Navigating ClojureScript's fire swamps

@spinningtopsofdoom / @bendyworks

As of ClojureScript 1.9.494

ClojureScript tooling seems Inconceivable

Sub second incremental transpilation with hot reloading

Sub second generation of Fully accurate source maps

Advanced Dead Code elimination and Cross Module Code Motion

Then you encounter your first ro.us

and spend hours hunting for a bug in externs

Let's start our journey through the fire swamps of ClojureScript

Google Closure Library
JavaScript Standard Library

Google Closure Library

- Equivalent to Ruby or Python Standard Libraries
- Packaged with ClojureScript
- Written for Google Closure Compiler

It's a great standard library

Except for the documentation

Source Code is very readable

Great buried gems

- AJAX
  - classic goog.net.XhrIo
  - New fetch API goog.net.FetchXmlHttp
- goog.async Namespace
  - Debouncer, Throttle, Delay
- goog.date Namespace
  - DateTime, Interval, Date, DateRange

goog.object Namespace

- Safe and robust interaction with JavaScript Objects
- Use goog.object/get and goog.object/set instead of aget and aset

goog.define

Very well hidden

Parameterize builds

Call with goog-define macro

```
(ns my.api)

(goog-define TIMEOUT 300)

(defn load-settings []
  (ajax-call {:timeout TIMEOUT}))
```

Override with closure-defines Complier Setting

```
:clojure-defines {'my.api.TIMEOUT 5000}
```

Feature Flags

Unused features removed via Dead Code Elimination

Boolean Values
^boolean type hint

```
(ns my.setting)

(goog-define ADMIN false)

(def permissions
  (if ^boolean ADMIN
    {:access :all}
    {:access :user}))
```

String Values
if or cond conditional expressions
identical? for comparison

```
(ns my.setting)

(goog-define USER "normal")

(def permissions
  (if (identical? USER "admin")
    {:access :all}
    {:access :user}))

(def oversees
  (cond
    (identical? USER "admin") #{"supervisors", "users"}
    (identical? USER "supervisor") #{"users"}:else
    :else #{}))
```

Externs

Third Party JavaScript Libraries

Not Handled by Google Closure Compiler

JavaScript Library

```
var foo = {};
foo.bar = function(greeting) {
  return greeting + " friend";
}
```

Calling library in ClojureScript

```
(.bar js/foo "hello")
```

Everything works fine in

- Development
- Testing
- QA
- Production Builds

Error in production application

Complied JavaScript

```
foo.w("hello");
```

What happened to foo.bar?

Google Closure Compiler Renaming Advanced Optimizations

```
foo.bar("hello");
```

Google Closure Compiler

```
foo.w("hello");
```

Inform Google Closure about external names

An "externs" file

```
var foo = {};
foo.bar = function(greeting) {};
```

Advanced Optimizations (with externs)

