Observable Cheatsheet

1 Standard Library

Dom [observable]

canvas context2d download element
input range select svg
text uid

Files [observable]

buffer text url

FileAttachments [attachment]

`achmt = FileAttachment("file")` # construct

url text json csv
tsv image arrayBuffer stream
blob

Promises [observable]

delay tick when

Generators [observable]

disposable filter input
map observe queue
range valueAt worker

Require [require]

`req = require("d3-array")` # from JSDelivr

resolve alias

Literals [observable]

The following are top-level objects. See sections on markdown and literals.

md html tex dot

Reactive Variables [observable]

These are top level objects that observe and react to changes on the notebook and server.

invalidation now width

2 Data

Inline

For small datasets:

```
primes = [2,3,5,7,11]                # manual entry
csv = d3.csvParse(`rawPaste`)        # copy-paste
```

Attachments

Attach a file in the UI and then:

```
a = FileAttachment("file") # attachment object
d = await a.json();        # promised data
t = d3.hierarchy(d)        # in-mem tree
```

In lieu of d3, `FileAttachment` *has its own csv, json, and text parsers that implicitly* **await***:*

```
c = FileAttachment("x.csv").csv()
```

Alternatively, upload directly from local f.s.:

```
viewof file = html`<input type=file>`
html`<img src="${URL.createObjectURL(file)}">`
```

Databases

Observable is meant to share notebooks and data. It is also great for prototyping. Using the runtime, you may even use it to construct full web-apps. However, notebooks themselves are not meant to serve full applications. Hence, UI-created db connections are only allowed for private notebooks.

```
npm install -g @observablehq/database-proxy
...                                  # run locally
```

Secrets

Use the UI to create a secret (a la Github). Then:

```
Secret(`MY_KEY`)  # expose it, or embed:
dat = d3Fetch.json(url + {`MY_KEY`}
```

Files

Use this API to retrieve from local filesystem.

```
viewof myText = html`<input type=file>`
Files.text(myText)          # invoke the API
```

3 Inputs

General Use

```
import {Checkbox} from ...          # from here
data = [...]; options = {...};      # setup
viewof cb = Checkbox(data, options)# create
html`checked:                       # reference
    ${cb.map(c => html`${c}`)}`
```

Most canonical inputs have the 2 arguments above, where `data` *is an array, and* `options` *is an object-literal, as described below.*

Options

Most inputs allow arbitrarily-formatted `data` *but then require using non-trivial* `keysof`, `valuesof` *closures for data extraction. Other options are mostly cosmetic.*

	Button	Chckbx	Toggle	Radio	Range	Search	Text	Textar.	Select	Table
Label	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Value	✓	✓	✓		✓		✓	✓	✓	✓
Width	✓				✓	✓	✓	✓	✓	✓
Disabl'd	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Sort		✓		✓					✓	✓
Unique		✓		✓					✓	
Locale		✓		✓		✓			✓	
Format		✓		✓	✓	✓			✓	✓
Keyof		✓		✓					✓	
Valueof		✓		✓					✓	
Plachldr					✓	✓	✓	✓		
Columns						✓				✓
Sp'lch'k						✓	✓	✓		
Requ'd						✓	✓	✓		✓
Rows								✓		✓
Datalst						✓	✓			
Multip.									✓	✓
Readо'ly							✓	✓		
Minlen.							✓	✓		
Maxlen.							✓	✓		
Validate							✓	✓		
Submit							✓	✓		

The following are exclusive to one input type:

- button required, reduce
- toggle values
- range step, transform, invert
- select size
- text type, pattern
- search query, filter
- table reverse, align, maxwidth, height, maxHeight, layout

Idiosyncratic / User-defined Inputs

- 2D Slider
- Binary Input
- Ternary Slider
- Fine-range Slider
- Form Input
- Range Slider
- Scrubber
- Player
- Inputs in a Grid
- Copier
- US Counties
- Periodic Table

4 Imports & Exports

Cell Imports

Successively: static import; multiple cells; with dependency injection; aliasing; non-public.

```
import {chart} from '@mbostock/phases-moon'
import {chart, viewof year} from ...
import {chart} with {mydata as data} from ...
import {chart as barchar} ...
```

