Elixir Functional language on BEAM

START HERE

`iex`
Interactive shell

`recompile`
Rebuild after edits

`iex -S mix`
Shell with your project

`value |> fun()`
Pipe into a call

`elixir <file.exs>`
Run a script

`%{a: x} = map`
Pattern match

`h <Module.fun>`
Read docs in IEx

`mix test`
Run the tests

Source .ex / .exs → `elixirc / mix` → **Bytecode** .beam files → `loads into` → **BEAM VM** lightweight processes → `message passing` → **Supervision** restart on crash

CONCEPTS

Everything is an expression and data is immutable: functions return new values, and = matches rather than assigns.

Files ending in .ex are compiled; .exs are scripts run as-is (tests, config, one-offs).

Mix (see the mix sheet) builds projects; Erlang tooling such as rebar3 covers Erlang dependencies.

RUN & COMPILE

`iex`
Start the interactive shell

`iex -S mix`
Shell with the project compiled and loaded

`elixir <file.exs>`
Run a script file

`elixir -e "<code>"`
Evaluate one expression

`elixir --version`
Show Elixir and OTP versions

`elixirc <file.ex> -o <dir>`
Compile files to .beam

`iex --sname <name>`
Shell as a named node

`iex --sname <name> --remsh <node>`
Attach to a running node
Both nodes need the same cookie; use --sname or --name on both.

`iex --erl "-kernel shell_history enabled"`
Persist shell history

IEX HELPERS

`h <Module.fun/arity>`
Show docs for a function

`i <term>`
Inspect type and details of a value

`v / v(<n>)`
Value of last / nth expression

`recompile`
Recompile the Mix project

`r <Module>`
Reload one module from source

`c "<file.ex>"`
Compile and load a file

`exports <Module>`
List a module's public functions

`clear`
Clear the screen

IEX HELPERS (cont.)

`flush`
Print and drop mailbox messages

`Ctrl+C Ctrl+C`
Quit the shell

`require IEx; IEx.pry()`
Pause here and inspect (needs iex -S mix)

`break! <Mod.fun/arity>`
Set a debugger breakpoint

`dbg(<expr>)`
Print expression with pipeline steps

DATA & PATTERN MATCHING

`{:ok, v} = <expr>`
Match a tuple, bind v

`[h | t] = <list>`
Split head and tail

`%{key: v} = <map>`
Pull a value out of a map

`^x = <expr>`
Pin: match against existing x

`%{m | key: v}`
Update an existing key

`<> and ++`
Concatenate binaries / lists

`"a #{x} b"`
String interpolation

`:atom / :"odd atom"`
Atom literals

`~w(a b c)a`
Word list sigil

`~r/<regex>/`
Regex sigil

`nil / true / false`
Only nil and false are falsy

`[a: 1, b: 2]`
Keyword list (atom keys, ordered)

`%MyStruct{field: v}`
Build or match a struct

CONTROL FLOW

`x |> f() |> g(a)`
Pipe result as first argument

`case <expr> do pattern -> ... end`
Branch on patterns

`if <c> do ... else ... end`
Plain conditional

`with {:ok, a} ← f(), {:ok, b} ← g(a) do ... end`
Chain matches, bail on first miss
An else block receives the first non-matching value.

CONTROL FLOW (cont.)

`for x ← <list>, <filter>, do: <expr>`
Comprehension

`for x ← l, into: %{}, do: {x, 1}`
Comprehension into a map

`try do ... rescue e → ... end`
Handle exceptions

`receive do msg → ... after <ms> → ... end`
Wait for a message

`when is_integer(x)`
Guard on a clause

`fn x → x + 1 end`
Anonymous function

`&(&1 + 1)`
Capture shorthand

`&Mod.fun/2`
Capture a named function

MODULES & FUNCTIONS

`defmodule <Name> do ... end`
Define a module

`def <name>(<args>) do ... end`
Public function

`defp <name>(<args>)`
Private function

`def f(0), do: 1`
One-line clause with pattern

`def f(x, y \\ 1)`
Default argument

`@doc "<text>"`
Document the next function

`@moduledoc "<text>"`
Document the module

`@spec f(integer) :: atom`
Declare a typespec

`alias A.B.C, as: D`
Shorten a module name

`import Mod, only: [f: 1]`
Import selected functions

`use Mod`
Run Mod's __using__ macro

`defstruct [:a, b: 0]`
Declare a struct

`@behaviour Mod / @impl true`
Implement a behaviour

STANDARD LIBRARY

`Enum.map(<en>, fn x → ... end)`
Transform each element

STANDARD LIBRARY (cont.)

`Enum.filter(<en>, &pred/1)`
Keep matching elements

`Enum.reduce(<en>, <acc>, fn x, acc → ... end)`
Fold into one value

`Enum.find(<en>, &pred/1)`
First match or nil

`Map.get(<map>, :k, <default>)`
Read with fallback

`Map.put(<map>, :k, v)`
Set a key

`String.split(<s>, ",")`
Split a string

`File.read!(<path>)`
Read file or raise

`IO.puts(<s>)`
Print a line

`IO.inspect(<v>, label: "<x>")`
Print any value, returns it

`Task.async(fn → ... end) |> Task.await()`
Run work concurrently

TESTING & DOCS

`mix test`
Run all tests

`use ExUnit.Case`
Make a module a test case

`test "<name>" do ... end`
Define a test

`assert <expr>`
Fail unless truthy

RECIPES

Script with args

```
# hello.exs
[name | _] = System.argv()
IO.puts("Hello #{name}")
# elixir hello.exs World
```

Debug a pipeline

```
data
|> Enum.map(&transform/1)
|> IO.inspect(label: "after map")
|> Enum.filter(&valid?/1)
|> dbg()
```

GenServer skeleton

```
defmodule Counter do
  use GenServer
  def start_link(n), do: GenServer.start_link(__MODULE__, n)
  def init(n), do: {:ok, n}
  def handle_call(:inc, _from, n), do: {:reply, n + 1, n + 1}
end
```

Remote shell into a node

```
iex --sname debug --cookie secret --remsh app@host
# then inspect live state:
Process.list() |> length()
```