Elixir is a modern functional language built on top of the Erlang VM. It's fully compatible with Erlang, but features a more standard syntax and many more features.

```
# Single line comments start with a number symbol.

# There's no multi-line comment,
# but you can stack multiple comments.

# To use the elixir shell use the `iex` command.
# Compile your modules with the `elixirc` command.

# Both should be in your path if you installed elixir correctly.

## ---------------------------
## -- Basic types
## ---------------------------

# There are numbers
3    # integer
0x1F # integer
3.0  # float

# Atoms, that are literals, a constant with name. They start with `:`.
:hello # atom

# Tuples that are stored contiguously in memory.
{1,2,3} # tuple

# We can access a tuple element with the `elem` function:
elem({1, 2, 3}, 0) #=> 1

# Lists that are implemented as linked lists.
[1,2,3] # list

# We can access the head and tail of a list as follows:
[head | tail] = [1,2,3]
head #=> 1
tail #=> [2,3]

# In elixir, just like in Erlang, the `=` denotes pattern matching and
# not an assignment.
#
# This means that the left-hand side (pattern) is matched against a
# right-hand side.
#
# This is how the above example of accessing the head and tail of a list works.

# A pattern match will error when the sides don't match, in this example
# the tuples have different sizes.
# {a, b, c} = {1, 2} #=> ** (MatchError) no match of right hand side value: {1,2}

# There are also binaries
<<1,2,3>> # binary
```

```
# Strings and char lists
"hello" # string
'hello' # char list

# Multi-line strings
"""
I'm a multi-line
string.
"""
#=> "I'm a multi-line\nstring.\n"

# Strings are all encoded in UTF-8:
"héllò" #=> "héllò"

# Strings are really just binaries, and char lists are just lists.
<<?a, ?b, ?c>> #=> "abc"
[?a, ?b, ?c]   #=> 'abc'

# `?a` in elixir returns the ASCII integer for the letter `a`
?a #=> 97

# To concatenate lists use `++`, for binaries use `<>`
[1,2,3] ++ [4,5]     #=> [1,2,3,4,5]
'hello ' ++ 'world'  #=> 'hello world'

<<1,2,3>> <> <<4,5>> #=> <<1,2,3,4,5>>
"hello " <> "world"  #=> "hello world"

# Ranges are represented as `start..end` (both inclusive)
1..10 #=> 1..10
lower..upper = 1..10 # Can use pattern matching on ranges as well
[lower, upper] #=> [1, 10]

## ---------------------------
## -- Operators
## ---------------------------

# Some math
1 + 1  #=> 2
10 - 5 #=> 5
5 * 2  #=> 10
10 / 2 #=> 5.0

# In elixir the operator `/` always returns a float.

# To do integer division use `div`
div(10, 2) #=> 5

# To get the division remainder use `rem`
rem(10, 3) #=> 1

# There are also boolean operators: `or`, `and` and `not`.
# These operators expect a boolean as their first argument.
```

```
true and true #=> true
false or true #=> true
# 1 and true    #=> ** (ArgumentError) argument error

# Elixir also provides `||`, `&&` and `!` which accept arguments of any type.
# All values except `false` and `nil` will evaluate to true.
1 || true  #=> 1
false && 1 #=> false
nil && 20  #=> nil
!true #=> false

# For comparisons we have: `==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` and `>`
1 == 1 #=> true
1 != 1 #=> false
1 < 2  #=> true

# `===` and `!==` are more strict when comparing integers and floats:
1 == 1.0  #=> true
1 === 1.0 #=> false

# We can also compare two different data types:
1 < :hello #=> true

# The overall sorting order is defined below:
# number < atom < reference < functions < port < pid < tuple < list < bit string

# To quote Joe Armstrong on this: "The actual order is not important,
# but that a total ordering is well defined is important."

## ---------------------------
## -- Control Flow
## ---------------------------

# `if` expression
if false do
  "This will never be seen"
else
  "This will"
end

# There's also `unless`
unless true do
  "This will never be seen"
else
  "This will"
end

# Remember pattern matching? Many control-flow structures in elixir rely on it.

# `case` allows us to compare a value against many patterns:
case {:one, :two} do
  {:four, :five} ->
    "This won't match"
  {:one, x} ->
```

```
    "This will match and bind `x` to `:two`"
  _ ->
    "This will match any value"
end

# It's common to bind the value to `_` if we don't need it.
# For example, if only the head of a list matters to us:
[head | _] = [1,2,3]
head #=> 1

# For better readability we can do the following:
[head | _tail] = [:a, :b, :c]
head #=> :a

# `cond` lets us check for many conditions at the same time.
# Use `cond` instead of nesting many `if` expressions.
cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "Me neither"
  1 + 2 == 3 ->
    "But I will"
end

# It is common to set the last condition equal to `true`, which will always match.
cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "Me neither"
  true ->
    "But I will (this is essentially an else)"
end

# `try/catch` is used to catch values that are thrown, it also supports an
# `after` clause that is invoked whether or not a value is caught.
try do
  throw(:hello)
catch
  message -> "Got #{message}."
after
  IO.puts("I'm the after clause.")
