Elixir Cheatsheet

Tooling & Project Basics Mix commands

- New project: `mix new my_app`
- Run: `iex -S mix  mix run`
- Compile: `mix compile`
- Tests: `mix test`
- Format: `mix format`
- Deps: `mix deps.get`, `mix deps.update`

Project layout

- `lib/` application code
- `test/` tests
- `mix.exs` project config

Basics: Modules, Values, Types Hello

```
IO.puts("Hello, Elixir")
```

Modules `defmodule MyMod do  def hello, do:  "hi" end`

Bindings

- Immutable: `x = 1`
- Rebinding allowed (new value): `x = x + 1`
- Pattern match: `{a, b} = {1, 2}`

Common types

- Numbers: `1`, `1.0`
- Atoms: `:ok`, `:error`
- Strings (UTF-8): `"hi"`
- Lists: `[1, 2, 3]`
- Tuples: `{1, 2}`
- Maps: `%a:  1`

Pattern Matching (Core Feature) Assignment is match

- `{:ok, v} = f()`
- Pin operator: `x̂ = expr`

Function heads `def f({:ok, v}), do:  v  def f({:error, _}), do:  nil`

Case `case x do  {:ok, v} -> v  _ -> :error end`

Control Flow If / unless

- `if cond, do:  a, else:  b`
- `unless cond, do:  a`

Cond `cond do  x > 0 -> :pos  x < 0 -> :neg  true -> :zero end`

With (happy path) `with {:ok, a} <- fa(),  {:ok, b} <- fb(a) do  {:ok, a + b} end`

Functions & Pipelines Functions

- `def add(a, b), do:  a + b`
- Private: `defp hidden()`

Anonymous

- `fn x -> x + 1 end`
- Capture: $\&1 + 1$

Pipeline operator

- `x |> f() |> g()`
- First arg threaded

Collections & Enum List ops

- Prepend: `[0 | xs]`
- Concat: `xs ++ ys`

Enum (eager)

- `Enum.map(xs, &1 * 2)`
- `Enum.filter(xs, &1 > 0)`
- `Enum.reduce(xs, 0, &+/2)`

Stream (lazy)

- `Stream.map(xs, ...)`
- Realize with `Enum.to_list`

Maps, Structs Maps

- Access: `m[:a]`, `m.a` (atoms only)
- Update: `%m | a:  2`
- Put: `Map.put(m, :b, 3)`

Structs `defmodule User do  defstruct [:id, :name] end u = %User{id:  1, name:  "A"}`

Option-like Patterns Atoms + tuples

- `:ok`, `:error`
- `{:ok, v}`, `{:error, reason}`

Nil

- Absence of value
- Only `false` and `nil` are falsey

Concurrency (BEAM Model) Processes

- Lightweight, isolated
- Communicate via messages

Spawn & send `pid = spawn(fn -> loop() end) send(pid, :ping)`

Receive `receive do  :ping -> IO.puts("pong") after 1000 -> :timeout end`

OTP Basics GenServer

- Standard behavior for stateful processes
- Call vs cast vs info

Skeleton `defmodule MySrv do use GenServer def start_link(s), do: GenServer.start_link(__MODULE__, s) def init(s), do: {:ok, s} end`

Supervision

- Let it crash
- Restart strategies: `:one_for_one`, etc.

Protocols & Polymorphism **Protocol**

`defprotocol Size do def size(x) end`

Implementation `defimpl Size, for: List do def size(xs), do: length(xs) end`

Error Handling **Errors vs exceptions**

- Prefer `{:ok, v} | {:error, r}`
- Exceptions for truly exceptional cases

Try/rescue `try do risky() rescue e in RuntimeError -> e end`

Testing **ExUnit**

- `mix test`
- `use ExUnit.Case`
- Assertions: `assert`, `refute`

Example `test "add" do assert add(1, 1) == 2 end`

Common Idioms

- Happy-path with `with`
- Pattern match on return values
- Keep processes small and supervised
- Push side effects to boundaries

Mini Reference

- `|>` pipeline
- `=` is match, not assignment
- Immutability everywhere
- Prefer messages over shared state