.g. the Apollo program switched to doing less unit tests and focused more on integration tests midstream.

---

But enough of the analogies. Let us get back to our actual project. For this example, unit tests could be as simple as the function parsing individual mnemonics. An integration test in contrast would mean to assemble a whole program and simulate it, making sure we get desired output when run.

That means the assembler has to be able to produce machine code that the simulator understands. But we can drill down further.

The `assemble` function e.g. calls other functions which could be tested separately, such as `assemble_mnemonic`:

```
@test assemble_mnemonic(["ADD", "12"]) == 112
```

Thus whether you view testing the `assemble` function as an integration test or a unit test depends on your perspective. Are you testing whether the assembler

(assemble) and simulator (simulate!) can work with each other?

```
path = joinpath(examples, "multiply.lmc")
program = assemble(path)
output = simulate!(program, [3, 4])
@test output == [3*4]
```

Or are you testing that assemble produce specific outputs for specific inputs defined by your unit test?

```
path = joinpath(examples, "add-two-inputs.machine")
program = load(path)
output = simulate!(program, [3, 4])
@test output[1] == 3 + 4
```

On larger projects we can imagine this pattern replicated at different levels in the hierarchy. Different developers on a project may create separate functionality which must be integration tested with functionality made by other team members.

Moving up the hierarchy, each team may be making a larger component which must be tested together with a component made by another team.

## **Test Driven Development and REPL Driven Development**

Test driven development (TDD) is very popular with object-oriented languages such as Python, Ruby and Java. With TDD, software is developed by writing tests first. In a nutshell it is about thinking about *how* your software is supposed to behave, *before* you write it. The theory is that by writing tests first you will not get tempted to write tests tailored towards your specific implementation.

Software development is then reduced to writing code until all the tests have passed. This approach works well if you have a clear specification of the software you are asked to make.

However this is by no means the only way of making quality software. Experienced software developers use a wide variety of approaches.

Through this book we have used a very different approach, usually referred to as REPL driven development. This is popular with more functional languages. E.g. you will see LISP, Clojure and yes Julia software developed this way.

With REPL driven development, implementation and testing is a merged activity. Code is written and tested in the REPL. It is quick to supply arguments to a function and look at outputs. This works well when you write pure functions where output only depends on inputs.

With object-oriented languages, this is less effective. Things like callbacks, overriding methods and mutating object-state makes REPL oriented development harder.

Let us revisit an earlier example of REPL oriented development. Remember how we developed a function converting snake case to camel case?

```
julia> s = "snake_case_example"
"snake_case_example"

julia> split(s, '_')
3-element Array{SubString{String},1}:
 "snake"
 "case"
 "example"

julia> uppercasefirst("hello world")
"Hello world"

julia> uppercasefirst.(split(s, '_'))
3-element Array{String,1}:
 "Snake"
 "Case"
 "Example"

julia> join(uppercasefirst.(split(s, '_')))
"SnakeCaseExample"

julia> camel_case(s) = join(uppercasefirst.(split(s, '_')));

julia> camel_case("hello_how_are_you")
"HelloHowAreYou"
```

This is a useful approach when you are not certain about the APIs you are using. E.g. does `split`, `uppercasefirst` etc work the way you expected. By trying out one small part of the functionality at a time and adding one little piece, you can gradually develop a solution.

Here we use the up-arrow key to bring back the code from previous statement, to add to it. Using `Ctrl-A` we jump to the beginning of the line to add another function call.

Because we are constantly calling the new modified call with the `s` argument, we are performing a simple test of the modified code each step of the way. Each time we get output verifying whether the function combinations do what we expect.

With this approach you typically add tests afterwards to make sure there are no regressions and that you have all corner cases covered.

Discussion of the best approach can easily turn into flame wars online. Here I will simply have to add my personal opinion: Try different approaches and see what works for you or your team. Don't get dogmatic. TDD and REPL based development are just tools for you to use.

## Testing at the Right Level

Whatever approach you follow for testing, there are some general good advice to follow. As discussed before, testing can happen at different levels. Initially

when developing the first solution, you may have very detailed tests at a low level to make sure you catch problems early. That is the equivalent to testing the bolts used in your rocket independently.

However as you modify and refactor<sup>45</sup> your code, a lot of low level details will change, even if the higher level behavior should stay the same.

In this case too low level tests can become a burden. Often developers don't want to change an obviously bad design, because it would require rewriting too many overly detailed tests. Don't get emotionally attached to your tests.

When they no longer serve a purpose, you should throw them away. Ultimately it is the high level tests that matter most. As your software matures, you may want to throw away the low level tests, as they will simply make refactoring harder than it needs to be.

E.g. in our LMC code example, what matters is that our `assemble`, `simulate!` and `disassemble` functions work properly.

Functions such as `symboltable` and `assemble_mnemonic` are just implementation details. Writing tests for them can be useful. What happens if you chance the implementation of `assemble` so it no longer needs these functions or these functions get a very different interface?

Just throw away those tests. They are of secondary importance anyway. It is the tests surrounding `assemble` which really give value.

---

<sup>45</sup>Refactoring code means modifying it, without modifying the external behavior of that code. People may also call this cleaning up your code.

# Logging

- **When** to use logging and what the purpose of logging is.
- **Logging Levels** describe how important a log message is. The logging system lets you use this to filter out less important messages.
- **Custom Loggers**. We will implement a simply custom logger.

In this chapter we will continue with the `LittleManComputer.jl` project from the Testing chapter to explore the Julia logging system.

## Logging vs Tests

Not every problem can be caught with tests. Tests are easy to create for pure<sup>46</sup> functions, where the output is entirely dependent on the input.

However the more a function depends on complex internal state for its behavior, the more difficult it would be to test it. An advantage of REPL based development is that it will naturally push you towards writing simple pure functions. Anything requiring a lot of scaffolding is tedious to use in a REPL environment.

Yet this is not always an option. For instance GUI and graphics oriented code tend to rely extensively on mutating state. Likewise when reading a database or communicating over the network, a lot can happen which was not taken into account when the unit tests where written.

In these cases logging is invaluable as you can record possible error situations as they occur.

## Logging vs Exceptions

Normally in Julia when unforeseen error events happen we use exceptions. In an ideal world you are able to catch these exceptions at a higher level where they are more easily handled. Afterwards the program can continue execution.

