



ClojureDart

Cheatsheet for Clojurists

CLI

```
clj -M:cljd init
clj -M:cljd flutter # press RET to restart
clj -M:cljd compile
clj -M:cljd clean # when in panic
clj -M:cljd upgrade # stay current w/ CLJD
```

Using Dart packages

In the `ns` form, `:require` as usual but with a **string** instead of a symbol:

```
(ns my.project
  (:require
    ["package:flutter/material.dart" :as m]))
```

Types and aliases

Unlike Clojure/JVM, **types can be prefixed** by an alias. `m/ElevatedButton` refers to `ElevatedButton` in the package aliased by `m`.

Types and nullability

In ClojureDart, `^String x` implies `x` | can't be `nil`. Use `^String? x` to allow for `nil`.

Parametrized types

Unlike Clojure/JVM, generics do exist at runtime so ClojureDart has to deal with them.

Dart	ClojureDart
<code>List</code>	<code>List</code>
<code>List<Map></code>	<code>#/(List Map)</code>
<code>List?<Map></code>	<code>#/(List? Map)</code>
<code>List<Map?></code>	<code>#/(List Map?)</code>

`#/(List Map)` is just the `List` symbol with some metadata! Thus you can write `^#/(List Map) x`.

Property access

Instance `get (. -prop obj)`, `set (. -prop! obj x)` or `(set! (. -prop obj) x)`

Static `m/Colors.purple`, this can even be chained `m/Colors.purple.shade900` or even terminated with a method call `(m/Colors.purple.shade900.withAlpha 128)`.

:flds, Object destructuring

`:keys`, `:strs` and `:syms` are proud to introduce `:flds` their object-destructuring counterpart.

`(let [{:flds [height width]} size] ...)` is equivalent to:

```
(let [height (. -height size)
      width (. -width size)]
  ...)
```

Constructors

In Dart, constructors do not always allocate, they can return existing instances. that's why there's **no new nor trailing .** (dot) in ClojureDart.

Default constructors are called with classes in function position.

```
(StringBuffer "hello")
```

Named constructors are called like static methods.

```
(List/empty .growable true)
```

Understanding Dart signatures

```
writeAll(Iterable objects,
  [String separator = ""])
```

One (positional) parameter `objects` of type `Iterable`, one optional positional parameter `separator` of type `String` whose default value is `" "`.

```
RegExp(String source,
  {bool multiLine = false,
   bool caseSensitive = true})
```

One (positional) parameter `source` of type `String`, two optional named `bool` parameters `multiLine` (defaults to `false`) and `caseSensitive` (defaults to `true`).

```
any(bool test(E element))
```

One (positional) parameter: `test`, a **function** of one argument `element` of type `E` (a generics type argument) returning a `bool`.

Named arguments

Some Dart functions and methods expect named arguments (`argname: 42` in Dart), in ClojureDart it's `.argname 42`.

```
(m/Text "Hello world"
  .maxLines 2
  .softWrap true
  .overflow m/TextOverflow.fade)
```

Optional params (named or not)

Sometimes interop requires implementing a function or method taking Dart optional parameters.

`[a b c .d .e]` 3 positional parameters and two named ones: `d` and `e`.

`[a b c ... d e]` 5 positional parameters, 3 fixed and 2 optional.

`[.a 42 .e]` two named parameters `a` and `b` where `a` defaults to 42.

`[... a 42 b]` 2 optional positional parameters `a` and `b` where `a` defaults to 42.

Enums

Enums values are just static properties on types. e.g. `m/TextAlign.left`

Consts and const opt-out

Dart has **consts**: deduplicated compile-time constant expressions in which `const` constructors may participate. It's an opt-in mechanism.

ClojureDart maximally infers const expressions. In some rare occasion (like creating a sentinel or a token) you have to **tag the expression as unique**: `^:unique (Object)`; otherwise you always get the same instance.

Creating classes

`reify`, `deftype` and `defrecord` gets new powers.

Creating classes: :extends

Extending classes, even abstract ones!

The `:extends` option specifies a class or a constructor call (the `super` constructor call).

Creating classes: mixins

Mixin types must be tagged with `^:mixin`.

