Gleam Cheatsheet

Tooling & Project Basics Commands

- New project: `gleam new my_app`
- Run: `gleam run`
- Test: `gleam test`
- Format: `gleam format`
- Build: `gleam build`

Targets

- Erlang/BEAM target (default) and JavaScript target
- Dependencies via Hex packages (`gleam.toml`)

Project layout

- `src/` Gleam code
- `test/` tests
- `gleam.toml` project config

Basics: Modules, Values, Types Module + main `import gleam/io pub fn main() { io.println("Hello, Gleam!") }`

Bindings

- Immutable: `let x = 1`
- Shadowing is allowed: `let x = x + 1`

Common types (conceptual)

- `Int`, `Float`, `Bool`, `String`
- `List(a)`, `Option(a)`, `Result(a, e)`
- Tuples: `(a, b)`

Functions Signatures

- `fn add(a: Int, b: Int) -> Int { a + b }`
- Public: `pub fn ...`

Anonymous functions

- `fn(x) { x + 1 }`

Pipelines (very common)

- `x |> f |> g`
- With args: `x |> map(with: fn(a) { ... })`

Algebraic Data Types & Pattern Matching

Custom types `pub type Color { Red Green Blue }`

Parameterized types `pub type Box(a) { Box(a) }`

Pattern match `pub fn to_rgb(c: Color) -> (Int, Int, Int) { case c { Red -> (255, 0, 0) Green -> (0, 255, 0) Blue -> (0, 0, 255) } }`

Guards (when available)

- `case x { n if n > 10 -> ...; _ -> ... }`

Option & Result (Everyday Error Handling)

Option

- `pub type Option(a) { Some(a) None }` (conceptually)
- Match:

```
case maybe { Some(x) -> x None -> 0 }
```

Result

- `Result(a, e)`: `Ok(a)` or `Error(e)`
- Propagate-ish via helpers + pipeline

Typical pattern (pipeline)

- `result |> result.map(fn(x) {...})`
- `result |> result.then(fn(x) { ... return Result ... })`

Lists & Common Data Work List basics

- Literal: `[1, 2, 3]`
- Prepend: `[0, ..xs]` (spread syntax)
- Concatenate: `xs ++ ys`

Map/filter/fold

- Usually from `gleam/list`:
- `list.map(xs, fn(x) {...})`
- `list.filter(xs, fn(x) {...})`
- `list.fold(xs, init, fn(acc, x) {...})`

Common pipeline style `xs |> list.filter(fn(x) { x > 0 }) |> list.map(fn(x) { x * 2 })`

Strings String notes

- Strings are UTF-8
- Common ops via `gleam/string`

Examples

- `string.length("hi")`
- `string.contains("hello", "ell")`
- `string.split("a,b", ",")`

Records, Tuples, Destructuring Tuples

- Make: `let pair = (1, "a")`
- Destructure: `let (a, b) = pair`

Records (custom types)

- ADT constructors often act like records:

```
pub type User { User(id: Int, name: String) }
let u = User(id: 1, name: "A")
```

Update (conceptual)

- Pattern match + rebuild (immutable)

Modules, Imports, Visibility Imports

- `import gleam/io`
- `import gleam/list`
- Alias: `import gleam/string as str`

Public API

- `pub` exposes functions/types
- Keep constructors private to enforce invariants (API design pattern)

Interop & Targets (BEAM / JS) External functions

- Use `@external` to bind to Erlang/JS implementations
- Keeps Gleam code typed while using platform libs

BEAM notes

- Interop with Erlang/Elixir ecosystems
- Concurrency via BEAM processes (typically through libraries)

JS notes

- Good for frontend/tooling; interop with JS via externals

Testing & Formatting Tests

- `gleam test`
- Tests live under `test/`
- Assertions via test libraries (`gleeunit` is common)

Style

- `gleam format` (use it; makes diffs sane)

Common Idioms (Quick Recipes) Parse / validate with Result

- Keep functions total: return `Result` instead of throwing
- Build pipelines of `result.then` steps

Keep IO at the edge

- Pure functions for logic, `main` for wiring/printing

Prefer ADTs over booleans

- Replace `Bool` flags with a `type Mode { ... }`

Make illegal states unrepresentable

- Use private constructors + smart constructors returning `Result`

Mini Reference

- `let` for binding; immutable by default
- `pub` to export; `import` to use modules
- `case` for pattern matching
- `|>` pipeline for readable transforms
- Prefer `Option`/`Result` over nulls/exceptions