However this is not always possible. The right choice depends on the type of software we are building. If you are writing scientific software to make an important calculations, then getting the correct output is more important than making sure the program doesn't crash.

---

<sup>46</sup>A pure function has no side-effects and don't depend on side-effects. That means it does not change behavior depending on state which is not part of the input arguments, nor does it mutate any of the input arguments.

However if you are writing server software with many concurrent users, then crashing the program is undesirable. The same applies if rendering a web page. If there is an error in the HTML code of the web page, should we crash the whole program? Of course not! It is better to try to ignore the problem and instead attempt to continue to render the rest of the page. Inaccurate choice of fonts and colors is much less problematic than the web page crashing.

In such cases it is better to log details about the encountered problems, so developers can review logs afterwards and come up with code fixes.

## Using Loggers

To perform logging in Julia we use various macros called `@debug`, `@info`, `@warn` and `@error`. Here is an example of usage in the REPL:

```
julia> @info "general information to the user."
[ Info: general information to the user.

julia> @warn "something is wrong and action is likely required"
└ Warning: something is wrong and action is likely required
└ @ Main REPL[4]:1
```

Notice that macros are usually called without parenthesis. If you use the `@debug` macro you can see that nothing apparently happens:

```
julia> @debug "information intended for the developer of the program."
```

However we can enable it by setting the `JULIA_DEBUG` environment variable. You can set that in the shell before calling Julia. Specifically how, will depend on what shell you are using:

```
$ export JULIA_DEBUG="all"    # bash shell
$ set -x JULIA_DEBUG "all"   # fish shell
```

But it is impractical to exit into the shell each time you need to toggle debug logging. However it is easy to enable and disable debug logging from inside the Julia REPL.

```
ENV["JULIA_DEBUG"] = "all"  # Enable debug logging
ENV["JULIA_DEBUG"] = "Main" # Debug log for Main module
ENV["JULIA_DEBUG"] = ""     # Disable debug logging
```

The `ENV` dictionary contains the values of all the environment variables of the shell that launched Julia.

## Using Logger in Assembler

We can use these simple tricks to enable and disable debug logging in our LMC assembler. For instance the implementation of `assemble` looks like this:

```
function assemble(filename::AbstractString)
    lines = readlines(filename)
    program = Int[]
```

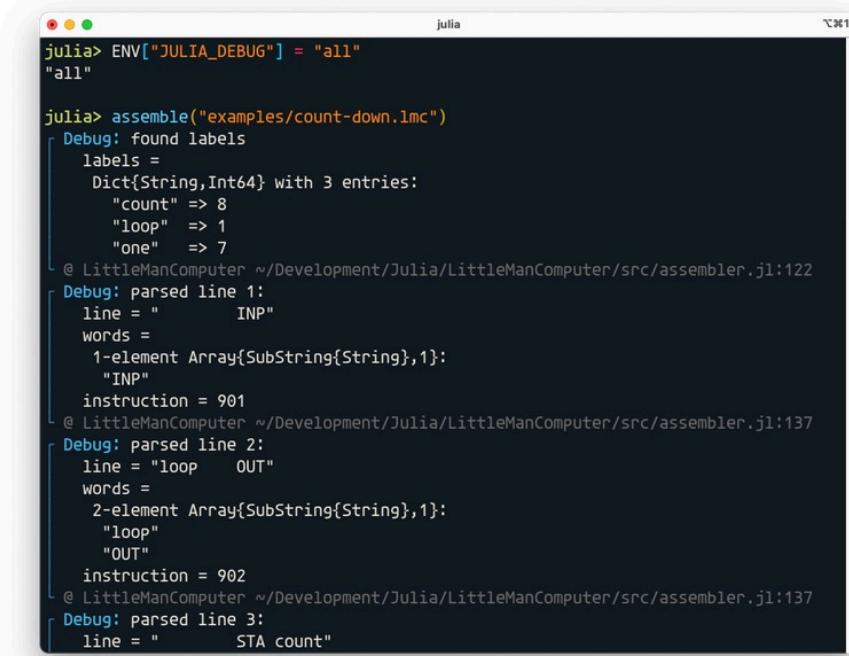
```
labels = symboltable(lines)
@debug "found labels" labels

for (i, line) in enumerate(lines)
    codeline = remove_comment(line)
    words = split(codeline)

    if isempty(words)
        continue
    end

    try
        instruction = assemble_mnemonic(words, labels)
        if instruction == nothing
            continue
        else
            @debug "parsed line $i:" line words instruction
            push!(program, instruction)
        end
    catch ex
        if isa(ex, InvalidMnemonicError)
            @error "Line $i: Encountered invalid mnemonic '$(ex.mnemonic)'"
        else
            rethrow()
        end
    end
end
program
end
```

The strategically placed `@debug` expressions allows us to run the `assemble` function and get a sense of how it is processing the inputs. We can toggle on debug logging and run the `assemble` function to see how the default logger works.



```
julia> ENV["JULIA_DEBUG"] = "all"
"all"

julia> assemble("examples/count-down.lmc")
Debug: found labels
labels =
Dict{String,Int64} with 3 entries:
"count" => 8
"loop"  => 1
"one"   => 7
@ LittleManComputer ~/Development/Julia/LittleManComputer/src/assembler.jl:122
Debug: parsed line 1:
line = "      INP"
words =
1-element Array{SubString{String},1}:
"INP"
instruction = 901
@ LittleManComputer ~/Development/Julia/LittleManComputer/src/assembler.jl:137
Debug: parsed line 2:
line = "loop    OUT"
words =
2-element Array{SubString{String},1}:
"loop"
"OUT"
instruction = 902
@ LittleManComputer ~/Development/Julia/LittleManComputer/src/assembler.jl:137
Debug: parsed line 3:
line = "      STA count"
```

Notice how the default logger neatly enclose each log statement with cyan colored lines. Consider the debug log statement below. It prints out multiple pieces of information.

```
@debug "parsed line $i:" line words instruction
```

Notice how we can add multiple variables we are interested to look at by just space separating them. Each time the debug macro is invoked it will produce output like this:

```
Debug: parsed line 3:
line = "      STA count"
words =
2-element Array{SubString{String},1}:
"STA"
"count"
instruction = 308
@ LittleManComputer ~/LittleManComputer/src/assembler.jl:137
```

Do you see how the debug macro is able to pick up both the names of the variables, we added, as well as their value? This is the kind of magic you can do with macros. Unlike functions, macros are partially run when the code is parsed. Long before the code is actually executed.