Note: Imported cells evaluate dependencies without binding them to the local notebook. Bindings are live (Δ with underlying dependencies), but lazily evaluated. Since you can only import ***named*** *cells, consider "forking" a notebook instead.*

Require

Used for plain JavaScript library imports.

```
mmnt = require("moment")      # moment lib
mmnt = require("moment@2")# vers. 2
ab = require(ä", "b")         # multiple → 1
```

Libs fetch from npm by default. Override like:

```
d3 = require("my.server/d3.js")     # www loc
d3 = require("localhost/d3.js")     # local
```

As an alternative, consider ES6-native imports:

```
d3_esm = import("https://some.cdn/d3@6")
```

Sharing

Notebooks default to "private" access but can also be "shared" (exposing an obfuscated url) or "published," all as available in the UI. Alternatively, click to left of a cell to download its contents to png or json, etc.

Embedding

To the left of a cell, select "embed" to generate copy-pasteable html `IFrame` *code (which will, in turn, include the observable runtime). Here is a tool that generates an IFrame for multiple cells. Alternatively, download a zip archive of project contents by clicking "download code." The generated archive is described here. Even better, just install it via npm using the url in the download code menu option:*

```
unix-shell$ @observablehq/runtime
unix-shell$ npm install "<url>"
```

Observable Runtime

A `Runtime` *instance can spawn a* `module`*, which, when passed an* `observer`*, spawns a* `variable`*, which is a piece of state in a reactive program. The standard observer is* `Inspector`*, which renders the current value of the given variable to its associated DOM element.*

```
<html><body><script type="module">
import {Runtime, Inspector} from ...
import define from "https://...";
const runtime = new Runtime();
const main = runtime.module(define,
      name => { if (name === "hello") {
          return new Inspector(document.body);
      }
});
</script>
```

5 Markdown

Headers

```
#  Title          # H1 header
##  Title         # H2 header
###  Title        # H3 header
```

Emphasis

```
**text**          # bold text
_text_            # italicized text
~~text~~          # striken-thru ~~text~~
> text            # quoted text
```

Hyperlinks, Images, & Videos

```
[http://google.com](click here!)
![http://clip.art](pic's subtitle)
```

Embed youtube video:

```
[![description]
(http://img.youtube.com/vi/myvidid/0.jpg)]
(http://www.youtube.com/watch?v=myvidid)
```

Lists

Unordered

```
* item1
* item2
* item3
```

Ordered

```
1. item1
1. item2
1. item3
```

Mixed, Nested lists:

```
1. First major item
    * 1st sub-item
    * 2nd sub-item          # whitespace matters!
1. Second major item
```

Code

```
normal `code()`text      # inline
```                      # start ...
myFunction(){}   # code block
```                      # ... end
```

In observable, backticks must be escaped (see below), so really, things look more like:

template literal
```
\`\`\`\`
myFunction(){}
\`\`\`\`
```

tagged TL
```
\`\`\`\`html
<i>ital</i>
\`\`\`\`
```

Escaping

The following characters must be escaped: `*`*, and* `>`*. Within observable, additionally escape* `` ` ``.

Tables

For all but the simplest tables, embed html, but for simple tables, use raw markdown syntax:

```
| head1 | head2 | head3 |     # headers
|-------|-------|-------|     # horizontal rule
| val | val | val |           # rows
```

Checklists

```
- [ ] Mercury      # unchecked
- [x] Venus        # checked
- [x] Earth        # note: '-' for list
- [ ] Mars         # space ↔ empty
```

HTML

Most html can be embedded successfully. Some idioms are below, the first of which is used to add dynamic html text highlighting:

```
<span ${spanStyle('link')}>*Link*</span>
<span>&# 10004<\span>          # unicode char

<dl>                           # begin definition list
<dt></dt>                      # definition header
<dd></dd>                      # definition
</dl>                          # end

<figcaption>text</figcaption>  # nice captn
<figure style='max-width:50px'># nice image
<img src='https://pic.location'>#
</figure>                      # end
```

6 Literals

No matter the language below, observable always requires "tagging" the "template literal," by prefixing the target language prior to the template-demarcating graves (`` ` ``*).*