end
#=> I'm the after clause
# "Got :hello"

## ---------------------------
## -- Modules and Functions
## ---------------------------

# Anonymous functions (notice the dot)
square = fn(x) -> x * x end
square.(5) #=> 25
```

```
# They also accept many clauses and guards.
# Guards let you fine tune pattern matching,
# they are indicated by the `when` keyword:
f = fn
  x, y when x > 0 -> x + y
  x, y -> x * y
end

f.(1, 3)  #=> 4
f.(-1, 3) #=> -3

# Elixir also provides many built-in functions.
# These are available in the current scope.
is_number(10)    #=> true
is_list("hello") #=> false
elem({1,2,3}, 0) #=> 1

# You can group several functions into a module. Inside a module use `def`
# to define your functions.
defmodule Math do
  def sum(a, b) do
    a + b
  end

  def square(x) do
    x * x
  end
end

Math.sum(1, 2)  #=> 3
Math.square(3) #=> 9

# To compile our simple Math module save it as `math.ex` and use `elixirc`
# in your terminal: elixirc math.ex

# Inside a module we can define functions with `def` and private functions with `defp`.
# A function defined with `def` is available to be invoked from other modules,
# a private function can only be invoked locally.
defmodule PrivateMath do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

PrivateMath.sum(1, 2)    #=> 3
# PrivateMath.do_sum(1, 2) #=> ** (UndefinedFunctionError)

# Function declarations also support guards and multiple clauses:
defmodule Geometry do
```

```
  def area({:rectangle, w, h}) do
    w * h
  end

  def area({:circle, r}) when is_number(r) do
    3.14 * r * r
  end
end

Geometry.area({:rectangle, 2, 3}) #=> 6
Geometry.area({:circle, 3})       #=> 28.25999999999999801048
# Geometry.area({:circle, "not_a_number"})
#=> ** (FunctionClauseError) no function clause matching in Geometry.area/1

# Due to immutability, recursion is a big part of elixir
defmodule Recursion do
  def sum_list([head | tail], acc) do
    sum_list(tail, acc + head)
  end

  def sum_list([], acc) do
    acc
  end
end

Recursion.sum_list([1,2,3], 0) #=> 6

# Elixir modules support attributes, there are built-in attributes and you
# may also add custom ones.
defmodule MyMod do
  @moduledoc """
  This is a built-in attribute on a example module.
  """

  @my_data 100 # This is a custom attribute.
  IO.inspect(@my_data) #=> 100
end

## ---------------------------
## -- Structs and Exceptions
## ---------------------------

# Structs are extensions on top of maps that bring default values,
# compile-time guarantees and polymorphism into Elixir.
defmodule Person do
  defstruct name: nil, age: 0, height: 0
end

joe_info = %Person{ name: "Joe", age: 30, height: 180 }
#=> %Person{age: 30, height: 180, name: "Joe"}

# Access the value of name
joe_info.name #=> "Joe"
```

```
# Update the value of age
older_joe_info = %{ joe_info | age: 31 }
#=> %Person{age: 31, height: 180, name: "Joe"}

# The `try` block with the `rescue` keyword is used to handle exceptions
try do
  raise "some error"
rescue
  RuntimeError -> "rescued a runtime error"
  _error -> "this will rescue any error"
end
#=> "rescued a runtime error"

# All exceptions have a message
try do
  raise "some error"
rescue
  x in [RuntimeError] ->
    x.message
end
#=> "some error"

## ---------------------------
## -- Concurrency
## ---------------------------

# Elixir relies on the actor model for concurrency. All we need to write
# concurrent programs in elixir are three primitives: spawning processes,
# sending messages and receiving messages.

# To start a new process we use the `spawn` function, which takes a function
# as argument.
f = fn -> 2 * 2 end #=> #Function<erl_eval.20.80484245>
spawn(f) #=> #PID<0.40.0>

# `spawn` returns a pid (process identifier), you can use this pid to send
# messages to the process. To do message passing we use the `send` operator.
# For all of this to be useful we need to be able to receive messages. This is
# achieved with the `receive` mechanism:

# The `receive do` block is used to listen for messages and process
# them when they are received. A `receive do` block will only
# process one received message. In order to process multiple
# messages, a function with a `receive do` block must recursively
# call itself to get into the `receive do` block again.

defmodule Geometry do
  def area_loop do
    receive do
      {:rectangle, w, h} ->
        IO.puts("Area = #{w * h}")
        area_loop()
      {:circle, r} ->
        IO.puts("Area = #{3.14 * r * r}")
```

```
        area_loop()
    end
  end
end

# Compile the module and create a process that evaluates `area_loop` in the shell
pid = spawn(fn -> Geometry.area_loop() end) #=> #PID<0.40.0>
# Alternatively
pid = spawn(Geometry, :area_loop, [])

# Send a message to `pid` that will match a pattern in the receive statement
send pid, {:rectangle, 2, 3}
#=> Area = 6
#   {:rectangle,2,3}

send pid, {:circle, 2}
#=> Area = 12.56000000000000049738
#   {:circle,2}

# The shell is also a process, you can use `self` to get the current pid
self() #=> #PID<0.27.0>
```

References

- Getting started guide from elixir webpage
- Elixir Documentation
- "Programming Elixir" by Dave Thomas
- Elixir Cheat Sheet
- "Learn You Some Erlang for Great Good!" by Fred Hebert
- "Programming Erlang: Software for a Concurrent World" by Joe Armstrong