Elixir Basics

Notes for «Elixir in Action» (2nd Edition)

Patrick Bucher

2023-10-09

Contents

These notes are based on Elixir in Action by Saša Jurić.

1 Setup

Install Elixir (on Arch Linux):

```
# pacman -S elixir
```

The following binaries are now available:

- `elixir(1)`: The Elixir script runner
- `elixirc(1)`: The Elixir compiler
- `iex(1)`: The Elixir shell

Check the installed version:

```
$ elixir --version
```

Run Elixir interactively:

```
$ iex
iex(1)> IO.puts("Hello, World!")
Hello, World!
:ok
```

Write an Elixir script (`examples/hello.exs`, the s stands for “script”):

```
IO.puts("Hello, World!")
```

Run the Elixir script:

```
$ elixir hello.exs
Hello, World!
```

1.1 Documentation

- Install Elixir
- Introduction
- Reference
- Erlang Documentation
- Elixir/Erlang Crash Course
- IEx
- Mix (Build Tool)

For interactive help, launch `iex` and type `h`:

```
$ iex
iex(1)> h
```

For help on a specific topic, launch `iex` and type `h [topic]`:

```
$ iex
iex(1)> h Kernel
```

2 Variables

Variables are dynamically typed. Variable assignments (*bindings*) are expressions; they return the value being assigned:

```
> a = 3
3
> b = 1.5
1.5
> c = a + b
4.5
```

Variable names must start with a lowercase alphabetic character or an underscore. By convention, only lowercase characters, numbers, and underscores are used; the last character can also be a question (?) or exclamation mark (!):

- `good_variable_name`
- `good_variable_name_2`
- `good_variable_name?`
- `good_variable_name!`
- `validButNotGoodVariableName`
- `InvalidVariableName`

Variables cannot be *changed*, but *rebound*:

```
> a = 3
3
> a = 4
4
```

3 Modules

Functions are grouped together in Modules (`examples/geometry.ex`):

```
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end

  def rectangle_perimeter(a, b) do
    2 * a + 2 * b
  end
end
```

Modules can be used interactively using `iex`:

```
$ iex
> Geometry.rectangle_area(3, 2)
6
> Geometry.rectangle_perimeter(3, 2)
10
```

Module names are written in CamelCase; alphanumeric characters and the dot are allowed in them.

Multiple modules can be defined in the same file. Modules can also be nested hierarchically (`examples/calculator.ex`):

```elixir
defmodule Calculator do
  defmodule Basic do
    def add(a, b) do
      a + b
    end
  end

  defmodule Advanced do
    def pow(a, b) do
      Integer.pow(a, b)
    end
  end
end
```

The module names are qualified with a dot:

```
$ iex examples/calculator.ex
> Calculator.Basic.add(5, 3)
8
> Calculator.Advanced.pow(5, 2)
25
```

3.1 Imports and Aliases

Other modules can be imported into the current module, so that function calls don't have to be qualified using their module name. It's also possible to use an alias name for an imported module (`examples/hello_calculator.ex`):

```elixir
defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end

  def rectangle_perimeter(a, b) do
    2 * a + 2 * b
  end
end

defmodule HelloCalculator do
  import IO
  alias Geometry, as: Geom

  def rect_info(a, b) do
```

```
    area = Geom.rectangle_area(a, b)
    perimeter = Geom.rectangle_perimeter(a, b)
    puts("Rectangle(#{a}, #{b}): Area #{area}, Perimeter: #{perimeter}")
  end
end
```

The module `HelloCalculator` imports `IO`, so `puts` can be used without further qualification (instead of `IO.puts`). `Geometry` is also imported, but using an alias, so that it can be used as `Geom`:

```
$ iex examples/hello_calculator.ex
> HelloCalculator.rect_info(3, 5)
Rectangle(3, 5): Area 15, Perimeter: 16
:ok
```

The Kernel module is always imported automatically, so that its functions can be used without further qualification.

3.2 Module Attributes

Module attributes are used to define constants and to provide documentation (`examples/free_fall.ex`):

```
defmodule FreeFall do
  @moduledoc "Offers functions concerning the free fall of objects"
  @gravity 9.81

  @doc "Computes the velocity on impact given the height"
  @spec impact_velocity(number) :: number
  def impact_velocity(height) do
    :math.sqrt(2 * @gravity * height)
  end

  @doc "Computes the time it takes for the object to reach the ground"
  @spec fall_time(number) :: number
  def fall_time(height) do
    impact_velocity(height) / @gravity
  end
end
```

`@gravity` defines a compile-time constant.

`@moduledoc` and `@doc` provides documentation for the surrounding module and for the following function, respectively. `@spec` provides type specificatons that can be analyzed using the `dialyzer` The module needs to be compiled in order to have this documentation accessible during runtime:

```
$ elixirc examples/free_fall.ex
$ file Elixir.FreeFall.beam
Elixir.FreeFall.beam: Erlang BEAM file
$ iex
> Code.fetch_docs(FreeFall)
{:docs_v1, 2, :elixir, "text/markdown",
 %{"en" => "Offers functions concerning the free fall of objects"}, %{},
 [
   {{:function, :fall_time, 1}, 10, ["fall_time(height)"],
    %{"en" => "Computes the time it takes for the object to reach the ground"},
    %{}},
   {{:function, :impact_velocity, 1}, 5, ["impact_velocity(height)"],
    %{"en" => "Computes the velocity on impact given the height"}, %{}}
 ]}
```

The help function (`h`) is more helpful for interactive use:

```
 > h FreeFall

                                FreeFall

Offers functions concerning the free fall of objects

> h FreeFall.impact_velocity

                      def impact_velocity(height)

  @spec impact_velocity(number()) :: number()

Computes the velocity on impact given the height
```

The Module documentation contains more information on built-in module attributes.

4 Functions

Functions must always be part of a module. The same naming rules as for variables apply, whereas `?` indicates a predicate function (that returns `true` or `false`), and `!` that a function may cause a runtime error.

Small functions can be written on a single line:

```elixir
defmodule Geometry do
  def rectangle_area(a, b), do: a * b
  def rectangle_perimeter(a, b), do: 2 * a + 2 * b
end
```

Notice the , after the parameter list, the : after do, and the missing end indicator after the function.

4.1 Function Composition

The module SwissGrading computes rounded grades from a number of points achieved and the maximum points achievable (examples/swiss_grading.ex):

```elixir
defmodule SwissGrading do
  def grade(points, max) do
    point_ratio = ratio(points, max)
    temp_grade = multiply(point_ratio, 5)
    exact_grade = add(temp_grade, 1)
    round(exact_grade, 0.1)
  end

  defp ratio(x, y) do
    x / y
  end

  defp multiply(x, y) do
    x * y
  end

  defp add(x, y) do
    x + y
  end

  defp round(x, precision) do
    round(x * 1 / precision) * precision
  end
end
```

Functions defined using defp are private to the module, i.e. not exported.

The grade/2 function uses temporary variables to hand over return values to other functions. The function calls could be nested instead (examples/swiss_grading_nested.ex, omitting the private functions):

```elixir
defmodule SwissGradingNested do
  def grade(points, max) do
    add(multiply(ratio(points, max), 5), 1)
  end

  # omitted private functions

end
```

However, this is not very readable, because the function name (`add`, `multiply`) is optically far removed from its second argument (`5`, `1`).

The pipeline operator `|>` offers a more succinct notation for this purpose:

```elixir
def grade(points, max) do
  points |> ratio(max) |> multiply(5) |> add(1) |> round(0.1)
end
```

For each use of the pipeline, the value of the expression from the left is taken and used as the first argument for the function call on the right.

Longer pipelines are usually spread out over multiple lines (`examples/swiss_grading_piped.ex`, omitting the private functions):

```elixir
defmodule SwissGradingPiped do
  def grade(points, max) do
    points
    |> ratio(max)
    |> multiply(5)
    |> add(1)
    |> round(0.1)
  end

  # omitted private functions

end
```

Comments start with the # character and go to the end of the line. There's no special syntax for multi-line comments, i.e. every line of a multi-line comment has to start with a #.

4.2 Arities and Default Values

The number of arguments a function expects is called the function's *arity*. This number is referred to in the documentation, e.g. `SwissGrading.grade/2` denoting that the `grade()` function of the `SwissGrading` module expects `2` arguments.

Lower-arity functions often use higher-arity functions to perform their work, as `inc/1` does using `inc/2`:

```
defmodule Increment do
  def inc(a, x) do
    a + x
  end

  def inc(a) do
    inc(a, 1)
  end
end
```

The two function definitions can be merged by using a default value for the `x` argument using the `\\` operator (`examples/increment.ex`):

```
defmodule Increment do
  def inc(a, x \\ 1) do
    a + x
  end
end
```

5 Data Types

5.1 Integers

Integers don't have an upper limit:

```
> 123456789 * 987654321 * 123456789 * 987654321
14867566530049990397812181822702361
```

The underscore character can be used as a visual delimiter:

```
> 100_000_000 * 0.753_214_978
75321497.8
```

Integer division and remainder are done using the `Kernel` functions `div` and `rem`:

```
> div(25, 4)
6
> rem(25, 4)
1
```

5.2 Atoms

Atoms are named constants that either start with a colon or an uppercase letter:

```
> :an_atom
:an_atom
> :"an atom with spaces"
:"an atom with spaces"
> AlsoAnAtom
AlsoAnAtom
```

Atoms are prefixed with `Elixir` automatically:

```
> AnAtom == Elixir.AnAtom
true
```

Boolean values are actually atoms:

```
> true == :true
true
> false == :false
true
```

And so is `nil`:

```
> nil == :nil
true
```

Both `nil` and `false` are treated as *falsy*, all the other values as *truthy*, i.e. they evaluate to `false` or `true`, respectively:

```
> nil || false || 4
4
```

5.3 Tuples

Tuples group values together in a collection with a fixed size:

```
> dilbert = {"Dilbert", 42, 120_000}
{"Dilbert", 42, 120000}
```

Elements can be accessed using the `Kernel` function `elem/2`:

```
> elem(dilbert, 0)
"Dilbert"
> elem(dilbert, 2)
120000
```

The `put_elem/3` function doesn't modify the tuple, but returns a copy of it, with the given element replaced:

```
> older_dilbert = put_elem(dilbert, 1, 43)
{"Dilbert", 43, 120000}
```

5.4 Lists

Lists are variable-sized collections to store multiple values

```
> numbers = [3, 7, 8, 2]
[3, 7, 8, 2]
```

They are implemented as linked lists, therefore many operations have a runtime complexity of O(n), as does the `length/1` function:

```
> length(numbers)
4
```

The `in` operator can be used to check whether or not a value is contained in a list:

```
> 7 in numbers
true
> 9 in numbers
false
```

Both the List and the Enum module offer functions for dealing with lists.