Creating classes: operators

Dart supports operators overloading but not all operators make valid Clojure symbol (`[]` for example). That's why **it's valid ClojureDart to have strings as methods names**.

```
(. list "[]" i 42) ; list[i] = 42
```

It's also valid to use strings-as-names when implementing operator overloads as part of a class definition.

Getters/Setters

Dart properties are not all fields: most are getters/setters. However they behave like fields, you get them (`[-prop obj]`) and you set them (`[set! (-prop obj) 42]` or `[-prop! obj 42]`).

Getters and setters are defined as regular methods, in the body of a `reify/deftype/defrecord`. A getter expects one parameters [`this`] while a setter expects two [`this v`].

If the property is defined in a parent class or interface, you don't need anything special. **If the property is newly introduced by the type you need to tag the method name with `^:getter` or `^:setter`.**



cljd.flutter: more Flutter, less clutter!

`cljd.flutter` is not a framework. It's an utility lib to cut down on the boilerplate and thus to make Flutter more pleasing to Clojurists palates. Bon Appétit!

It's dangerous to go alone!

Require this.

```
[cljd.flutter :as f]
["package:flutter/material.dart" :as m]
```

f/run [& widget-body]

Called from `main`, starts the application (a `Widget`). Its body is interpreted as per `f/widget`. Makes the root of the app **reload-friendly**.

f/widget [& widget-body]

Mother of all macros. Evaluates to a `Widget`.

Its body is made of **interleaved expressions and directives**.

Directives are always **keywords followed by one other form**. Directives range from the mundane `:let` to the specific `:vsync`.

Expressions are **.child-threaded**.

.child-threading

Inside a “widget-body”, expressions are threaded through the named param `.child`:

```
(f/widget
 m/Center
 (m/Text "hello"))
```

is equivalent to:

```
(m/Center .child (m/Text "hello"))
```

When the two expressions are separated by a dotted symbol, this symbol is used to thread them:

```
m/MaterialApp
 .home
 m/Scaffold
 .body
 (m/Text "Don't stop it now!")
```

:let directive

`(f/widget :let [some bindings] ...)` rewrites as `(let [some bindings] (f/widget ...))`.

:key directive

Keys are primordial for recognizing sibling widgets in a list (or anywhere a `.children` argument is expected) updates after updates without losing or mixing state. It will wrap its value inside a `ValueKey` so that you don't have to.

:watch directive

Or how to react to IO and state.

`:watch` takes a bindings vector. Unlike `:let`, expressions must be **watchable** and binding forms will be successively bound to values produced by their watchable.

```
:watch [v an-atom]
(m/Text (str v))
```

When `an-atom` changes, everything after `:watch` will update with `v` bound to the current value of `an-atom`.

watchables

`nil`, `atoms`, `cells`, `Streams`, `Futures`, `Listenables` and `ValueListenables` and any extension of the `Subscribable` protocol.

Cells

*Great for maintaining and reusing derived state; **tip**: try to share them via inherited bindings (see `:bind`) rather than function arguments.*

`f/$` (“cache”) creates a cell.

`(f/$ expr)`, like a **spreadsheet cell**, updates its value each time a dependency changes.

Dependencies can be any **watchable** including other cells. Dependencies are read using `f/<!` (“take”). `f/<!` can be used in any function called directly or **indirectly** from a cell.

:managed directive

*Automatic lifecycle management for `*Controllers` and the like.*

`:managed` takes a bindings vector like `:let` but expressions must produce objects in needs of being **disposed** (using the `.dispose` method by default).

:bind directive

*Dynamic binding but along the widgets tree, not the call tree. **Inherited** bindings in Flutter speak.*

`:bind { :k v }` establishes a inherited binding from `:k` to `v` visible from all descendants.

:get directive

`:get [:k1 :k2]` retrieves values bound to `:k1` and `:k2` via `:bind`.

`:get [m/Navigator]` retrieves instance returned by `(m/Navigator.of context)`

:context directive

For when Flutter asks for more context.

`:context ctx` binds `ctx` to a `BuildContext` instance.

:vsync directive

Have you ever chased an electron beam across a phosphor screen?

`:vsync clock` binds `clock` to a `TickerProvider`, generally required by animations.