This means macros can pull tricks impossible to achieve with functions. And of major significance in this case, @debug macros have zero overhead when debug logging is toggled off. When turned off, there is actually no compiled code

present at the location of the debug macro call. That means you do not have to worry about adding debug macros to your code. They will not slow down your programs.

For instance if you have more complex calculations which you want to perform for debugging purposes, you can use `begin` and `end` to create a multiline expression and feed it to the `@debug` macro:

```
julia> xs = [4, 5, 3]
3-element Array{Int64,1}:
 4
 5
 3

julia> @debug begin
           total = sum(xs)
           "sum(xs) = $total"
       end
└ Debug: sum(xs) = 12
└ @ Main REPL[15]:1
```

When debug logging is toggled off, the sum calculation will not be performed.

## Understanding the Logging System

If you compare the Julia logging system to many other logging systems, the most immediate visible difference is that you are not invoking the logging functions on a particular logging object. Many logging systems log like this:

```
logger.warning("something bad happened, fix it!")
```

So why is Julia different? The problem with the approach above is that creator of a library decides for you how logging is done.

Every library might log in a different style and to a different location. Instead what we want in Julia is for the user of a library to decide how the logging is done. Julia allows you to specify which logger should be used for a specific function call. Say you want to call `assemble` with a specific logger, you can do like this:

```
with_logger(logger) do
    assemble("examples/count-down.lmc")
end
```

All the code between `do-end` will use the specified logger. The Julia Logging module contains a simple logger called `SimpleLogger`, which we can use to demonstrate this.

First we create an `IO` object to log to. That could be `stdout` if we are lazy, but let us use a real file:

```
io = open("log.txt", "w+")
```

The `w+` mode indicates that we want to open the file for both writing and appending. What does that mean? It means we don't write to the file from the start, overwriting existing content. Instead we write to the end, preserving existing content.

With an `IO` object we can create the logger:

```
logger = SimpleLogger(io, Logging.Debug)
```

However the second argument `Logging.Debug` needs an explanation. This is the desired logging level.

## Logging Levels

A logging level says something about how important the message is. It is actually a signed integer value. If you look at the Julia source code you will find this definition of the different log levels.

```
const Debug      = LogLevel( -1000)
const Info       = LogLevel(     0)
const Warn       = LogLevel(    1000)
const Error      = LogLevel(   2000)
```

Basically the higher the log level value the more likely it will be logged. Thus using the `@error` macro you log message with log level 2000, but if you use the `@debug` macro the importance of the log message will be recorded as -1000.

Thus when you write `SimpleLogger(io, Logging.Debug)` it means that `Debug` is the lowest log level we will log. But how do you log at an arbitrary logging level?

If you want another level than the predefined `Debug`, `Info`, `Warn` and `Error` ones you can use the `@logmsg` macro:

```
julia> @logmsg Logging.LogLevel(-400) "hello world"

julia> @logmsg Logging.LogLevel(0) "hello world"
[ Info: hello world

julia> @logmsg Logging.LogLevel(500) "hello world"
[ LogLevel(500): hello world

julia> @logmsg Logging.LogLevel(1500) "hello world"
└ LogLevel(1500): hello world
└ @ Main REPL[37]:1
```

Thus e.g. `@warn` is simply shorthand for:

```
julia> @logmsg Logging.LogLevel(1000) "hello world"
└ Warning: hello world
└ @ Main REPL[38]:1
```

## Choosing Logger

Okay, now we know the various parts involved so we can give a more complete example of picking a specific logger when calling `assemble`.

```
using Logging

open("log.txt", "w+") do io
    logger = SimpleLogger(io, Logging.Debug)
    with_logger(logger) do
        assemble("examples/counter.lmc")
    end
end
```

You can do this in a multitude of ways of course. Here I am using the `do` form of `open` to have my opened file automatically get closed when I am done. Afterwards you can open the `log.txt` file and see that it contains text looking like this:

```
shell> cat log.txt
Debug: found labels
└── labels = Dict("LOOP" => 2, "ONE" => 7, "QUIT" => 6)
    @ LittleManComputer assembler.jl:122
Debug: parsed line 1:
└── line =   INP
    words = SubString{String}["INP"]
    instruction = 901
    @ LittleManComputer assembler.jl:137
Debug: parsed line 2:
└── line =   OUT // Initialize output
    words = SubString{String}["OUT"]
    instruction = 902
    @ LittleManComputer assembler.jl:137
...
@ LittleManComputer assembler.jl:137
```

Of course this type of logging may not be too your satisfaction. Fortunately Julia lets you implement your own logger.

## Custom Loggers

All custom loggers must be subtypes of `AbstractLogger` found in the `Logging` module. We are going to make a custom logger called `AssemblerLogger`.

There are 3 different functions you can add methods to.

### Handle Message

```
handle_message(
    logger::AbstractLogger,
```

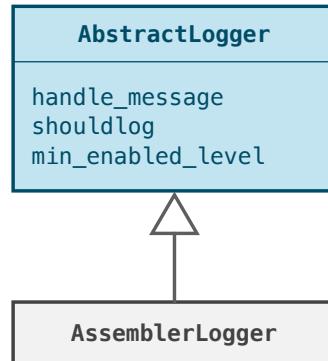


Figure 73: Type hierarchy for our custom AssemblerLogger

```

level::LogLevel,
message::AbstractString,
_module::Module,
group::Symbol,
id::Symbol,
file::AbstractString,
line; kwargs...)

```

This is the most important to implement. logger must be specified as the type of your logging subclass.

Let me cover the most important ones:

- **level** - The log level that was logged such as Warn, Debug and Error.
- **message** - The message provided to the log macro.
- **\_module** - The module the log message occurred in. Would be Little-ManComputer in this case.
- **group** - A unique identifier for the file message was produced in. Same as filename without suffix.
- **file** - Full path of file producing log message.
- **line** - Line number log message was produced
- **kwargs** - Keyword arguments. List of pairs for each variable added. The name of the variable as well as its value.

Let me elaborate on the last part. Say I wrote the following code:

```

x = 12
msg = "world"
@info "hello" msg x

```

The kwargs of handle\_message would then be an array looking like this:

```
[:msg => "world, :x => 12]
```

level would be equal to Info, and message would contain "hello".