A list element can be accessed by its index using the `Enum.at/2` function:

```
> Enum.at(numbers, 0)
3
> Enum.at(numbers, 3)
2
```

Like tuples, lists are immutable. Modified copies of them can be created using functions such as `List.repalce_at/3` and `List.insert_at/3`:

```
> new_numbers = List.replace_at(numbers, 0, 1)
[1, 7, 8, 2]
> more_numbers = List.insert_at(numbers, 2, 4)
[3, 7, 4, 8, 2]
```

Use the index `-1` to append an element at the end of a list:

```
> even_more_numbers = List.insert_at(more_numbers, -1, 5)
[3, 7, 4, 8, 2, 5]
```

Two lists can be concatenated using the ++ operator:

```
> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

5.4.1 Cons Cells

Lists are implemented as *cons cells* (i.e. like in LISP) and support a special head/tail syntax:

```
[head | tail]
```

The head goes to the left side of the `|`, the tail to its right side:

```
> numbers = [1 | [2, 3, 4]]
[1, 2, 3, 4]
```

The `hd` (head) and `tl` (tail) function can be used to access the head (a single value) and the tail (usually a list) of a list:

```
> hd(numbers)
1
> tl(numbers)
[2, 3, 4]
```

Lists with a tail that is not itself a list are called *improper lists*.

The head/tail syntax can be used to push elements at the front of a list with O(1) complexity:

```
> numbers_from_zero = [0 | numbers]
[0, 1, 2, 3, 4]
```

5.5 Maps

Maps are key/value stores commonly used as dynamically sized associative arrays or as records. A map is created using the `%{}` syntax:

```
> empty_map = %{}
%{}
```

A map can be created with initial values defined as key/value pairs:

```
> squares = %{1 => 1, 2 => 4, 3 => 9}
%{1 => 1, 2 => 4, 3 => 9}
```

Using the `Map/new/1` function, a map is created based on a list of key/value tuples:

```
> doubles = Map.new([{1, 2}, {2, 4}, {3, 6}])
%{1 => 2, 2 => 4, 3 => 6}
```

A value can be retrieved by indicating the key in square brackets:

```
> squares[1]
1
> squares[2]
4
```

If the key is not found in the map, `nil` is returned:

```
> squares[4]
nil
```

The `Map.get/3` function accepts a fallback value for this case:

```
> Map.get(squares, 3, :not_found)
9
> Map.get(squares, 4, :not_found)
:not_found
```

The `Map.fetch/2` function indicates whether or not the value was found:

```
> Map.fetch(squares, 3)
{:ok, 9}
> Map.fetch(squares, 4)
:error
```

The `Map.fetch!/2` function (notice the `!`) raises an error if the key indicated is not contained in the given map.

Other useful functions for Maps are `Map.put/3`, `Map.delete/2`, and `Map.update/4`, as well as others in the Map module.

Since maps are enumerables, the functions of the Enum module can be used on them, too.

5.5.1 Records

Maps can be used as records:

```
> dilbert = %{:name => "Dilbert", :age => 42, :job => "Engineer"}
%{age: 42, job: "Engineer", name: "Dilbert"}
> dilbert[:name]
"Dilbert"
```

If the keys are atoms, this shorter syntax can be used:

```
> ashok = %{name: "Ashok", age: 25, job: "Intern"}
%{age: 25, job: "Intern", name: "Ashok"}
> ashok.name
"Ashok"
```

Existing fields can be updated using this special syntax:

```
> promoted_ashok = %{ashok | age: 26, job: "Engineer"}
%{age: 26, job: "Engineer", name: "Ashok"}
```

5.6 Binaries

Binaries are sequences of bytes enclosed in << and >>:

```
> <<1, 16, 128>>
<<1, 16, 128>>
```

Values bigger than 255 (2^8-1) are truncated:

```
> <<255, 256, 257, 511, 512, 513>>
<<255, 0, 1, 255, 0, 1>>
```

The amount of bits to be used for each value can be defined:

```
> <<15::4>>
<<15::size(4)>>
> <<15::4, 12::4>>
<<252>>
```

For the output, the two binaries 15 (1111) and 12 (1100) are normalized (11111100), which results in the value 252.

A sequence of binaries only consisting of items with the size of a single bit is called a bitstring:

```
> <<1::1, 0::1, 1::1, 1::1>>
<<11::size(4)>>
```

The bit sequence 1011 is a decimal 11 in the normalized form.

Multiple binaries can be combined using the <> operator:

```
> <<1, 2, 3>> <> <<4, 5, 6>>
<<1, 2, 3, 4, 5, 6>>
```

5.7 Strings

Elixir has no dedicated string type, but stores them either as binaries or as lists of characters.

5.7.1 Binary Strings

Strings can be defined using double quotes:

```
> name = "Dilbert"
"Dilbert"
```

Expressions can be embedded into strings using `#{}` (string interpolation):

```
> name = "Dilbert"
> age = 42
> profession = "Engineer"
> description = "#{name} is a #{age} years old #{profession}."
"Dilbert is a 42 years old Engineer."
```

Escape sequences such as `\t`, `\n`, `\r`, `\\`, and `\"` are supported, too.

Strings can also be defined using the sigil `~s()`, which allows the use of unescaped double quotes within the string:

```
> IO.puts(~s("Trust me, I'm an engineer!", Dilbert said.))
"Trust me, I'm an engineer!", Dilbert said.
```

The sigil `~S()` ignores interpolation and escaping:

```
> ~S(#{name} is a #{age} years old #{profession}.)
"\#{name} is a \#{age} years old \#{profession}."
> ~S(age:\t42 years)
"age:\\t42 years"
```

The special heredoc syntax supports multi-line strings:

```
> """
> This is on a single line.
> """
"This is on a single line.\n"
```

Since strings are binaries, they can be concatenated using the <> operator:

```
> profession = "Engineer"
> "Dilbert's profession: " <> profession
"Dilbert's profession: Engineer"
```

The String module contains functions for handling (UTF-8) strings.

5.7.2 Character Lists

Strings can also be represented as lists of characters within single quotes:

```
> 'ABC'
'ABC'
```

Which is syntactic sugar for creating a list of their ASCII codes:

```
> [65, 66, 67]
'ABC'
```

If a list consists of numbers representing printable characters, it is displayed as characters.

Character lists are incompatible to binary strings, but offer similar features (escaping, interpolation, sigils, heredocs):

```
> name = "Dilbert"
> age = 42
> IO.puts('Name:\t#{name}\nAge:\t#{age}')
Name:   Dilbert
Age:    42
> IO.puts(~c('My name is #{name}', he said.))
'My name is Dilbert', he said.
> IO.puts(~C('My name is #{name}', he said.))
'My name is #{name}', he said.
```

A character list can be converted into a binary string using `List.to_string/1`:

```
> List.to_string('ABC')
"ABC"
```

A binary string can be converted to a character list using `String.to_charlist/1`:

```
> String.to_charlist("ABC")
'ABC'
```

In general, binary strings should be preferred to character lists. However, some Erlang libraries require the use of character lists, in which case the conversion functions above are helpful.

5.8 First-Class Functions

Functions are first-class citizens; they can be assigned to a variable.

Anonymous functions or lambdas can be created using the `fn` keyword:

```
> twice = fn x -> 2 * x end
```

Calling a lambda requires using the dot operator:

```
> twice.(5)
10
```

Functions can be passed to other functions, e.g. to process lists of items:

```
> Enum.map([1, 2, 3], twice)
[2, 4, 6]
```

The function argument can also be a function literal:

```
> Enum.map([1, 2, 3], fn x -> 2 * x end)
[2, 4, 6]
```

An existing function, like `IO.puts/1`, can be used as a lambda with the capture operator `&`:

```
> Enum.each([1, 2, 3], &IO.puts/1)
1
2
3
```

Lambda expressions can be shortened by using the capture operator and by referring to the n-th parameter as `&[n]` in the function definition:

```
> Enum.map([1, 2, 3], &(2 * &1))
[2, 4, 6]
```

5.8.1 Closures

A lambda function captures variables bound at the time of its definition:

```
> percentage = 75
> get_percentage = fn x -> (percentage / 100) * x end
> percentage = 99
> Enum.map([1, 2, 3], get_percentage)
[0.75, 1.5, 2.25]
```

The percentage 75 is used and not 99, because the first value was bound at the time of the function definition.

5.9 Higher-Level Types

5.9.1 Ranges

A range of numbers can be expressed using the `..` notation:

```
> numbers = 1..10
```

The `in` operator can be used to determine whether or not a number is within a range:

```
> 0 in numbers
false
> 10 in numbers
true
```

A range is an enumerable, and, thus, can be processed using the functions of the `Enum` module:

```
> Enum.each(1..3, &IO.puts/1)
1
2
3
```

5.9.2 Keyword Lists

Some functions such as `IO.inspect/2` expect optional arguments as a keyword list, which can be constructed as a list of atom/value tuples:

```
> options = [{:width, 3}, {:limit, 2}]
> IO.inspect([100, 200, 300], options)
[100,
 200,
 ...]
```

This alternative syntax makes the definition more elegant:

```
> options = [width: 3, limit: 2]
> IO.inspect([100, 200, 300], options)
[100,
 200,
 ...]
```

Or even shorter without an intermediate variable and square brackets:

```
> IO.inspect([100, 200, 300], width: 3, limit: 2)
[100,
 200,
 ...]
```

To write functions using optional arguments, consider the Keyword module (`examples/salary.ex`):

```
defmodule Salary do
  def pay_out(employee, salary, opts \\ []) do
    bonus = Keyword.get(opts, :bonus, 0)
    taxes = Keyword.get(opts, :taxes, 0)
    gross = salary + bonus
    net = gross - gross * taxes
    IO.puts("#{employee} earns $#{net}")
  end
end
```

The function `Salary.pay_out/3` now supports optional keywords:

```
$ iex examples/salary.ex
> Salary.pay_out("Dilbert", 80000)
Dilbert earns $80000
> Salary.pay_out("Dilbert", 80000, bonus: 10000)
Dilbert earns $90000
> Salary.pay_out("Dilbert", 80000, bonus: 10000, taxes: 0.2)
Dilbert earns $7.2e4
```

5.9.3 MapSet

Sets only contain each value once and are implemented as a `MapSet` (module MapSet):

```
> numbers = MapSet.new([1, 3, 6])
#MapSet<[1, 3, 6]>
> numbers = MapSet.put(numbers, 2)
#MapSet<[1, 2, 3, 6]>
> numbers = MapSet.put(numbers, 3)
#MapSet<[1, 2, 3, 6]>
```

A `MapSet` is an enumerable:

```
> Enum.each(numbers, &IO.puts/1)
1
2
3
6
```

5.9.4 Date and Time

Date and time objects can be conveniently be created using sigils:

```
> today = ~D[2021-12-28]
> today.year
2021
> today.month
12
> today.day
28

> lunch = ~T[12:30:00]
lunch.hour
```

```
> lunch.minute
30
> lunch.second
0
```

The Date and Time module contain useful functions to work with those types.

Date and time can be combined to a *naive* date time, i.e. without time zone:

```
> lunch_today = ~N[2021-12-28 12:30:00]
> lunch_today.year
2021
> lunch_today.minute
30
```

A time zone can be added as follows:

```
> lunch_today_utc = DateTime.from_naive!(lunch_today, "Etc/UTC")
~U[2021-12-28 12:30:00Z]
> lunch_today_utc.time_zone
"Etc/UTC"
```

See the modules NaiveDateTime and DateTime.

5.9.5 IO Lists

IO lists are special kinds of lists to build up data for I/O incrementally. They only must consist of integers (0..255), binaries, and other IO lists:

```
> output = [['F', 'o'], 'o'], "ba", 'r']
> IO.puts(output)
Foobar
```

Appending to a list is an O(1) operation, i.e. very efficient:

```
> output = []
[]
> output = [output, "Hello"]
[[], "Hello"]
> output = [output, ", "]
[[[], "Hello"], ", "]
```

```
> output = [output, "World!"]
[[[[], "Hello"], ", "], "World!"]
> IO.puts(output)
Hello, World!
```

6 Operators

Operators are implemented as functions of the `Kernel` module:

```
> 3 + 5
8
> Kernel.+(3, 5)
8
```

Instead of defining lambda functions:

```
> Enum.reduce([1, 2, 3], fn x, y -> x + y end)
6
```

The operator functions of the `Kernel` module can be used:

```
> Enum.reduce([1, 2, 3], &Kernel.+/2)
6
```

Or shorter (`Kernel` is imported automatically):

```
> Enum.reduce([1, 2, 3], &+/2)
6
```

There are comparison operators for weak and strict equality:

```
> 1 == 1.0
true
> 1 === 1.0
false
```

7 Pattern Matching

The matching operator = is more powerful than an assignment operator in other languages. A pattern on the left side matching the expression on the right side creates variable bindings:

```
> employee = {"Dilbert", 42}
> {name, age} = employee
```

7.1 Matching with Constants

The pattern can contain constants that must be matched:

```
> dilbert = {:employee, "Dilbert", 42}
> dogbert = {:consultant, "Dogbert", 7}
> {:employee, name, _} = dilbert
> name
"Dilbert"
> {:employee, name, _} = dogbert
** (MatchError) no match of right hand side value: {:consultant, "Dogbert", 7}
```

For values not to be bound, the anonymous variable (_) can be used in the pattern to ignore them. A variable starting with _ won't be bound, but has a descriptive name:

```
> {:employee, name, _age} = dilbert
```

Functions like `File.read/1` return a tuple of either the form `{:ok, value}` or `{:error, reason}`, which can be matched accordingly:

```
> File.read("/home/patrick/.vimrc")
{:ok, "..."}
> File.read("/home/patrick/.foobar")
{:error, :enoent}
```

7.2 Nested Patterns

Patterns can be nested:

```
> corporation = {:anycorp, {:ceo, "Pointy Haired Boss"}}
> {:anycorp, {:ceo, ceo_name}} = corporation
> ceo_name
"Pointy Haired Boss"
```

7.3 Re-using Bindings

For values that are expected to be equal, the same binding can be used multiple times:

```
> red_rgb = {255, 0, 0}
> {red, other, other} = red_rgb
> red
255
> {value, value, value} = red_rgb
** (MatchError) no match of right hand side value: {255, 0, 0}
```

7.4 Pinning

For matching against the content of a variable, use the pin operator `^`:

```
> redish_color = {255, 34, 78}
> max_rgb = 255
> {^max_rgb, green, blue} = redish_color
> green
34
> blue
78
```

7.5 Matching Lists

Lists can be matched using individual elements:

```
> [a, b, c] = [1, 2, 3]
> a
1
```

Or by splitting the head from the tail:

```
> [head | tail] = [1, 2, 3]
> head
1
> tail
[2, 3]
```

7.6 Matching Maps

Maps can be matched partially:

```
> dilbert = %{name: "Dilbert", age: 42, job: "Engineer"}
> %{name: name} = dilbert
> name
"Dilbert"
```

7.7 Matching Binaries

Binaries can be matched completely:

```
> numbers = <<1, 2, 3>>
> <<a, b, c>> = numbers
> c
3
```

Or using a the special `:: binary` syntax, indicating that the `rest` is a binary of arbitrary length:

```
> <<first, rest :: binary>> = numbers
> first
1
> rest
<<2, 3>>
```

Or using a specified amounts of bits to be matched:

```
> <<a :: 2, b :: 4, c :: 2>> = << 151 >>
> a
2
> b
5
> c
3
```

Here, 151 (`10010111`) is split into 2 (`10`), 5 (`0101`), and 3 (`11`).

7.8 Matching Strings

Since strings are based on binaries, they can be matched the same:

```
> <<a, b, c>> = "ABC"
> a
65
> b
66
> c
67
```

This is error-prone when dealing with unicode strings. Matching the beginning of a string is more practical:

```
> command = "ping paedubucher.ch"
> "ping " <> domain = command
> domain
"paedubucher.ch"
```

7.9 Compound Matches

Matches can be chained to extract values on different levels on a single line:

```
> :calendar.local_time()
{{2022, 1, 7}, {7, 37, 44}}
> {{year, _, _}, {hour, _, _}} = {date, time} = now = :calendar.local_time()
> year
2022
> hour
7
> date
{2022, 1, 7}
> time
{7, 39, 4}
> now
{{2022, 1, 7}, {7, 39, 4}}
```

The sequence doesn't matter, as long as the patterns all match:

```
> {date, time} = {{year, _, _}, {hour, _, _}} = now = :calendar.local_time()
{{2022, 1, 7}, {7, 40, 26}}
```

7.10 Matching Functions

When multiple functions with the same name are available, the arguments are matched against the parameter patterns defined by the functions (`examples/area.ex`):

```
defmodule Area do
  def area({:square, s}) when is_number(s) and s > 0 do
    s * s
  end

  def area({:rectangle, w, h}) when is_number(w) and is_number(h) and w > 0 and h > 0 do
    w * h
  end

  def area({:circle, r}) when is_number(r) and r > 0 do
    :math.pi() * :math.pow(r, 2)
  end

  def area(_) do
    {:invalid_shape}
  end
end
```

Two mechanisms are used to find the proper clause upon a function call:

1. Structural matching of the argument: It must be a tuple beginning with one of the given atom (`:square`, `:rectangle`, etc.).
2. Guards: Constraints such as `is_number` must be fulfilled. (See the Guards documentation for a list of allowed expressions.)

The last clause, `area(_)`, will match any caller providing a single argument:

```
$ iex examples/area.ex
> Area.area({:square, 3})
9
> Area.area({:rectangle, 2, 3})
6
> Area.area({:circle, 5})
78.53981633974483
> Area.area({:triangle, 4, 3, 2})
{:invalid_shape}
> Area.square({:square, 3, 2})
* (UndefinedFunctionError) function Area.square/1 is undefined or private
Area.square({:square, 3, 2})
```

The order of the clauses matters: Make sure that the *catch-all* clause `area(_)` is listed as the last one.

All the clauses of the same arity are captured together:

```
> area_fun = &Area.area/1
> area_fun.({:square, 3})
9
> area_fun.({:rectangle, 3, 2})
6
```

7.10.1 Multiclause Lambdas

Lambdas can also consist of multiple clauses (`examples/testnum.exs`):

```
test_num = fn
  x when is_number(x) and x > 0 ->
    :positive

  x when is_number(x) and x < 0 ->
    :negative

  x when is_number(0) and x == 0 ->
    :zero
end

IO.puts(test_num.(13))
IO.puts(test_num.(-6))
IO.puts(test_num.(0))
```

Notice that clauses are not terminated explicitly; they end when the next clause begins, or the lambda expression ends.

```
$ elixir examples/testnum.exs
positive
negative
zero
```

7.11 Conditionals

These four implementations of FizzBuzz demonstrate different approaches for dealing with conditionals (`examples/fizzbuzz.ex`):

```elixir
defmodule FizzBuzz do
  defmodule UnlessIfElse do
    def fizzbuzz(min, max) when min <= max do
      Enum.each(min..max, &fizzbuzz/1)
    end

    defp fizzbuzz(x) do
      unless rem(x, 3) == 0 or rem(x, 5) == 0 do
        IO.puts(x)
      end

      if rem(x, 15) == 0 do
        IO.puts("FizzBuzz")
      else
        if rem(x, 3) == 0 do
          IO.puts("Fizz")
        else
          if rem(x, 5) == 0 do
            IO.puts("Buzz")
          end
        end
      end
    end
  end

  defmodule Multiclause do
    def fizzbuzz(min, max) when min <= max do
      Enum.each(min..max, &fizzbuzz/1)
    end

    defp fizzbuzz(x) when rem(x, 15) == 0, do: IO.puts("FizzBuzz")
    defp fizzbuzz(x) when rem(x, 3) == 0, do: IO.puts("Fizz")
    defp fizzbuzz(x) when rem(x, 5) == 0, do: IO.puts("Buzz")
    defp fizzbuzz(x), do: IO.puts(x)
  end

  defmodule Cond do
    def fizzbuzz(min, max) when min <= max do
      Enum.each(min..max, &fizzbuzz/1)
    end

    defp fizzbuzz(x) do
      cond do
        rem(x, 15) == 0 -> IO.puts("FizzBuzz")
```

```
        rem(x, 3) == 0 -> IO.puts("Fizz")
        rem(x, 5) == 0 -> IO.puts("Buzz")
        true -> IO.puts(x)
      end
    end
  end

  defmodule Case do
    def fizzbuzz(min, max) when min <= max do
      Enum.each(min..max, &fizzbuzz/1)
    end

    defp fizzbuzz(x) do
      case {rem(x, 3), rem(x, 5)} do
        {0, 0} -> IO.puts("FizzBuzz")
        {0, _} -> IO.puts("Fizz")
        {_, 0} -> IO.puts("Buzz")
        {_, _} -> IO.puts(x)
      end
    end
  end
end
```

All modules have a function `fizzbuzz/2` that takes the lower and upper bounds for a range of numbers to process, and a function `fizzbuzz/1` that deals with an individual number; the latter function being called from the former with each element out of the range. The function `fizzbuzz/1` is implemented using different language constructs in each sub-module:

- `UnlessIfElse` uses branching as known from procedural languages with constructs like `unless`, `if`, and `else`.
- `Multiclause` uses guards to dispatch the function call to the right clause.
- `Cond` makes use of the `cond` construct, which provides a branching facility with multiple alternatives, reminiscent of `if`/`else if` from procedural languages. The last condition, `true`, is similar to the `default` arm in the `switch`/`case` construct from procedural programming languages.
- `Case` makes use of the `case` construct, which is quite similar to `cond`, but works rather like `switch`/`case` than `if`/`else if` from procedural language, because all the arms are based on the initially stated expression. It is far more powerful than `switch`/`case` though, because it uses pattern matching instead of a simple equality check.

The implementation using `cond` is the shortest. The implementation using `case` arguably the clearest; the multiclause implementation the most idiomatic from a functional programming perspective. The implementation using `if`, `unless`, and `else` looks the most convoluted; those constructs are too blunt for dealing with many possibilities.

Notice that all the constructs return a value; however, only the side effect of `IO.puts/1` is of interest in this example.

7.12 With

Consider this list of employees (`examples/users.exs`):