Markdown

(See separate section on md.)

```
md`# Page Title`
```

HTML

```
html`<p>I am a <i>par</i> element!</p>`
html`<style>                   # this css
  .highlight {background: yellow;} # applies
</style>`                      # globally
```

SVG

```
svg`<svg width=60 height=60>
    ${svg.fragment`
      <circle cx=30 cy=30 r=30></circle>`
</svg>`
```

GraphViz / DOT

```
dot`digraph { a -> b; }`
```

TEX

```
tex`E = mc^2`          # inline
tex.block`...`         # block-mode
```

Lorem Ipsem

Really just a cell import from here, then:

```
md`${loremipsum({using: "Bro Ipsum"})}`
```

Template Expansion

All literals are actually templates that are parsed into intermediate languages (like TEX and markdown) and ultimately resolved to html. Prior to parsing, observable expands "interpolated expressions" (those inside `${...}`*) into their "substituted" string value.*

```
`<h1>Hello ${value}</h1>`
```

Nesting Literals

```
md`Some tex: ${tex`\tau`}.`   # TEX in md
```

Deep nesting: given observable / js variable:

```
array = [tex`A=1`, tex`A=2`]
```

... nest TEX inside js inside html inside js inside html inside md!

```
md`A list of elements:
<ul>
  ${clone(array).map(a => html`<li>${a}`)}
</ul>`
```

7 Keyboard Shortcuts

Cell Editing

```
⇧-↵  ^-S   # run current cell
⌥-⇧-↓      # focus next cell
⌥-⇧-↑      # focus prev. cell
⌥-↵        # split cell @ cursor
⌥-⇧-↵      # ibid, focusing 1st
⌥-↵        # insert cell, after
⌥-⇧-↵      # insert cell, before
⌥-⇧-↑      # move cell up
⌥-⇧-↓      # move cell down
⌥-⌫        # merge cell w/ prev.
^-⇧-D      # merge cell w/ next
^-⇧-P      # [un]pin cell
^-J        # jump to ref'd cell
⎋          # blur current cell
```

Cursor

```
^-←        # previous word
^-→        # next word
⌥-←        # start line
⌥-→        # end line
^-HOME     # to start of cell
^-END      # to end of cell
^-CLICK    # multiple carets
```

Selection

```
⇧-←        # extend sel. left
⇧-→        # extend sel. right
⇧-↑        # extend sel. up
⇧-↓        # extend sel. down
^-⇧-←      # extend sel. word
^⇧-→       # extend sel. word
⌥-⇧-←      # to start of line
⌥-⇧-→      # to end of line
```

Text

```
^-BACKSPACE  # delete word before
^-⌫          # delete word after
⇧-⇥          # auto-indent
^-]          # indent once
^-[          # de-indent once
^-/          # toggle comment (JS-only)
```

Cell Navigation

```
k            # select cell above
j            # select cell below
⇧-↵          # run selected cells
⇧-k          # expand selection up
⇧-j          # expand selection down
⇧-↑          # move cells up
⇧-↓          # move cells down
o            # insert cell after
⇧-o          # insert cell before
x            # toggle select cell
⇧-a          # select all cells
⎋            # deselect all cells
↵            # open cell editor
d            # delete selected cells
p            # [un]pin selected cells
```

Other

```
^-c          # copy selection / line
^-x          # cut selection / line
^-v          # paste selection / line
^-z          # undo
^-y          # redo
^-f          # search text
^-g          # find next occurrence
^-⇧-g        # previous occurrence
```

8 Compiler

Grammar

```
Cell:
    ImportCell
    NamedCell
    Block
    Expression
ImportCell:
    import NamedImports
           from moduleSpecifier
    import NamedImports
           with namedImports
           from moduleSpecifier
NamedCell:
    Identifier = Block
    Identifier = Expression
    FunctionExpression
    ClassExpression
Identifier:
    IdentifierName
    viewof IdentifierName
    mutable IdentifierName
```

Parser

See examples here, or do the following to see how parser creates a json tree or your choosing:

```
import {parseCell} from "@observablehq/parser";
const cell = parseCell(`hello = "world"`);
```

9 Misc. Techniques

The below are cool techniques that blend much of the above in a unique, non-categorizable way:

- Gallery of gifs
- CLI-based notebook rendering
- GIF animations from Canvas using gif.js