## Should Log

```
shouldlog(
    logger::AbstractLogger,
    level::LogLevel,
    _module::Module,
    group::Symbol,
    id::Symbol) -> Bool
```

This should look at its inputs what the level is, module and file and use that to return `true` or `false` indicating whether message should be logged or not.

## Min Enabled Level

```
min_enabled_level(logger::AbstractLogger) -> LogLevel
```

Use this to remove log messages. Any log message below this level will be removed or ignored. For your own loggers this is used instead of the `JULIA_DEBUG` environment variable.

## Assembler Logger

We will create the `AssemblerLogger` so it is very similar to the `SimpleLogger`. To do this you need to explicitly import the logging functions we are adding methods to.

```
using Logging
import Logging: handle_message,
                shouldlog,
                min_enabled_level
```

Next we create our logger as a subtype of `AbstractLogger`.

```
struct AssemblerLogger <: AbstractLogger
    io::IO
    min_level::LogLevel
end
```

This holds the `IO` stream you are logging to. By default we use `stderr` but it could be a file stream as well. We set the default logging level to be `Info`, which is the same as level `0`. That means `Debug` logging is disabled since `debug` is at level `-1000`.

```
function AssemblerLogger(
    io::IO=stderr,
    level=Logging.Info)

    AssemblerLogger(io, level)
end
```

Although storing this log level doesn't actually cause any logging to be enabled or disabled. We must implement `min_enabled_level` to use our member value.

```
function min_enabled_level(log::AssemblerLogger)
```

```
    log.min_level
end
```

We don't get fancy with `shouldlog`. Usually here we would check whether it is the desired module, or whether we have reached a logging limit etc. Also notice we don't need to specify the types for anything but the logger. That is the important one otherwise this `shouldlog` does not apply to our specific subtype of `AbstractLogger`.

```
function shouldlog(
    logger::AssemblerLogger,
    level, _module, group, id)
    true
end
```

The important functionality is in `handle_message`. This is where we actually write out the log message. I will write out the types of the arguments, just for reference. It helps to see what type of data is supplied to this rather long function signature.

```
function handle_message(
    logger::AssemblerLogger,
    level::LogLevel,
    message::AbstractString,
    _module::Module,
    group::Symbol,
    id::Symbol,
    file::AbstractString,
    line; kwargs...)

    io = logger.io
    println(io, message)

    for (key, value) in kwargs
        println(io, " ", key, " = ", value)
    end
end
```

You can try this out in the Julia REPL. Either you have to paste this code into the REPL or have it stored in a file which you load into the REPL.

```
julia> x = 12
julia> msg = "world"

julia> with_logger(log) do
           @info "hello" msg x
       end
hello
msg = world
x = 12
```

As you can see it is relatively easy to use create your own logger. The Julia manual provides a lot more technical details in how you can tailor logging to

your needs.



# Debugging

- **Stepping** through code can be done in multiple ways in the debugger. We will explore stepping through lines, expressions and calls.
- **Breakpoints** specify locations you want to stop execution of a program being debugged.
- **Watch lists** allows you to monitor the values of one or more variables of interest while debugging.

Debugging is the process of identifying and removing errors from your software. We will discuss the process of debugging and how to use Julia's interactive debugger to facilitate debugging.

Like so many other things in Julia, the Julia debugger is distributed as a regular Julia package. You install it with the Julia package manager:

```
(@v1.5) pkg> add Debugger
```

To enable the debugger you simply load the `Debugger` package into your REPL environment. The debugger works by adding another mode to your REPL. Just like there is a help, shell and julia mode already in the REPL. This mode however is only active while you are in a debug session.

You start a debug session prefixing a function call or expression you want to debug with the `@enter` macro.

The screenshot below is an example of doing just that. In this case we are stepping through the code executed when adding two numbers in Julia.

Notice the use of the `s` command to make a single step in the code. But let us not get ahead of ourselves. Let me give you a birds-eye perspective first. What is a debugger exactly? When do we use them? And are they the only way to debug?

## What is a Debugger?

A debugger is a piece of software that typically lets you execute one line of code at a time. Alternatively you can place a breakpoint at a particular line of code. Next you run the code until it reaches this point. The program will then pause and the debugger will come into view. The debugger will let you inspect the current state of the program. What are the values of different variables? What is the stack backtrace? The stack backtrace refers to a list of function calls that led to the current code being executed. We will look more at that later.

There are many kinds of debuggers. Often sophisticated integrated development environments have graphical debuggers, where you have different windows or panes showing:

- The content of different variables.
  - The current stack backtrace.
  - Maybe even all the active threads in the program.

The debugger I am showing you here is a text based debugger. My reason for teaching you this as opposed to a shiny graphical debugger, is that it is:

- Feature rich.
  - Highly dependable. It will always work, regardless of editor you use.

- Work across network connections. Such as when you use SSH<sup>47</sup> interact with Julia on another computer.

The Atom and VSCode editors give a graphical interface to this debugger, but may require more extensive configuration to work properly.

## The Art of Debugging

It is easy to get into the mindset that debugging is the same as running a debugger, but actually it is *any* process used to narrow down a problem in your code and help you fix it.

In fact numerous developers don't use debuggers at all. Linus Torvalds the creator of the Linux Kernel doesn't:

I don't like debuggers. Never have, probably never will. I use gdb all the time, but I tend to use it not as a debugger, but as a disassembler on steroids that you can program.

The creator of the Python programming language Guido van Rossum, also doesn't use debuggers, preferring print statement instead. That is a less sophisticated version of the @debug logging we did in the Logging chapter. Interestingly if you decide to research this topic, you will find quite a number of high profile software developers, who don't use debugger. Or who only use them in a very limited fashion.

It may seem strange to start a chapter discussing debugger by discouraging their use. However I want to encourage you to not get too tool focused. A good debugger is not a substitute for developing good habits in reading, analyzing and understanding code.

I have seen many times how people mindlessly step through code with their shiny graphical debugger (I was once one of them) wasting time that could have been spent understanding the code and thinking about the problem.

So before actually looking at how to use the interactive debugger, let me say some words about the alternatives.

## Alternatives to Debuggers

REPL oriented development significantly cut down on the need to run debuggers. If you create small self contained functions which you are actively testing in the REPL environment as you are developing your code, then this will help you a lot in discovering problems.