```
employees = [
  %{
    "name" => "Dilbert",
    "username" => "dilbo",
    "password" => "Uyee7oox0OK8johG",
    "email" => "dilbo@corp.com",
    "age" => 42
  },
  %{
    "name" => "Pointy Haired Boss",
    "username" => "theboss",
    "email" => "boss@corp.com",
    "age" => 52,
    "golf_handicap" => 17,
    "cars_owned" => 3
  },
  %{
    "name" => "Wally",
    "username" => "lazybone",
    "password" => "qwerty",
    "email" => "wally@corp.com",
    "age" => 47,
    "years_wasted" => 27
  },
  %{
    "name" => "Dogbert",
    "email" => "doggo@corp.com",
    "age" => 13,
    "current_lawsuits" => 3,
    "allegations" => ["fraud", "arson", "tax evasion"]
  },
  %{
    "name" => "Alice",
    "username" => "alicepro",
    "password" => "IHateThisPlace",
    "email" => "alice@corp.com",
```

```
    "age" => 39
  },
  %{
    "name" => "Catbert",
    "username" => "thecat",
    "password" => "23jd92039d20",
    "age" => 11,
    "years_in_jail" => 5,
    "former_employers" => ["aramco", "facebook"]
  }
]
```

The list's items are heterogenous, i.e. the maps contain different sets of keys: Some contain all the credentials (`"username"`, `"email"`, and `"password"`), some don't. The credentials shall be extracted and printed using this pipeline:

```
employees |> Enum.map(&Credentials.extract/1) |> Enum.each(&IO.inspect/1)
```

The `Credentials` module is implemented as follows (`examples/users.exs`):

```
defmodule Credentials do
  def extract(employee) do
    case extract_username(employee) do
      {:error, reason} ->
        {:error, reason}

      {:ok, username} ->
        case extract_email(employee) do
          {:error, reason} ->
            {:error, reason}

          {:ok, email} ->
            case extract_password(employee) do
              {:error, reason} ->
                {:error, reason}

              {:ok, password} ->
                %{username: username, email: email, password: password}
            end
        end
    end
  end

  defp extract_username(%{"username" => username}), do: {:ok, username}
  defp extract_username(_), do: {:error, "username missing"}
```

```
  defp extract_email(%{"email" => email}), do: {:ok, email}
  defp extract_email(_), do: {:error, "email missing"}

  defp extract_password(%{"password" => password}), do: {:ok, password}
  defp extract_password(_), do: {:error, "password missing"}
end
```

The private helper functions on the bottom are used to extract specific fields. They return `{:ok, value}`, if the desired field is found, and `{:error, reason}` otherwise.

The `extract/1` function stops after the first field of the item isn't found and propagates the error to the caller. This approach using nested `case` constructs doesn't scale well, because there's an additional indentation level for each field to be extracted.

This code can be rewritten using the `with` special form:

```
def extract(employee) do
  with {:ok, username} <- extract_username(employee),
       {:ok, email} <- extract_email(employee),
       {:ok, password} <- extract_password(employee) do
    %{username: username, email: email, password: password}
  end
end
```

The pattern on the left must be matched by the expression on the right. If it matches, the next pattern is matched against the expression; otherwise the expression that didn't match is returned:

```
$ elixir examples/users.exs
%{email: "dilbo@corp.com", password: "Uyee7oox0OK8johG", username: "dilbo"}
{:error, "password missing"}
%{email: "wally@corp.com", password: "qwerty", username: "lazybone"}
{:error, "username missing"}
%{email: "alice@corp.com", password: "IHateThisPlace", username: "alicepro"}
{:error, "email missing"}
```

See the documentation on `with/1` for further details.

8 Iterations

Elixir has no loop constructs such as `while` and `do/while`. Iterations, therefore, must be implemented using recursion.

8.1 Recursion and Tail-Call Optimization

The module `Factorial` implements a factorial function in two ways (`examples/factorial.ex`):

```
defmodule Factorial do
  def factorial(0), do: 1
  def factorial(x) when x > 0, do: x * factorial(x - 1)

  def factorial_tail(0), do: 1
  def factorial_tail(x) when x > 0, do: factorial_tail(x, 1)
  defp factorial_tail(0, acc), do: acc
  defp factorial_tail(x, acc), do: factorial_tail(x - 1, x * acc)
end
```

The first implementation (`factorial`) uses classic iteration. For every recursive function call, a new stack frame is created:

```
Factorial.factorial(5)
    5 * Factorial.factorial(4)
        5 * 4 * Factorial.factorial(3)
            5 * 4 * 3 * Factorial.factorial(2)
                5 * 4 * 3 * 2 * Factorial.factorial(1)
                    5 * 4 * 3 * 2 * 1 * Factorial.factorial(0)
                        5 * 4 * 3 * 2 * 1 * 1
                    5 * 4 * 3 * 2 * 1
                5 * 4 * 3 * 2
            5 * 4 * 6
        5 * 24
    120
```

The first clause is the *basic case*, which is often based on a mathematical definition (e.g. *the factorial of 0 is 1*). The second clause is the *general case*, which makes subsequent calls to itself in order to reduce the problem towards the basic case.

The second implementation (`factorial_tail`) uses tail-call optimization. The intermediate result is carried over using an accumulator parameter. Since the subsequent function call is the last thing the function does, and there's no pending multiplication to be done, the runtime can re-use the existing stack frame:

```
Factorial.factorial_tail(5)
Factorial.factorial_tail(5, 1)
Factorial.factorial_tail(4, 5)
Factorial.factorial_tail(3, 20)
Factorial.factorial_tail(2, 60)
```

```
Factorial.factorial_tail(1, 120)
Factorial.factorial_tail(0, 120)
120
```

Except for very small recursive tasks, recursive functions should be implemented using tail-calls.

Accumulator parameters are an implementation detail. Therefore, two clauses without accumulators are exported. The clauses dealing with accumulators are not exported, and the exported clause for the general case deals with the initialization of the accumulator.

8.2 Higher-Order Functions

Iterations often are performed over existing enumerations of values. The Enum module provides a lot of functions for this purpose.

Consider this `Iteration` module (`examples/iteration.ex`):

```
defmodule Iteration do
  def each([head], func) do
    func.(head)
  end

  def each([head | tail], func) do
    func.(head)
    each(tail, func)
  end
end
```

Which can be used as follows:

```
$ iex examples/iteration.ex
> Iteration.each([1, 2, 3], &IO.puts/1)
1
2
3
```

The same can be achieved using `Enum.each/2` without writing any recursive code:

```
$ iex
> Enum.each([1, 2, 3], &IO.puts/1)
1
2
3
```

Higher-order functions, such as `Enum.each/2`, expect a function as an argument, and/or return a function as their return value. The *filter, map, reduce* pattern is a common combination of such higher-order functions:

- *filter*: Only retain elements matching a certain condition (as defined in a predicate function).
- *map*: Transform each element using the given function into another value.
- *reduce*: Combine all the values to a single one.

`HigherOrder.sum_of_squares/1` accepts an enumeration of values, only retains the numbers (filter), squares them (map), and sums up those values (reduce) in a pipeline (`examples/higher_order/1`):

```
defmodule HigherOrder do
  def sum_of_squares(values) do
    values
    |> Enum.filter(fn v -> is_number(v) end)
    |> Enum.map(fn v -> v * v end)
    |> Enum.reduce(fn v, acc -> v + acc end)
  end
end
```

Which can be used as follows:

```
$ iex examples/higher_order.ex
> HigherOrder.sum_of_squares(["foo", 3, 2, "bar", 4])
29
```

Notice that `Enum.sum/1` could have been used instead of `Enum.reduce/2`:

```
$ iex
> Enum.sum([1, 2, 3])
6
```

`Enum.reduce/3` accepts an additional initialization value for the accumulator:

```
$ iex
> Enum.reduce([1, 2, 3], 100, &+/2)
106
```

The sum of the given enumeration (1 + 2 + 3 = 6) is added up to the provided accumulator of 100. Instead of defining a lambda for summing up two values (`fn a, b -> a + b end`), the `Kernel.+/2` function is captured as a lambda (`&+/2`).

8.3 Comprehensions

Comprehensions are used to iterate over one or many *collectables* (lists, maps, ranges, etc.), thereby producing a new collection.

```
$ iex
> for i <- [1, 2, 3], do: i * 2
[2, 4, 6]
> for j <- 1..5, do: j * j
[1, 4, 9, 16, 25]
```

Reading from multiple collectables is similar to using nested loops in a structured programming language:

```
> for i <- 1..10, j <- 1..10, do: i * j
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 3, 6, 9, 12,
 15, 18, 21, 24, 27, 30, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 5, 10, 15, 20,
 25, 30, 35, 40, 45, 50, ...]
```

Comprehensions can produce collections other than lists by specifying an `into` clause:

```
> for i <- 1..4, j <- 1..4, into: %{}, do: {{i, j}, i * j}
%{
  {1, 1} => 1,
  {1, 2} => 2,
  {1, 3} => 3,
  {1, 4} => 4,
  {2, 1} => 2,
  {2, 2} => 4,
  {2, 3} => 6,
  {2, 4} => 8,
  {3, 1} => 3,
  {3, 2} => 6,
  {3, 3} => 9,
  {3, 4} => 12,
  {4, 1} => 4,
  {4, 2} => 8,
  {4, 3} => 12,
  {4, 4} => 16
}
```

Also, an optional filter clause can be defined (here: `i*j < 10`):

```
> for i <- 1..10, j <- 1..10, i*j < 10, do: {{i, j}, i * j}
[
  {{1, 1}, 1},
  {{1, 2}, 2},
  {{1, 3}, 3},
  {{1, 4}, 4},
  {{1, 5}, 5},
  {{1, 6}, 6},
  {{1, 7}, 7},
  {{1, 8}, 8},
  {{1, 9}, 9},
  {{2, 1}, 2},
  {{2, 2}, 4},
  {{2, 3}, 6},
  {{2, 4}, 8},
  {{3, 1}, 3},
  {{3, 2}, 6},
  {{3, 3}, 9},
  {{4, 1}, 4},
  {{4, 2}, 8},
  {{5, 1}, 5},
  {{6, 1}, 6},
  {{7, 1}, 7},
  {{8, 1}, 8},
  {{9, 1}, 9}
]
```

See for special form for further details.

8.4 Streams

The `Enum` module works *eagerly*, i.e. it performs its work as its functions are invoked on the entire collection. Consider the function `even_fizz_buzz_enum` (`examples/special_numbers.ex`):

```elixir
defmodule SpecialNumbers do
  def even_fizz_buzz_enum(n) do
    1..100
    |> Enum.filter(fn x -> rem(x, 2) == 0 end)
    |> Enum.filter(fn x -> rem(x, 3) == 0 end)
    |> Enum.filter(fn x -> rem(x, 5) == 0 end)
    |> Enum.take(n)
  end
end
```

The function is supposed to return `n` elements that are divisible without remainder by 2, 3, and 5. There are two problems:

First, each invocation of the `filter` function iterates over the entire collection it's called on. (Reversing the order of the filter operations would reduce the workload, because less numbers are divisible by 5 then by 2.)

Second, the function only works for small `n` with the given range:

```
$ iex examples/special_numbers.ex
> SpecialNumbers.even_fizz_buzz_enum(3)
[30, 60, 90]
> SpecialNumbers.even_fizz_buzz_enum(10)
[30, 60, 90]
```

In order to support bigger `n`, the initial range needed to be way bigger, increasing the performance penalty by the multiple iterations necessary.

The `Stream` module is the lazy brother of `Enum`. Rather than performing the operations as requested, its functions keep track of the operations to be performed on each element. An eager operation, like from the `Enum` module, will then finally perform those operations:

Here's the function from above re-implemented using the `Stream` module:

```
defmodule SpecialNumbers do
  def even_fizz_buzz_stream(n) do
    Stream.iterate(1, fn x -> x + 1 end)
    |> Stream.filter(fn x -> rem(x, 2) == 0 end)
    |> Stream.filter(fn x -> rem(x, 3) == 0 end)
    |> Stream.filter(fn x -> rem(x, 5) == 0 end)
    |> Enum.take(n)
  end
end
```

First, the finite range of numbers is replaced by `Stream.iterate/2`, which produces an endless stream of number, each successor value being calculated by the lambda expression based on the initial value.

Second, the `filter/2` function of the `Stream` module is used instead of the one from the `Enum` module, turning the eager operation into a lazy one.

This implementation also supports bigger `n` arguments:

```
$ iex examples/special_numbers.ex
> SpecialNumbers.even_fizz_buzz_stream(3)
[30, 60, 90]
```

```
> SpecialNumbers.even_fizz_buzz_stream(10)
[30, 60, 90, 120, 150, 180, 210, 240, 270, 300]
```

9 Abstraction

In object-oriented languages, methods are called on instances of classes:

```
Employee dilbert = new Employee();
dilbert.setSalary(120000);
```

In Elixir, the functions of a module are used to work on its data type:

```
dilbert = Employee.new()
dilbert = Employee.set_salary(dilbert, 120000)
```

Modifier functions take a data object as the first parameter and return a new data object with the modification applied:

```
new_list = List.insert_at(old_list, -1, :hello)
```

Query functions also take a data object as the first (and often: sole) parameter and return some information about the given data object:

```
name_length = String.length(name)
```

By convention, many modules provide a `new` function, which creates a data object of the type the respective module deals with.

```
days = MapSet.new()
```

Due to the convention that the module's data object is the first parameter, the operations mentioned can be pipelined:

```
$ iex
> MapSet.new() |> MapSet.put(:mo) |> MapSet.put(:tu) |> MapSet.member?(:mo)
true
```

9.1 Basic Abstraction

These principles are applied to create a `Buddies` module, which can be used to manage your friends living in different cities. The module shall be used as follows (`examples/buddies/buddies_v1.exs`):

```
buddies =
  Buddies.new()
  |> Buddies.add_entry("Rome", "Giorgio")
  |> Buddies.add_entry("Rome", "Matteo")
  |> Buddies.add_entry("Moscow", "Yuri")
  |> Buddies.add_entry("Moscow", "Ivan")

Buddies.entries(buddies, "Rome")
|> Enum.each(&IO.puts/1)
```

1. A new data object is created using the `new/0` function.
2. New entries are added using the `add_entry/3` modifier function.
3. The entries of a certain city are returned using the `entries/2` query function.

The module is implemented as follows (`examples/buddies/buddies_v1.exs`):

```
defmodule Buddies do
  def new() do
    %{}
  end

  def add_entry(buddies, city, name) do
    Map.update(buddies, city, [name], fn names -> [name | names] end)
  end

  def entries(buddies, city) do
    Map.get(buddies, city, [])
  end
end
```

1. The `new/0` function returns an empty map, which is the data structure being used to store the buddies with their city.
2. The `add_entry/3` function adds a new buddy with a city. `Map.update/4` provides a powerful API for this purpose:
   - If `city` does not yet exist as a key in the map, the third argument (`[name]`) is used to create the initial value to be stored under that key.
   - If `city` does exist already as a key in the map, an updater lambda is used to update the value. The existing value is passed as the lambda's sole parameter. The new item is added at the front of the existing entries.

3. The `entries/2` function returns the list stored under the given city. An empty list (`[]`) is given as the third argument to `Map.get/3`, which is returned if no elements are stored under `city` in the `buddies` map.

9.2 Composing Abstractions

Storing a list of values under a key can be used in more situations than for storing buddy entries. The details of managing such a map thus can be abstracted away by a new module called `MultiDict` (`examples/buddies/multi_dict.ex`):

```
defmodule MultiDict do
  def new(), do: %{}

  def add(dict, key, value) do
    Map.update(dict, key, [value], &[value | &1])
  end

  def get(dict, key) do
    Map.get(dict, key, [])
  end
end
```

Notice that in `add/3`, the updater lambda has been re-written using the capture operation. The `Buddies` module can now be expressed more simply in terms of `MultiDict` (`examples/buddies/buddies_v2.exs`):

```
defmodule Buddies do
  def new(), do: MultiDict.new()

  def add_entry(buddies, city, name) do
    MultiDict.add(buddies, city, name)
  end

  def entries(buddies, city) do
    MultiDict.get(buddies, city)
  end
end
```

The `MultiDict` module has to be compiled so that `Buddies` can make use of it:

```
$ cd examples/buddies
$ elixirc multi_dict.ex
$ elixir buddies_v2.exs
Matteo
```

```
Giorgio
```

9.3 Structuring Data with Maps

The current implementation is tedious to extend: If an additional field should be stored, multiple functions needed to extended. It's therefore more flexible to store the data in a map (`examples/buddies/buddies_v3.exs`):

```
defmodule Buddies do
  def new(), do: MultiDict.new()

  def add_entry(buddies, entry) do
    MultiDict.add(buddies, entry.city, entry)
  end

  def entries(buddies, city) do
    MultiDict.get(buddies, city)
  end
end
```

The `MultiDict` module works without any modifications. The client code, however, becomes more verbose, because a map with keys needs to be provided:

```
buddies =
  Buddies.new()
  |> Buddies.add_entry(%{city: "Rome", name: "Giorgio"})
  |> Buddies.add_entry(%{city: "Rome", name: "Matteo"})
  |> Buddies.add_entry(%{city: "Moscow", name: "Yuri"})
  |> Buddies.add_entry(%{city: "Moscow", name: "Ivan"})

Buddies.entries(buddies, "Rome")
|> Enum.each(&IO.inspect/1)
```

9.4 Abstracting with Structs

A struct is a special kind of map that defines which fields can be stored in it. A module can have at most one struct, and a struct belongs to one module. For die `Buddies` module, a sub-module called `Entry` is defined, which contains the struct definition and the function `new/2` to create a new one (`examples/buddies/buddies_v4.exs`):

```
defmodule Buddies do
  def new(), do: MultiDict.new()
```

```
  def add_entry(buddies, entry) do
    MultiDict.add(buddies, entry.city, entry)
  end

  def entries(buddies, city) do
    MultiDict.get(buddies, city)
  end

  defmodule Entry do
    defstruct city: nil, name: nil

    def new(city, name) do
      %Entry{city: city, name: name}
    end
  end
end
```

The client code not necessarily becomes shorter, but safer:

```
buddies =
  Buddies.new()
  |> Buddies.add_entry(Buddies.Entry.new("Rome", "Giorgio"))
  |> Buddies.add_entry(Buddies.Entry.new("Rome", "Matteo"))
  |> Buddies.add_entry(Buddies.Entry.new("Moscow", "Yuri"))
  |> Buddies.add_entry(Buddies.Entry.new("Moscow", "Ivan"))

Buddies.entries(buddies, "Rome")
|> Enum.each(&IO.inspect/1)
```

A struct can be matched by a map:

```
$ iex examples/buddies/buddies_v4.exs
> entry = Buddies.Entry.new("Bern", "Urs")
> %{city: city, name: name} = entry
> city
"Bern"
> name
"Urs"
```

However, a map cannot be matched by a struct:

```
> %Buddies.Entry{city: city, name: name} = %{city: "Bern", name: "Urs"}
** (MatchError) no match of right hand side value: %{city: "Bern", name: "Urs"}
```

This is because a struct has an additional field called `__struct__` indicating the type:

```
> Map.to_list(entry)
[__struct__: Buddies.Entry, city: "Bern", name: "Urs"]
```

Existing fields of a struct can be updated with the same syntax as for a map:

```
> entry = Buddies.Entry.new("Bern", "Urs")
> entry = %Buddies.Entry{entry | city: "Biel"}
%Buddies.Entry{city: "Biel", name: "Urs"}
```

Since structs *are* maps, functions of the `Map` module can be used on structs. However, structs do not implement the enumerable protocol, so functions of the `Enum` module cannot be used on them:

```
> Enum.member?(entry, :city)
** (Protocol.UndefinedError) protocol Enumerable not implemented for %Buddies.Entry{[…]
```

Records are similar to structs, but based on tuples instead of maps. They are used for interoperability with Erlang libraries that work with records themselves.

9.5 CRUD Operations

The above `Buddies` module supports creating new and reading existing entries. In order to implement an update and a delete operation, the entries must be identified using a unique value (`examples/buddies/buddies_v5.exs`):

```
defmodule Buddies do
  defstruct auto_id: 1, entries: %{}

  def new(), do: %Buddies{}

  def add_entry(buddies, entry) do
    entry = Map.put(entry, :id, buddies.auto_id)
    new_entries = Map.put(buddies.entries, buddies.auto_id, entry)
    %Buddies{buddies | entries: new_entries, auto_id: buddies.auto_id + 1}
  end

  def entries(buddies, city) do
    buddies.entries
    |> Stream.filter(fn {_, entry} -> entry.city == city end)
    |> Enum.map(fn {_, entry} -> entry end)
```

```
  end

  def update_entry(buddies, entry_id, updater_fun) do
    case Map.fetch(buddies.entries, entry_id) do
      :error ->
        buddies

