Elixir Functional language on BEAM

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

`iex --sname <name> --remsh <node>`
Attach to a running node

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

`Ctrl+C Ctrl+C`
Quit the shell

`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

`%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

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

`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

MODULES & FUNCTIONS (cont.)

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

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

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

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

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

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

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

STANDARD LIBRARY

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

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

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

STANDARD LIBRARY (cont.)

`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