Writing extensive unit tests is another approach. Should you discover that there is a problem in older code, you can still use this approach. Use the scientific

---

<sup>47</sup>SSH is short for Secure Shell. It is a way method of having an encrypted telnet session with another computer. Basically it is to access the terminal of another computer using an encryption network connection.

method. Develop one or more hypothesis for what is causing your code to behave the way it does. Using the REPL you can then test different hypothesis and confirm whether your hunch is right.

Initially you may not understand or know enough about the code to develop a hypothesis. Then running functions with different input combined with reading the code helps you develop a sense of what the code does.

When a function is large and does many things, then this will not be enough. In this case you can add @debug macros to the function you are studying.

In these cases it is very useful to use the `Revise` package. It allows you to modify the function of interest and have those changes immediately reflected in the REPL.

One technique is to combine experimental code changes, with the REPL and the git version control system. Make sure you have commit your last code changes. Then you can make edits to the code to try out ideas immediately in the REPL. These changes are not attempted fixes but simply modification of the code to learn how the function behaves. It is throw away code, you can easily discard later using your version control system.

## Debugging the Simulator

I am going to show you how to use an interactive debugger by stepping through the code for the `LittleManComputer` simulator.

When writing this part I pondered whether I should introduce a deliberate bug which we could hunt down together in the debugger and fix.

The reason I decided against this is because in practice this is almost never how I use a debugger. Instead I use it to gain an understanding of the code. After this understanding has been gained I can look at the source code and better determine what is going wrong.

Thus instead the focus here will be on how we can use a debugger to step through code in order to better comprehend the behavior of the code at runtime.

To help us do that, we will run the `add-two-inputs.lmc` program in the simulator. This is a very simple program reading two input values, adding them and writing the result to output. You can find it in the example subdirectory.

```
INP
STA total
INP
ADD total
OUT
HLT
total DAT
```

Just like a normal computer, our simulator does not understand assembly code. It can only run machine code and hence we need to translate our assembly code into machine code:

```
julia> code = assemble("examples/add-two-inputs.lmc")
7-element Array{Int64,1}:
 901
 306
 901
 106
 902
 0
 0
```

Here I run the simulator with 3 and 4 as input values. As expected the simulator returns with 7 as output.

```
julia> simulate!(code, [3, 4])
1-element Array{Int64,1}:
 7
```

The memory containing the code has also been mutated. The last entry in our program was:

```
total DAT
```

This marks the storage cell for our `total` variable. The first value read is stored here. We can observe that by dumping contents of `code`. Notice the last value has switched from 0 to 3:

```
julia> code
7-element Array{Int64,1}:
 901
 306
 901
 106
 902
 0
 3
```

For this reason it is often useful to copy the code before running, so we can use unmodified code on each repeated run.

To run the debugger, we load it like any other Julia package by using `using`. Next we add `@enter` in front of our function call to step into the debugger.

```
julia> using Debugger

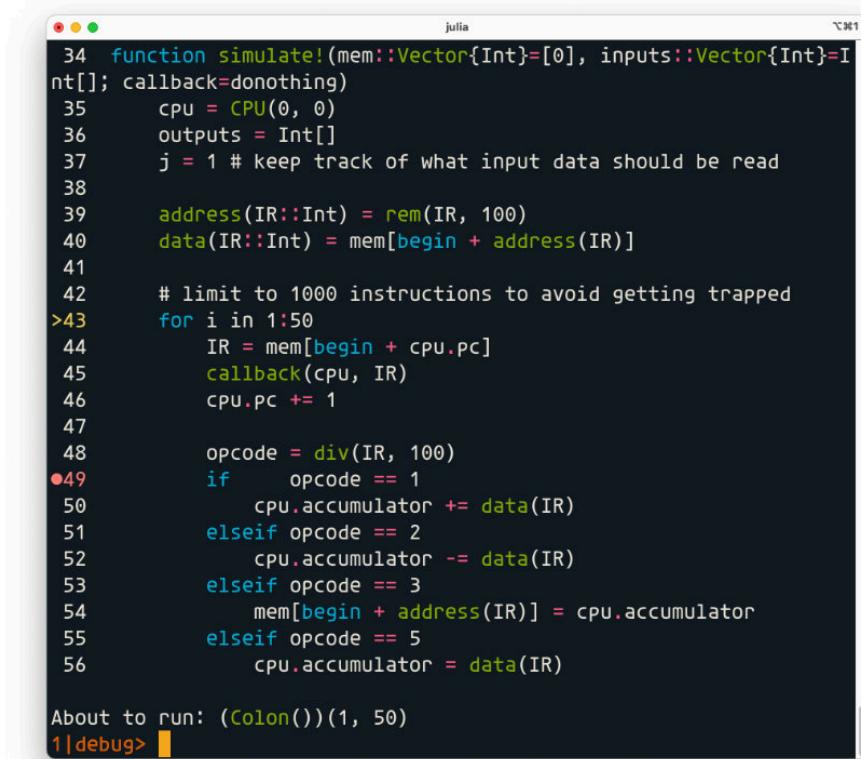
julia> code = assemble("examples/add-two-inputs.lmc");

julia> @enter simulate!(code, [3, 4])
```

This will bring us into a debug session. Below you can see the state of the debugger after I made the following changes:

1. I used the + key to expand the number of lines you see while stepping through your program. Use - to reduce the number of lines shown.
2. A breakpoint has been added to line 49. Shown with a red dot.

3. I have executed a few lines of code. The next line of code to be executed is at line 43 marked in yellow.



The screenshot shows a Julia debugger window. The code being run is:

```

34 function simulate!(mem::Vector{Int}=[0], inputs::Vector{Int}=I
nt[]
35     cpu = CPU(0, 0)
36     outputs = Int[]
37     j = 1 # keep track of what input data should be read
38
39     address(IR::Int) = rem(IR, 100)
40     data(IR::Int) = mem[begin + address(IR)]
41
42     # limit to 1000 instructions to avoid getting trapped
>43     for i in 1:50
44         IR = mem[begin + cpu.pc]
45         callback(cpu, IR)
46         cpu.pc += 1
47
48         opcode = div(IR, 100)
●49         if      opcode == 1
50             cpu.accumulator += data(IR)
51         elseif opcode == 2
52             cpu.accumulator -= data(IR)
53         elseif opcode == 3
54             mem[begin + address(IR)] = cpu.accumulator
55         elseif opcode == 5
56             cpu.accumulator = data(IR)

About to run: (Colon())(1, 50)
1|debug>

```

The line at index 49 is highlighted in yellow, indicating it is the next line to be executed. The prompt at the bottom has changed to "1|debug>".