      {:ok, old_entry} ->
        new_entry = %{id: ^entry_id} = updater_fun.(old_entry)
        new_entries = Map.put(buddies.entries, entry_id, new_entry)
        %Buddies{buddies | entries: new_entries}
    end
  end

  def delete_entry(buddies, entry_id) do
    new_entries = Map.filter(buddies.entries, fn {_, e} -> e.id != entry_id end)
    %Buddies{buddies | entries: new_entries}
  end
end
```

Notice the following changes and additions:

1. Instead of re-using the `MultiDict` module, `Buddies` defines its own struct consisting of `auto_id` (the identifier for the next new entry) and `entries` (the actual entries, which used to be contained in the `MultiDict` in earlier versions).
2. The `add_entry/2` function now puts the computed `auto_id` as the `:id` attribute into the entry to be added. A new struct is returned with the entries now containing the new entry, and the `auto_id` being incremented.
3. The `entries/2` function filters the entry by the given city and then extracts the entry from the key/value pair. (The `id` is only used internally.)
4. The `update_entry/3` function looks up an entry with the given `entry_id`.
    - If the lookup fails (`:error`), the old struct is returned.
    - The the lookup succeeds, the `old_entry` is updated using a provided updater lambda function. To make sure that this lambda does not alter the `id` of the entry, it is matched against `%{id: ^entry_id}` with the given `entry_id` being pinned. The updated entry is stored in the in `buddies.entries`, which is then used to replace the old `entries`.
5. The `delete_entry/2` function filters out the entry with the given `entry_id` and returns the entries not containing the one with the given id.

The client code looks as follows:

```
buddies =
  Buddies.new()
  |> Buddies.add_entry(%{city: "Palermo", name: "Vito"})
```

```
  |> Buddies.add_entry(%{city: "Bern", name: "Urs"})

Enum.each(buddies.entries, &IO.inspect/1)

buddies =
  buddies
  |> Buddies.delete_entry(2)
  |> Buddies.update_entry(1, fn e -> Map.put(e, :name, "Don Corleone") end)

Enum.each(buddies.entries, &IO.inspect/1)
```

Notice the lambda replacing the `:name` of the entry with the identifier `1`. The output then looks as follows:

```
{1, %{city: "Palermo", id: 1, name: "Vito"}}
{2, %{city: "Bern", id: 2, name: "Urs"}}
{1, %{city: "Palermo", id: 1, name: "Don Corleone"}}
```

9.6 Protocols

Consider the module `Person` (`examples/person.ex`):

```
defmodule Person do
  defstruct name: "" , age: 0
  def new(name, age) do
    %Person{name: name, age: age}
  end
end
```

Since the data is stored in a map, it's possible to print a `Person` using `IO.inspect/1`, but `IO.puts/1` fails:

```
$ iex examples/person.ex
> dilbert = Person.new("Dilbert", 42)
> IO.inspect(dilbert)
%Person{age: 42, name: "Dilbert"}
> IO.puts(dilbert)
** (Protocol.UndefinedError) protocol String.Chars not implemented for %Person{age: 42, […]
```

The `Person` module does not implement the `String.Chars` *protocol*, which is defined as follows:

```
defprotocol String.Chars do
    def to_string(thing)
end
```

Notice the `def` macro that does not define a function body, but only the function header (name and arity).

A protocol can be implemented using the `defimpl` macro:

```
defimpl String.Chars, for: Person do
  def to_string(thing) do
    "#{thing.name} (age: #{thing.age})"
  end
end
```

Which makes it possible to use `IO.puts/1` on `Person` now:

```
> dilbert = Person.new("Dilbert", 42)
> IO.inspect(dilbert)
%Person{age: 42, name: "Dilbert"}
> IO.puts(dilbert)
Dilbert (age: 42)
```

`IO.puts/1` is not aware of the different types and how they are supposed to be printed, it just delegates the function call to `String.Chars.to_string/1`, which dispatches the call to the type implementing the protocol.

It's possible to define protocol implementations for any type; including for the `Any` fallback in order to provide a default implementation.

Consider implementing the following protocols for your types:

- `String.Chars` to provide a string representation for output (e.g. with `IO.puts/1`).
- `Inspect` to provide a detailed string representation for your type (for `IO.inspect/1`).
- `Enumerable` so that your type works with the functions of `Enum` and `Stream`.
- `Collectable` so that you can collect data into your type using comprehensions.

10 Concurrency Primitives

A *process* is a lightweight unit of execution in the Erlang virtual machine, as opposed to the heavy process of the operating system. Every process has its own identifier (the PID), which is assigned upon starting it using the `spawn/1` function. A process also has a mailbox, queueing up a (theoretically) unlimited amount of messages. If the PID of a process is known, a message can be sent to it using the `send/2` function. The messages are processed using `receive`.

10.1 Stateless Server Process

This small echo server demonstrates those principles (`examples/echoserver.exs`):

```
defmodule Echo do
  def loop() do
    receive do
      {:message, payload} -> IO.puts(payload)
      {:important_message, payload} -> IO.puts(String.upcase(payload))
      unknown -> IO.puts(:stderr, "unsupported message format")
    end

    loop()
  end
end

printer = spawn(&Echo.loop/0)

send(printer, {:message, "Hello"})
send(printer, {:message, "World"})
send(printer, {:important_message, "and beyond"})
send(printer, {:unknown, "Universe"})
send(printer, {:message, "Goodbye"})

Process.sleep(1000)
```

- The `Echo` module defines a single function `loop/0`, which handles the incoming messages. The `receive` construct matches the incoming message against different patterns and deals with them accordingly. The catch-all pattern (`unknown`) ensures that no messages are queueing up ad infinitum. Since `receive` only awaits a single message, the `loop/0` function calls itself, so that the next message can be dealt with.
- A process is created using the `spawn` function by indicating the `Echo.loop/0` function; a PID is returned and saved in `printer`.
- Various messages are sent to the `printer` by handing over its PID and a message. Since the Echo server runs asynchronously, the `Process.sleep/1` call makes sure that the `Echo` process has enough time to finish its work.

The echo server is run as follows:

```
$ elixir examples/echoserver.exs
Hello
World
AND BEYOND
Goodbye
unsupported message format
```

10.2 Stateful Server Process (Simple State)

A server process can handle state by passing it to subsequent calls of its `loop` function. An initial state can be provided by calling `loop` with an argument; modified state is provided by further recursive calls to `loop`. Consider this `BankAccount` module (`examples/paymentserver.exs`):

```
defmodule BankAccount do
  def pay_async(pid, payment) do
    case payment do
      {:incoming, amount} -> send(pid, {:pay_in, amount, self()})
      {:outgoing, amount} -> send(pid, {:pay_out, amount, self()})
    end
  end

  def query_balance(pid) do
    send(pid, {:query, self()})
  end

  def get_result() do
    receive do
      {:ok, balance} -> {:ok, balance}
    after
      1000 -> {:err, :timeout}
    end
  end

  def start(balance) do
    spawn(fn -> loop(balance) end)
  end

  defp loop(balance) do
    balance =
      receive do
        {:query, pid} ->
          send(pid, {:ok, balance})
          balance

        {:pay_in, amount, _} ->
          balance + amount

        {:pay_out, amount, _} ->
          new_balance = balance - amount

          if new_balance >= 0 do
```

```
            new_balance
          else
            balance
          end
        end

      loop(balance)
    end
  end
```

- The `pay_async/2` function expects a `pid` and a `payment` message. This message is forwarded to itself, changing the atom (`:incoming` to `:pay_in`, `:outgoing` to `:pay_out`) and enriching the message with the PID of the main process, which is gotten hold of by a call to `self/0`. (The response is supposed to be sent back to that particular PID later on.)
- The `query_balance/1` function also expects a `pid`, to which it sends a `:query` message, also providing its PID for getting the response.
- The `get_result/0` function awaits the incoming answer to the balance query. If it is not answered within 1000 milliseconds (`after` clause), a `:timeout` error is returned.
- The `start/1` function is used to spawn the new process. The given initial state (`balance`) is provided by a lambda.
- The `loop/1` function handles the incoming messages. Since `receive` is an expression, the (potentially) modified `balance` stores its returned value. The first clause catches `:query` results, which are sent back to the caller PID. The current balance is returned unmodified. The second clause catches incoming payments (`:pay_in`), which are added to the balance. The third clause (`:pay_out`) first checks the current balance, and only reduces it by the given `amount` if the new balane would not be lower than zero. The `loop` is called again with the (potentially) updated `balance`.

In this example, multiple processes are created, which run independently of each other:

```
accounts = %{
  "Dilbert" => BankAccount.start(25_200),
  "Alice" => BankAccount.start(52_900),
  "Wally" => BankAccount.start(12_500)
}

# random spending 1..1000 ten times
1..10
|> Enum.each(fn _ ->
  {name, account} = Enum.random(accounts)
  amount = :rand.uniform(1000)
  BankAccount.pay_async(account, {:outgoing, amount})
  IO.puts("#{name} spent #{amount}.")
end)
```

```
# random salary 7000..9000
accounts
|> Enum.each(fn {name, account} ->
  salary = :rand.uniform(2000) + 7000
  BankAccount.pay_async(account, {:incoming, salary})
  IO.puts("#{name} received a salary of #{salary}.")
end)

accounts
|> Enum.each(fn {name, account} ->
  BankAccount.query_balance(account)

  case BankAccount.get_result() do
    {:ok, balance} ->
      IO.puts("At the end of the month, #{name} has a balance of #{balance}.")

    {:err, :timeout} ->
      IO.puts("Retrieval of balance of #{name}'s account failed with a timeout.")
  end
end)
```

Which produces the following output:

```
Alice spent 230.
Alice spent 538.
Alice spent 463.
Alice spent 491.
Wally spent 748.
Alice spent 112.
Alice spent 883.
Alice spent 712.
Alice spent 365.
Wally spent 220.
Alice received a salary of 8512.
Dilbert received a salary of 7684.
Wally received a salary of 8518.
At the end of the month, Alice has a balance of 57618.
At the end of the month, Dilbert has a balance of 32884.
At the end of the month, Wally has a balance of 20050.
```

10.3 Stateful Server Process (Nested State)

The state of a server process is not restricted to simple atomic variables such as numbers. An existing module, such as the `Buddies` module (`examples/buddies/buddies_v5.exs`), can be used as the foundation for such a server (`examples/buddies/buddies_v6.exs`):

```
defmodule BuddyServer do
  def start() do
    spawn(fn -> loop(Buddies.new()) end)
  end

  def add_entry(entry) do
    send(:buddy_server, {:add, entry})
  end

  def update_entry(entry_id, updater_fun) do
    send(:buddy_server, {:upd, entry_id, updater_fun})
  end

  def delete_entry(entry_id) do
    send(:buddy_server, {:del, entry_id})
  end

  def entries(city) do
    send(:buddy_server, {:entries, city, self()})

    receive do
      {:ok, entries} -> entries
    end
  end

  defp loop(buddies) do
    new_buddies =
      receive do
        {:add, entry} ->
          Buddies.add_entry(buddies, entry)

        {:upd, entry_id, updater_fun} ->
          Buddies.update_entry(buddies, entry_id, updater_fun)

        {:del, entry_id} ->
          Buddies.delete_entry(buddies, entry_id)

        {:entries, city, pid} ->
```

```
            send(pid, {:ok, Buddies.entries(buddies, city)})
            buddies
        end

      loop(new_buddies)
    end
end
```

The struct defined in the `Buddies` module is passed around in between loops. The messages (`:add`, `:upd`, `:del`, and `:entries`) are forwarded as calls to the underlying `Buddies` module (`add_entry/2`, `update_entry/3`, `delete_entry/2`, and `entries/2`, respectively).

Notice that the server sends its messages to an atom `:buddy_server` instead of to a specific PID. This is because the server process is registered under that atom before:

```
pid = BuddyServer.start()
Process.register(pid, :buddy_server)
```

The server is then used as follows:

```
IO.puts("adding some buddies")
BuddyServer.add_entry(%{city: "Berlin", name: "Hans"})
BuddyServer.add_entry(%{city: "Berlin", name: "Hermann"})
BuddyServer.add_entry(%{city: "Palermo", name: "Vito"})

IO.puts("entries from Berlin")
Enum.each(BuddyServer.entries("Berlin"), &IO.inspect/1)

IO.puts("entries from Berlin after deleting entry with id 2")
BuddyServer.delete_entry(2)
Enum.each(BuddyServer.entries("Berlin"), &IO.inspect/1)

IO.puts("entries from Palermo after updating entry with id 3")
BuddyServer.update_entry(3, fn e -> Map.put(e, :name, "Don Corleone") end)
Enum.each(BuddyServer.entries("Palermo"), &IO.inspect/1)
```

Which produces this output:

```
adding some buddies
entries from Berlin
%{city: "Berlin", id: 1, name: "Hans"}
%{city: "Berlin", id: 2, name: "Hermann"}
entries from Berlin after deleting entry with id 2
%{city: "Berlin", id: 1, name: "Hans"}
entries from Palermo after updating entry with id 3
%{city: "Palermo", id: 3, name: "Don Corleone"}
```

11 Generic Server Process

Server processes have some duties in common, such as spawning a process, running the message loop while carrying over state, dispatching messages etc. It therefore is beneficial to separate the common server concerns from the actual business logic.

11.1 Implementing a Generic Server Process

The module `ServerProcess` (`examples/server_process/v1/server_process.ex`) implements a generic server process, which relies on another module to perform the business logic:

```
defmodule ServerProcess do
  def start(callback_module) do
    spawn(fn ->
      initial_state = callback_module.init()
      loop(callback_module, initial_state)
    end)
  end

  def call(server_pid, request) do
    send(server_pid, {request, self()})

    receive do
      {:response, response} -> response
    end
  end

  defp loop(callback_module, current_state) do
    receive do
      {request, caller} ->
        {response, new_state} =
          callback_module.handle_call(
            request,
            current_state
          )

        send(caller, {:response, response})
        loop(callback_module, new_state)
    end
  end
end
```

- The `start/1` function accepts a module reference and spawns a process. The initial state is provided by the callback module's `init/0` function, which must be provided. The initial state is then passed into the message loop.
- Synchronuous messages are handled using the `call/2` function, which reqires a PID and a `request`. This request is sent to the indicated process; the message is enriched using the current PID. The answer is awaited synchronously using `receive`, and the response message is returned to the caller.
- The `loop/2` function keeps track of the callback module and the state. A request is simply forwarded to the callback module's `handle_call/2` function, which must be provided. The response returned thereby is forwarded to the calling process, and the loop is run with the updated state.

Any module providing `init/0` and `handle_call/2` can be using together with this generic server process, e.g. this simple `KeyValueStore` (`examples/server_process/v1/key_value_store.ex`):

```
defmodule KeyValueStore do
  def init do
    %{}
  end

  def handle_call({:put, key, value}, state) do
    {:ok, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, state) do
    {Map.get(state, key), state}
  end
end
```

- The `init/0` function just creates an empty map as the initial state.
- The `handle_call/2` function has two clauses:

    1. One to handle `:put` calls to add a value to the map.
    2. One to handle `:get` calls, which returns the value for a given key.

The modules can be used together as follows:

```
$ cd examples/server_process/v1/
$ elixirc key_value_store.ex key_value_store.ex
$ iex
> pid = ServerProcess.start(KeyValueStore)
> ServerProcess.call(pid, {:put, :name, "Dilbert"})
:ok
> ServerProcess.call(pid, {:get, :name})
"Dilbert"
```

11.2 Transparent Use of Server Process

In the current implementation of `KeyValueStore`, the client has to deal explicitly with the `ServerProcess` module. This can be made transparently by wrapping the calls to `ServerProcess` with the *interface functions* `start/0`, `put/3`, and `get/2` in `KeyValueStore` (`examples/server_process/v2/key_value_store.ex`):

```
defmodule KeyValueStore do
  def start do
    ServerProcess.start(KeyValueStore)
  end

  def put(pid, key, value) do
    ServerProcess.call(pid, {:put, key, value})
  end

  def get(pid, key) do
    ServerProcess.call(pid, {:get, key})
  end

  def init do
    %{}
  end

  def handle_call({:put, key, value}, state) do
    {:ok, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, state) do
    {Map.get(state, key), state}
  end
end
```

Which can be used as follows:

```
$ cd examples/server_process/v2/
$ elixirc key_value_store.ex key_value_store.ex
$ iex
> pid = KeyValueStore.start()
> KeyValueStore.put(pid, :name, "Wally")
:ok
> KeyValueStore.get(pid, :name)
"Wally"
```

Notice that the *interface functions* run in the client process, whereas the *handler functions* run in the server process.

11.3 Asynchronous Requests

The `get` operation needs to be synchronous, since the client wants the response right away. The `put` operation, on the other side, could be made asynchronous, working in a *fire and forget* manner. In Erlang/OTP, those kinds of requests are called *call* (synchronous) and *cast* (asynchronous).

The server process needs to be updated in order to support both call and cast requests (`examples/server_process/v3/server_process.ex`):

```
defmodule ServerProcess do
  def start(callback_module) do
    spawn(fn ->
      initial_state = callback_module.init()
      loop(callback_module, initial_state)
    end)
  end

  def call(server_pid, request) do
    send(server_pid, {:call, request, self()})

    receive do
      {:response, response} -> response
    end
  end

  def cast(server_pid, request) do
    send(server_pid, {:cast, request})
  end

  defp loop(callback_module, current_state) do
    receive do
      {:call, request, caller} ->
        {response, new_state} =
          callback_module.handle_call(
            request,
            current_state
          )

        send(caller, {:response, response})
        loop(callback_module, new_state)
```

```
      {:cast, request} ->
        new_state =
          callback_module.handle_cast(
            request,
            current_state
          )

        loop(callback_module, new_state)
    end
  end
end
```

- The message sent by the `call/2` function now includes the `:call` atom to indicate the messaging mode. In the message loop, the `:call` atom is used for matching those messages.
- A new function `cast/2` is introduced, which sends messages prefixed with the `:cast` atom to the server process. Those messages are handled in the event loop in a way that only the state is updated, but no message is sent back to the caller.

The `KeyValueStore` module must be updated accordingly in order to support `cast` operations (`examples/server_process/v3/key_value_store.ex`):

```
defmodule KeyValueStore do
  def start do
    ServerProcess.start(KeyValueStore)
  end

  def put(pid, key, value) do
    ServerProcess.cast(pid, {:put, key, value})
  end

  def get(pid, key) do
    ServerProcess.call(pid, {:get, key})
  end

  def init do
    %{}
  end

  def handle_cast({:put, key, value}, state) do
    Map.put(state, key, value)
  end

  def handle_call({:get, key}, state) do
    {Map.get(state, key), state}
```

```
  end
end
```

- The `put/3` function now calls `ServerProcess.cast/2`.
- The first clause of the `handle_call/2` function has been replaced with a new `handle_cast/2` function.

The new implementation now can be used as follows:

```
$ cd examples/server_process/v3
$ elixirc key_value_store.ex server_process.ex
$ iex
> pid = KeyValueStore.start()
> KeyValueStore.put(pid, :name, "Dogbert")
> KeyValueStore.get(pid, :name)
"Dogbert"
```

It's now up to the interface functions to decide to operate synchronously or asynchronously.

11.4 GenServer

There's no need to implement a generic server process manually. It is already available through the `GenServer` module (based on Erlang's `:gen_server`). `GenServer` is a behaviour, which requires the implementation of various functions, such as `handle_call/3`, `handle_cast/2`, and `init/1`. There's no need to implement all of them, since default implementations can be taken from `GenServer` with the `use` macro as follows (`examples/key_value_gen_server.exs`):

```
defmodule KeyValueStore do
  use GenServer

  def start do
    GenServer.start(KeyValueStore, nil)
  end

  def put(pid, key, value) do
    GenServer.cast(pid, {:put, key, value})
  end

  def get(pid, key) do
    GenServer.call(pid, {:get, key})
  end

  def init(_) do
```

```
    {:ok, %{}}
  end

  def handle_cast({:put, key, value}, state) do
    {:noreply, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, _, state) do
    {:reply, Map.get(state, key), state}
  end
end

{:ok, pid} = KeyValueStore.start()
KeyValueStore.put(pid, :name, "Dilbert")
name = KeyValueStore.get(pid, :name)
IO.puts(name)
```

- The `start/0` function delegates the startup to `GenServer` by providing the module referenec. No second parameter is needed, but would be handed over to `init/1`, if provided.
- The `put/3` and `get/2` functions are interface functions that delegate their calls to `GenServer` using `cast` (asynchronous) or `call` (synchronous).
- The `handle_cast/2` function is the callback for asynchronous calls, and, by convention, returns a tuple starting with `:noreply`.
- The `handle_call/3` function is the callback for synchronous calls. Its second parameter is an internal request ID, which can be ignored here. It's returned tuple starts with the `:reply` atom by convention.

Notice that `GenServer.start/2` retrurns a tuple, indicating whether or not the process started successfylly. After creation, the module can be used through its interface function; its use of `GenServer` is completely transparent.

Check out the `GenServer` documentation for further information.

11.5 Worker Pools

The `PrimeServer` module uses `GenServer` to find prime numbers using the `PrimeNumbers` module (`examples/prime_genserver.exs`):

```
defmodule PrimeNumbers do
  def is_prime(x) when is_number(x) and x == 2, do: true

  def is_prime(x) when is_number(x) and x > 2 do
    from = 2
    to = trunc(:math.sqrt(x))
```

```
    n_total = to - from + 1

    n_tried =
      Enum.take_while(from..to, fn i -> rem(x, i) != 0 end)
      |> Enum.count()

    n_total == n_tried
  end

  def is_prime(x) when is_number(x), do: false
end

defmodule PrimeServer do
  use GenServer

  def start() do
    GenServer.start(__MODULE__, nil)
  end

  def is_prime(pid, x) do
    GenServer.call(pid, {:is_prime, x})
  end

  def init(_) do
    {:ok, %{}}
  end

  def handle_call({:is_prime, x}, _, state) do
    {:reply, PrimeNumbers.is_prime(x), state}
  end
end

{:ok, pid} = PrimeServer.start()

1..20
|> Stream.filter(fn i -> PrimeServer.is_prime(pid, i) end)
|> Enum.each(fn i -> IO.puts("#{i} is a prime number.") end)
```

Notice that the work is not parallelized, because all the calls are handled synchronously.