You can see at the bottom that the prompt has changed to indicate that we are in debug mode:

1|debug>

If you hit the backticks key, then you jump into Julia mode, where you can write code like in a normal REPL:

```

1|julia> j
1

1|julia> inputs
2-element Array{Int64,1}:
3
4

1|julia> println("hello")
hello

```

To get back to debug mode you hit the *backspace* key. By writing ? you can show the help page with an overview of valid commands:

```
1 |debug> ?
Debugger commands
=====
```

Below, square brackets denote optional arguments.

#### Misc:

- o: open the current line in an editor
- q: quit the debugger, returning nothing
- C: toggle compiled mode

I am not showing the full list here. At the bottom you see *compiled mode* toggled by hitting C. We will look at that later. You can look at the full list of commands yourself but here I will cover the most important ones:

#### **Exit**

The most annoying thing about any command line tool is getting stuck and not getting out. Use q to exit debugger. Alternatively Ctrl-D, same as used to exit Julia.

## Stepping Through Code

You can step through your code in multiple ways. In order the most important ones are:

- n: Step to next line. Runs one line of code.
- s: Step into next call.
- c: Continue execution until a breakpoint is hit.
- so: Step out of current call.

Typically each line of code calls one or more functions. You don't want to look at what lines of code `println` performs or what code is performed when you access an index. Thus you use n to step whole lines, most of the time.

You use s if you want to look at how a particular function being called works. It causes you to step into a function and see the code it is made up of. If you don't want to step through this function you can get back by using so. It will run all the code in the current function and then stop executing once you are back at the call site for this function.

Try using n to step one line at a time until you hit line 43, where the for-loop is. Your terminal should look something like this (edited for clarity):

```
35      cpu = CPU(0, 0)
36      outputs = Int[]
37      j = 1
38
39      address(IR::Int) = rem(IR, 100)
40      data(IR::Int) = mem[begin + address(IR)]
41
42
```

```

>43      for i in 1:50
44          IR = mem[begin + cpu.pc]
45          callback(cpu, IR)

About to run: (Colon())(1, 50)
1|debug>

```

Notice how it says `About to run (Colon())(1, 50)` rather than what you might have expected:

`About to run for i in 1:50`

Why is that? Because the for-loop is not a single call. It is made up of multiple parts. The first thing done is to construct a range object using the call `1:50`. Remember this is actually a function call `(:)(1, 50)`. Just like `5 + 3` is short for the function call `+(5, 3)`. However to dig deeper into the rabbit hole `(:)` is actually short for `Colon()`. `Colon` is a singleton just like `Nothing` and `Missing`.

From the Callable Objects and Command Pattern you may remember how we can define functions on objects. I used the example of a polynomial:

```
(f::Polynomial)(x::Real) = f.a*x^2 + f.b*x + f.c
```

This is the same thing going on here. You can think of the range operator as being defined like this:

```

colon = Colon()
function (colon)(start::T, stop::T) where {T<:Real}
    UnitRange{T}(start, stop)
end

```

This can be a bit of the challenge using the Debugger as it pulls you down the rabbit hole, exposing a lot of details of how Julia works under the hood.

We can step into this colon function using `s` to see how it is implemented.

```

1|debug> s
In Colon(start, stop) at range.jl:5
>1 1 — %1 = (Core.apply_type)(UnitRange, $(Expr(:static_parameter, 1)))
2 |   %2 = (%1)(start, stop)
3 |   return %2

```

What you see here is not normal Julia code but what is called *lowered code*. When Julia code is compiled by the JIT, it goes through several code transformations. One of the first is called *lowering*. High level Julia code is transformed into much lower level code. I am not going to teach you that here, but it is useful to know about its existence as you will get exposed to it on occasion.

Since we are not interested in stepping through this lowered code and want to get back to our call site we use the `so` command to step out of the current call and get back:

```

1|debug> so
36     outputs = Int[]
37     j = 1
38

```

```

39      address(IR::Int) = rem(IR, 100)
40      data(IR::Int) = mem[begin + address(IR)]
41
42
>43      for i in 1:50
44          IR = mem[begin + cpu.pc]

```

About to run: (iterate)(1:50)  
1|debug>

Yet again we are told that the next function being called, `iterate(1:50)`, is not something we directly see in the code.

Remember how in chapter Object Collections we covered how the for-loop is just syntactic sugar for a while-loop. Thus the code we are actually looking at looks like this:

```

next = iterate(1:5)
while next != nothing
    (x, i) = next
    IR = mem[begin + cpu.pc]
    ...
    next = iterate(1:5, i)
end

```

If we step into this function with `s` we get the following:

```

1|debug> s
In iterate(r) at range.jl:620
>620 iterate(r)::OrdinalRange = isempty(r) ? nothing : (first(r), first(r))

```

About to run: (isempty)(1:50)  
1|debug so

We don't want to look at this code in further detail either so we step out with `so`.

## Using a Watch List

When stepping through code we want to keep track of how different variables change. In graphical debuggers we have the concept of a *watch list*, which is a separate pane in the GUI showing the variables you want to keep track of.

You can create something which is conceptually the same in the Julia debugger with the `w` command. As we step through the code we might be interested in what inputs our assembly program is reading. We can keep track of that with:

```
1|debug> w add inputs[j]
1] inputs[j]: 3
```

We would also like to know where in the assembly program we are currently located, which is stored in the *program counter* `cpu.pc`:

```
1|debug> w add cpu.pc
1] inputs[j]: 3
```

```
2] cpu.pc: 0
```

You can see each time we add something to the watch list we get an overview of everything currently in the watch list. The numbers lets you refer to an item in the watch list. Let me add another item to demonstrate:

```
1|debug> w add j
1] inputs[j]: 3
2] cpu.pc: 0
3] j: 1
```

You might decide that showing the value of `j` is pointless as we already get what we are interested in directly from `inputs[j]`. We can then remove the third entry with:

```
1|debug> w rm 3
1] inputs[j]: 3
2] cpu.pc: 0
```

It is also possible to add variables which are not yet known such as `IR` (Instruction Register), which contains the instruction to be executed next.

```
1|debug> w add IR
1] inputs[j]: 3
2] cpu.pc: 0
3] IR: UndefVarError(:IR)
```

In this case the debugger will simply tell us that the value is not yet defined.

Let us go back to stepping through the code to see how the watchlist is updated. At this point you may want to see what source code line we are currently at. You get that back with the `st` command.

Use the `n` command to repeatedly step until you hit line 49. Then we can use the `w` command to see the content of our watch list:

```
48      opcode = div(IR, 100)
>49      if      opcode == 1
50          cpu.accumulator += data(IR)
51      elseif opcode == 2

About to run: ==(9, 1)
1|debug> w
1] inputs[j]: 3
2] cpu.pc: 1
3] IR: 901
```

At this point you may want to look at e.g. the content of the `opcode` variable. You got many options at your disposal. You could add it to your watch list, or you could simply hit backtick to go into Julia mode. Here you can inspect it as normal:

```
1|julia> opcode
9
```

The benefit of this mode is that you can experiment or explore anything. Are you curious what `div(IR, 100)` function call does, or what `opcode == 1` will result in? No problem just try it yourself:

```
1|julia> div(IR, 100)
9
```

```
1|julia> opcode == 1
false
```

You could even change the value of variables or add new functions. If you have `Revise` loaded, you can even modify the code you are stepping through in the code editor and the Julia Debugger will pick up this change.

Alternatively we can go back to debug mode by hitting backspace and use the `frame` command `fr` to see the value of variables in the current frame. For clarity I have edit the output:

```
| mem::Array{Int64,1} = [901, 306, 901, 106, 902, 0, 0]
| inputs::Array{Int64,1} = [3, 4]
| cpu::CPU = CPU(0, 1)
| outputs::Array{Int64,1} = Int64[]
| j::Int64 = 1
| i::Int64 = 1
| IR::Int64 = 901
| opcode::Int64 = 9
```

*Frame* is a concept we have not yet covered.

## What is a Frame?

Functions have local variables to store results of temporary calculations. These variables need some memory to actually be stored. When a function gets called a chunk of memory is set aside for that function to store its local variables. This chunk of memory is called the *frame*. The function that called another function also had a frame. Thus frames exist in a numbered hierarchy. `fr` is just short for `fr 1` which is the current frame. You could look at the frame one step higher:

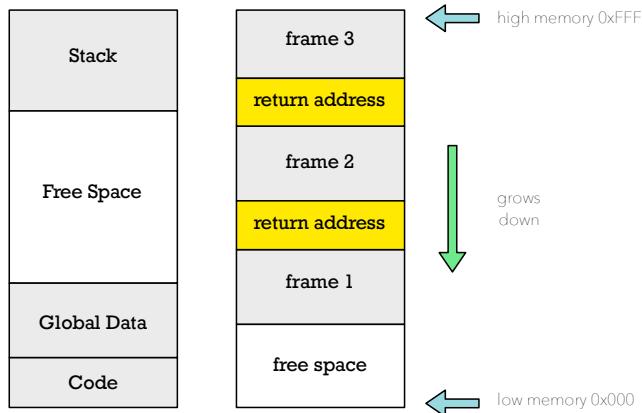
```
1|debug> fr 2
| mem::Array{Int64,1} = [901, 306, 901, 106, 902, 0, 0]
| inputs::Array{Int64,1} = [3, 4]
1|debug>
```

## The Stack

Debugging expose you to a number of concepts that one typically ignored when working with higher level programming languages such as Julia. One of this concepts is what we simply call *The Stack*. The name refers to a strategy for allocating memory, called stack-based memory allocation.

The name comes from the fact that it works similar to how the stack data structure works. You can use a regular `Array` in Julia as a stack by pushing and popping values with the `push!` and `pop!` functions.

The Stack works the same way. We add slabs of memory needed on top of the stack and release memory from the top of the stack. It is a last-in-first-out (LIFO) data structure.



If function f calls function g which calls function h, then the stack grows so that on the top of the stack you find the frame of function h. This is admittedly a bit confusing to talk about since the stack grows downwards in memory. Hence what is the *top* of the stack is actually at the bottom of the diagram below. So we got *frame 1* at the top of the stack containing variables for function h. Right below we got the return address which takes us back to where the call to h happened inside the g function.

As a Julia programmer it is not important to have a deep understanding of this. If you were an assembly programmer or C/C++ programmer then this would be more important to know.

Yet the concept is useful to be aware of since it pops up in the terminology used in most debuggers. The variables found in the current function being executed will be called the *frame*. Sometimes you will see the term *activation frame* used.

Another commonly used terminology is the stack backtrace.

## Stack Backtrace

Usually the stack backtrace shows a list of function calls which got you to your current location. In the Julia debugger it gives you a list of function call along with their frames. You can open another terminal window and start a debugger to observe how this works with a simpler function call.

Here I will use pop! to remove an element from a short array xs:

```
julia> xs = [4, 5, 3];
julia> @enter pop!(xs)
In pop!(a) at array.jl:1145
  1145  function pop!(a::Vector)
>1146      if isempty(a)
```

```
1147         throw(ArgumentError("array must be non-empty"))
1148     end
```

You can experiment with this yourself. Just keep using the step-into command `s`. Here you can see I have done this repeatedly until I hit the `length` function:

```
1|debug> s
In length(a) at array.jl:219
>219  length(a::Array) = arraylen(a)
```

```
About to run: return 3
1|debug>
```

By writing `bt` (back trace) I can see all function calls and the frame for each of them:

```
1|debug> bt
[1] length(a) at array.jl:219
| a::Array{Int64,1} = [4, 5, 3]
[2] isempty(a) at abstractarray.jl:989
| a::Array{Int64,1} = [4, 5, 3]
[3] pop!(a) at array.jl:1146
| a::Array{Int64,1} = [4, 5, 3]
1|debug>
```

## Using Breakpoints

Manually stepping through every line of code quickly gets tedious. This is why breakpoints are helpful. They allow you to mark off lines in your code where you want to pause and look at the state of your variables.

All breakpoint commands start with `bp`. Break points are numbered according to when they got added. Very similar to how the watch list works. This is handy to know if you need to remove a breakpoint.

```
43  for i in 1:50
44      IR = mem[begin + cpu.pc]
45      callback(cpu, IR)
46      cpu.pc += 1
47
48      opcode = div(IR, 100)
>49      if      opcode == 1
```

Line 49 is a nice place to pause execution, because there you can see what LMC machine instruction you got and you can step through manually to see the simulator making a decision on what to do.