```
$ elixir examples/prime_genserver.exs
2 is a prime number.
5 is a prime number.
7 is a prime number.
```

```
11 is a prime number.
13 is a prime number.
17 is a prime number.
19 is a prime number.
```

In order to find prime numbers with multiple processes performing work in parallel, so-called *worker processes* are needed. In this more involved example (`examples/prime_workers.exs`), concurrency primitives are used to get more fine-grained control over the distribution of the work:

```
defmodule PrimeNumbers do
  def is_prime(x) when is_number(x) and x == 2, do: true

  def is_prime(x) when is_number(x) and x > 2 do
    from = 2
    to = trunc(:math.sqrt(x))
    n_total = to - from + 1

    n_tried =
      Enum.take_while(from..to, fn i -> rem(x, i) != 0 end)
      |> Enum.count()

    n_total == n_tried
  end

  def is_prime(x) when is_number(x), do: false
end

defmodule PrimeWorker do
  def start() do
    spawn(fn -> loop() end)
  end

  defp loop() do
    receive do
      {:is_prime, x, pid} ->
        send(pid, {:prime_result, x, PrimeNumbers.is_prime(x)})
        loop()

      {:terminate, pid} ->
        send(pid, {:done})
    end
  end
end
```

```
defmodule PrimeClient do
  def start() do
    spawn(fn -> loop(0) end)
  end

  def loop(found) do
    receive do
      {:prime_result, _, prime} ->
        if prime do
          loop(found + 1)
        else
          loop(found)
        end

      {:query_primes, pid} ->
        send(pid, {:primes_found, found})
    end

    loop(found)
  end
end

args = System.argv()
[n, n_workers | _] = args
{n, ""} = Integer.parse(n, 10)
{n_workers, ""} = Integer.parse(n_workers, 10)

client = PrimeClient.start()

workers =
  for i <- 0..(n_workers - 1),
      into: %{},
      do: {i, PrimeWorker.start()}

Enum.each(2..n, fn x ->
  i_worker = rem(x, n_workers)
  worker = Map.get(workers, i_worker)
  send(worker, {:is_prime, x, client})
end)

workers
|> Enum.each(fn {_, w} ->
  send(w, {:terminate, self()})
```

```
  receive do
    {:done} -> {:nothing}
  end
end)

send(client, {:query_primes, self()})

receive do
  {:primes_found, found} ->
    IO.puts("Found #{found} primes from 2 to #{n}.")
end
```

The workers are picked fairly for every number to be processed based on a modulo calculator (`rem/2`). The program can be started with command-line arguments (the upper limit of the range to find prime numbers in, and the number of worker processes to be used):

```
$ elixir examples/prime_workers.exs 1000000 1
Found 78497 prime numbers from 2 to 1000000.
```

The program speeds up considerably if *more* workers are used. On a computer with eight cores, using 7 workers seems to sound reasonable (leaving 1 core for scheduling and other tasks):

```
$ time elixir examples/prime_workers.exs 1000000 7
Found 78497 prime numbers from 2 to 1000000.

real    0m16.531s
user    1m32.998s
sys     0m9.431s
```

However, the CPU usage never completely maxes out, so using more workers might be a good idea:

```
$ time elixir examples/prime_workers.exs 1000000 100
Found 78497 prime numbers from 2 to 1000000.

real    0m10.679s
user    1m21.081s
sys     0m0.477s
```

100 might sound like a lot, but since BEAM processes are lightweight, even more of them can be spawned:

```
$ time elixir examples/prime_workers.exs 1000000 1000
Found 78497 prime numbers from 2 to 1000000.

real    0m1.746s
user    0m10.579s
sys     0m0.300s
```

Using 1000 instead of 100 workers leads to a considerable speedup. One possible interpretation is that the task to be performed by a process (figuring out whether or not a particular number is a prime number) is relatively small, so it's a good idea to have many such tasks queued up for the scheduler.

12 Error Handling

There are three types of runtime errors:

1. errors
    - dividing a number by zero
    - calling a non-existing function
    - no matching pattern
    - `raise("something went wrong")`
    - functions throwing errors end their name with `!` (e.g. `File.open!`)
2. exits
    - a process is exited
    - `exit("process exits")`
3. throws
    - for non-local returns (resembles `break`, `goto` in procedural languages)
    - `throw(:thrown_value)`

12.1 `try/catch`

Any of those errors can be caught using the `try/catch` special form:

`examples/error_handling.ex`:

```
defmodule ErrorHandling do
  def execute(f) do
    try do
      f.()
      IO.puts("ok")
```

```
    catch
      type, value ->
        IO.puts("Error: #{inspect(type)} #{inspect(value)}")
    end
  end
end
```

```
$ iex examples/error_handling.ex

> ErrorHandling.execute(fn -> 3 / 5 end)
ok

> ErrorHandling.execute(fn -> 3 / 0 end)
Error: :error :badarith

> ErrorHandling.execute(fn -> raise("whatever") end)
Error: :error %RuntimeError{message: "whatever"}

> ErrorHandling.execute(fn -> :erlang.error("raw error") end)
Error: :error "raw error"

> ErrorHandling.execute(fn -> exit("ciao") end)
Error: :exit "ciao"

> ErrorHandling.execute(fn -> throw(:goodbye) end)
Error: :throw :goodbye
```

12.2 `try/catch` Expression

The `try/catch` special form is an expression and returns the value from the last executed statement; either from the `try` or `catch` block (`examples/error_expression.ex`):

```
defmodule ErrorHandling do
  def execute(f) do
    try do
      f.()
    catch
      type, value -> %{type: type, value: value}
    end
  end
end
```

```
$ iex examples/error_expression.ex
```

```
> add_numbers = fn -> 3 + 5 end
> ErrorHandling.execute(add_numbers)
8

> divide_by_zero = fn -> 3 / 0 end
> ErrorHandling.execute(divide_by_zero)
%{type: :error, value: :badarith}

> ErrorHandling.execute(&rem/2)
%{type: :error, value: {:badarity, {&:erlang.rem/2, []}}}
```

12.3 Error Matching

Errors can be matched and dealt with at different layers (`examples/error_layers.ex`):

```elixir
defmodule ErrorLayers do
  def execute(f) do
    try do
      try do
        f.()
      catch
        :error, err -> {:failed, :with_error}
      end
    catch
      :throw, err -> {:failed, :with_throw}
    end
  end
end
```

```
$ iex examples/error_layers.ex

> throw_f = fn -> throw(:fail_with_throw) end
> ErrorLayers.execute(throw_f)
{:failed, :with_throw}

> error_f = fn -> 3 / 0 end
> ErrorLayers.execute(error_f)
{:failed, :with_error}
```

12.4 Cleanup Code

Code in the `after` block is always executed, regardless of whether an error occured (`examples/error_after.ex`):

```
defmodule ErrorAfter do
  def execute(f) do
    try do
      f.()
    catch
      type, value -> IO.puts("error: #{type} (#{value})")
    after
      IO.puts("done, cleaning up…")
    end
  end
end
```

```
$ iex examples/error_after.ex

> failing_func = fn -> 3 / 0 end
> ErrorAfter.execute(failing_func)
error: error (badarith)
done, cleaning up…

> working_func = fn -> IO.puts("working") end
> ErrorAfter.execute(working_func)
working
done, cleaning up…
```

Notice that `after` does *not* affect the return value of the `try`/`catch` special form!

12.5 Defining Exceptions

The `defexception` macro can be used to define new exceptions.

12.6 Errors in Concurrent Systems

Processes are isolated and don't share memory. If one process crashes, the other processes are not affected (`examples/proc_crash.exs`):

```
spawn(fn ->
  spawn(fn ->
    Process.sleep(1000)
    IO.puts("Process 2: finishing...")
  end)

  raise("Process 1: failing...")
end)
```

If the program is run, Process 1 will fail due to an exception. Process 2, which was spawned from Process 1, won't be affected and finishes successfully:

```
$ iex examples/proc_crash.exs
09:55:48.783 [error] Process #PID<0.109.0> raised an exception
** (RuntimeError) Process 1: failing...
    proc_crash.exs:7: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
Process 2: finishing...
```

Processes can be bidirectionally linked together. If one process fails, the linked process will also fail (`examples/proc_link.exs`):

```
spawn(fn ->
  spawn_link(fn ->
    Process.sleep(1000)
    IO.puts("Process 2: finishing...")
  end)

  raise("Process 1: failing...")
end)
```

Notice that `spawn_link/1` links the spawned process to the spawning process:

```
$ iex examples/proc_link.exs
09:57:07.740 [error] Process #PID<0.109.0> raised an exception
** (RuntimeError) Process 1: failing...
    proc_link.exs:7: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
```

A crashing process emits an exit signal to all its linked processes.

12.6.1 Trapping Exits

A linked process can react to an exit signal by setting up a *trap* using `Process.flag/2`. If a linked process fails, an according message can be received (`examples/proc_trap.exs`):

```
spawn(fn ->
  Process.flag(:trap_exit, true)

  spawn_link(fn ->
    raise("Something went wrong")
  end)

  receive do
```

```
    msg -> IO.inspect(msg)
  end
end)
```

An message of the form `{:EXIT, [from_pid], [exit_reason]}` is received and printed:

```
$ iex examples/proc_trap.exs
10:03:14.897 [error] Process #PID<0.110.0> raised an exception
** (RuntimeError) Something went wrong
    proc_trap.exs:5: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
{:EXIT, #PID<0.110.0>,
 {%RuntimeError{message: "Something went wrong"},
  [
    {:elixir_compiler_0, :"-__FILE__/1-fun-0-", 0,
     [file: 'proc_trap.exs', line: 5, error_info: %{module: Exception}]}
  ]}}
```

12.6.2 Monitors

Links are bidirectional. If the errors should only be propagated in one way, a *monitor* can be used instead (`examples/proc_monitor.exs`):

```
proc_finishing =
  spawn(fn ->
    Process.sleep(1000)
    IO.puts("process 1 finishing")
  end)

proc_failing =
  spawn(fn ->
    raise("process 2 failing")
  end)

Process.monitor(proc_finishing)
Process.monitor(proc_failing)

receive do
  msg -> IO.inspect(msg)
end

receive do
  msg -> IO.inspect(msg)
end
```

`Process.monitor/1` establishes an unidirectional link which allows the spawning process to monitor the other processes. The monitoring process receives a message if a monitored process finishes or fails:

```
$ elixir examples/proc_monitor.exs
10:30:31.372 [error] Process #PID<0.99.0> raised an exception
** (RuntimeError) process 2 failing
    proc_monitor.exs:9: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
{:DOWN, #Reference<0.2077770098.3528720387.219719>, :process, #PID<0.99.0>,
 {%RuntimeError{message: "process 2 failing"},
  [
    {:elixir_compiler_0, :"-__FILE__/1-fun-1-", 0,
     [file: 'proc_monitor.exs', line: 9, error_info: %{module: Exception}]}
  ]}}
process 1 finishing
{:DOWN, #Reference<0.2077770098.3528720387.219718>, :process, #PID<0.98.0>,
 :normal}
```

Notice that process 1 finishes with `:normal`, while process 2 throws a `RuntimeError`.

Notice that the monitoring process does not crash itself when a monitored process crashes.