We add a breakpoint to line 49 with:

```
1|debug> bp add 49
1] ~/Development/LittleManComputer/src/simulator.jl:49
```

We can keep adding more breakpoints just to see how this works. Remember you can use `st` to list the code whenever you want.

We can add a breakpoint at line 76. That is where the output is written.

```
1|debug> bp add 76
1] ~/Development/LittleManComputer/src/simulator.jl:49
2] ~/Development/LittleManComputer/src/simulator.jl:76
```

To demonstrate removal of breakpoints let us add a pointless breakpoint at line 35, where the function starts.

```
1|debug> bp add 35
1] ~/Development/LittleManComputer/src/simulator.jl:49
2] ~/Development/LittleManComputer/src/simulator.jl:76
3] ~/Development/LittleManComputer/src/simulator.jl:35
```

Say I accidentally remove the middle breakpoint:

```
1|debug> bp rm 2
1] ~/Development/LittleManComputer/src/simulator.jl:49
2] ~/Development/LittleManComputer/src/simulator.jl:35
```

You can see the numbering of breakpoints are not fixed. The number which refers to the breakpoint on line 35 has now changed from 3 to 2. The watch list works the same way. Let us fix the breakpoint list by removing the pointless breakpoint and add the one we cared about:

```
1|debug> bp rm 2
1] ~/Development/LittleManComputer/src/simulator.jl:49
```

```
1|debug> bp add 76
1] ~/Development/LittleManComputer/src/simulator.jl:49
2] ~/Development/LittleManComputer/src/simulator.jl:76
```

We you can then keep hitting `c` (continue) to run through the whole loop until you hit the breakpoint on line 49. Then hitting `w` you can check the current state of things or use `fr`.

```
1|debug> c
Hit breakpoint: simulator.jl:34
 48  opcode = div(IR, 100)
>49  if      opcode == 1
 50      cpu.accumulator += data(IR)
```

About to run: (==)(9, 1)

```
1|debug> w
1] inputs[j]: 4
2] cpu.pc: 3
3] IR: 901
```

```
1|debug> c
Hit breakpoint: simulator.jl:34
 48  opcode = div(IR, 100)
>49  if      opcode == 1
 50      cpu.accumulator += data(IR)
```

```
About to run: (==)(1, 1)
1|debug> w
1] inputs[j]: BoundsError([3, 4], (3,))
2] cpu.pc: 4
3] IR: 106
```

## Using the Debugger Effectively

Debuggers in a dynamic language like Julia with a powerful REPL environment will be utilized differently from a debugger in statically typed language. The advantage of a dynamic environment is that you can easily enhance your environment using the language itself.

For instance a common problem in debuggers when looking at the state of variables is how to visualize that state. In Julia however it would be trivial to add your own collection of functions specifically designed to aid in debugging. You could even put them in separate package. Say you are debugging code dealing with a binary tree. In theory you could write a function which gives a graphical display of the binary tree. You could add your own `show` function to display variables in a manner more practical while debugging.

And don't forget that by using the `Revise` package together with the debugger you have a powerful combination. This allows you to modify a program while you are debugging. This is useful both in terms of trying to fix the problem while debugging or simply to make random changes to better understand how the code works.



# Where Now?

- **Further study.** What are ways to improve your Julia knowledge further?
- **Julia packages** which are worth knowing about.
- **Community.** Getting in touch with the Julia community.

There are so many things about Julia I would have wished I could have written more about, but there is only so much one can cover in a book which is aimed at beginners of all backgrounds. Had I covered data science, machine learning and visualization as well that would have severely bloated this book.

But now that you have been introduced to Julia, you may wonder what your next step should be. What are other interesting things to explore?

The main Julia web site has its a comprehensive list of resources, which are worth examining.

If you liked this book you may also want to check out the video course: Getting Started With Julia, as it is by the same author (me).

It does cover some more advance topics not covered in this book, such as meta-programming and performance. Be warned that it is based on an old version of Julia.

I am also a regular writer on Medium covering Julia among many other topics. These articles often take angles less suitable in a book such as more in depth comparison between Julia and other languages as well as getting deeper into specific technical details. You can find an overview of articles at: <http://blog.translusion.com/stories/programming/>

## Julia Packages

The goto place to explore Julia packages is juliahub.com. I am not intimate with all the packages but I will give a short overview of some of the packages I know and find useful.

- **Plots** is the best all around package for plotting in Julia. It may not be the most sophisticated or promising, but it is well supported, stable and works. I recommend other plotting packages as you get more experienced and as they mature.
- **DataFrames** is for working with tabular data. You can sort, filter, categorize, summarize and numerous other things with data stored in tables.

- **Luxor** is a user-friendly and flexible package for making beautiful 2D vector graphics.
- **Pluto** is an innovative notebook concept for Julia. It lets you mix text, math equations, and illustrations with Julia code. You can run code inside a document. Very useful for scientists presenting their findings and letting students or readers experiment with their published results.
- **Flux** is an awesome machine learning library, specifically aimed at deep learning. This I would argue is the easiest to learn modern high performance machine learning library you can find.
- **Makie** is a very promising plotting and visualization library. If it was more mature and complete I would recommend it over Plots. It seems to make nicer looking plots, offer better performance and features. However my latest experience has been that it is not quite ready for prime time, but worth trying out.
- **DifferentialEquations**. Not a package I am personally familiar with, but which has made a name for Julia as it is considered by many as the best in the industry for any programming language. It is for people who want to solve differential equations.
- **HTTP** for working with the HTTP protocol. You can use it to serve web pages or request web pages or other data through the HTTP protocol.
- **Genie** is a Model-View-Controller web framework for creating modern web applications in Julia.
- **Franklin** for creating static web sites. Similar to Jekyll and Hugo. Aims to be light weight and work well with typical Julia work, hence it has good support for rendering math equations.

## Julia Community

On the Julia home page you can find an overview of the Julia community: <https://julialang.org/community/>

This includes youtube, github, twitter and chat on Slack, which I can highly recommend. The Julia community is very friendly and willing to answer questions.

JuliaCon is the official Julia conference, held each year. Attending is a great way to get connected to the wider Julia community. JuliaCon 2021 will be online and free.

For me personally, JuliaCon has been one of my best conference experience. A lot of that comes down to the people that make up the Julia community. It is not just the spirit of the community but also the wide variety of interesting topics covered at Julia conferences.

I hope you will enjoy Julia as much as I have.