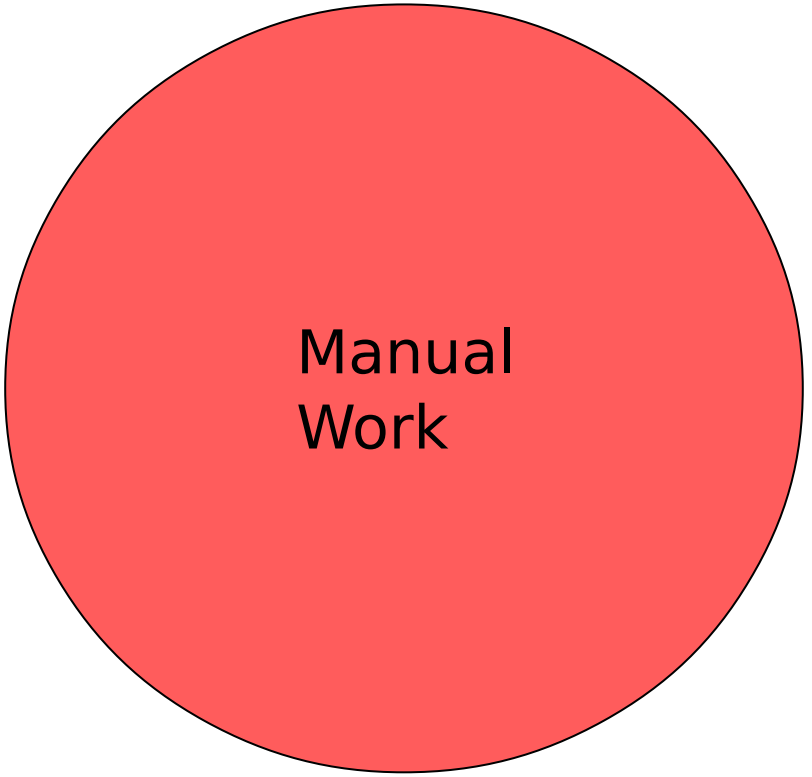
```
foo.bar("hello");
```

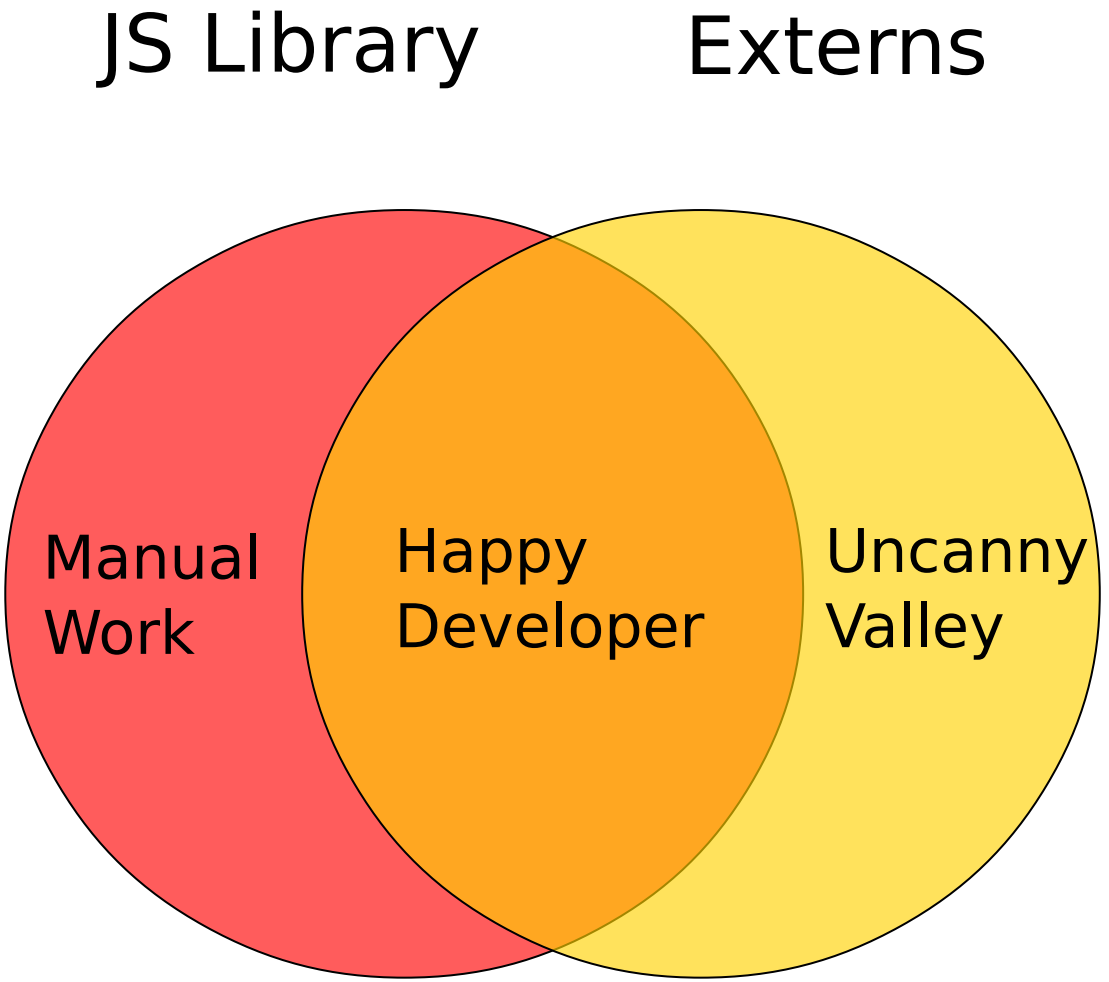
Google Closure Compiler  foo.bar

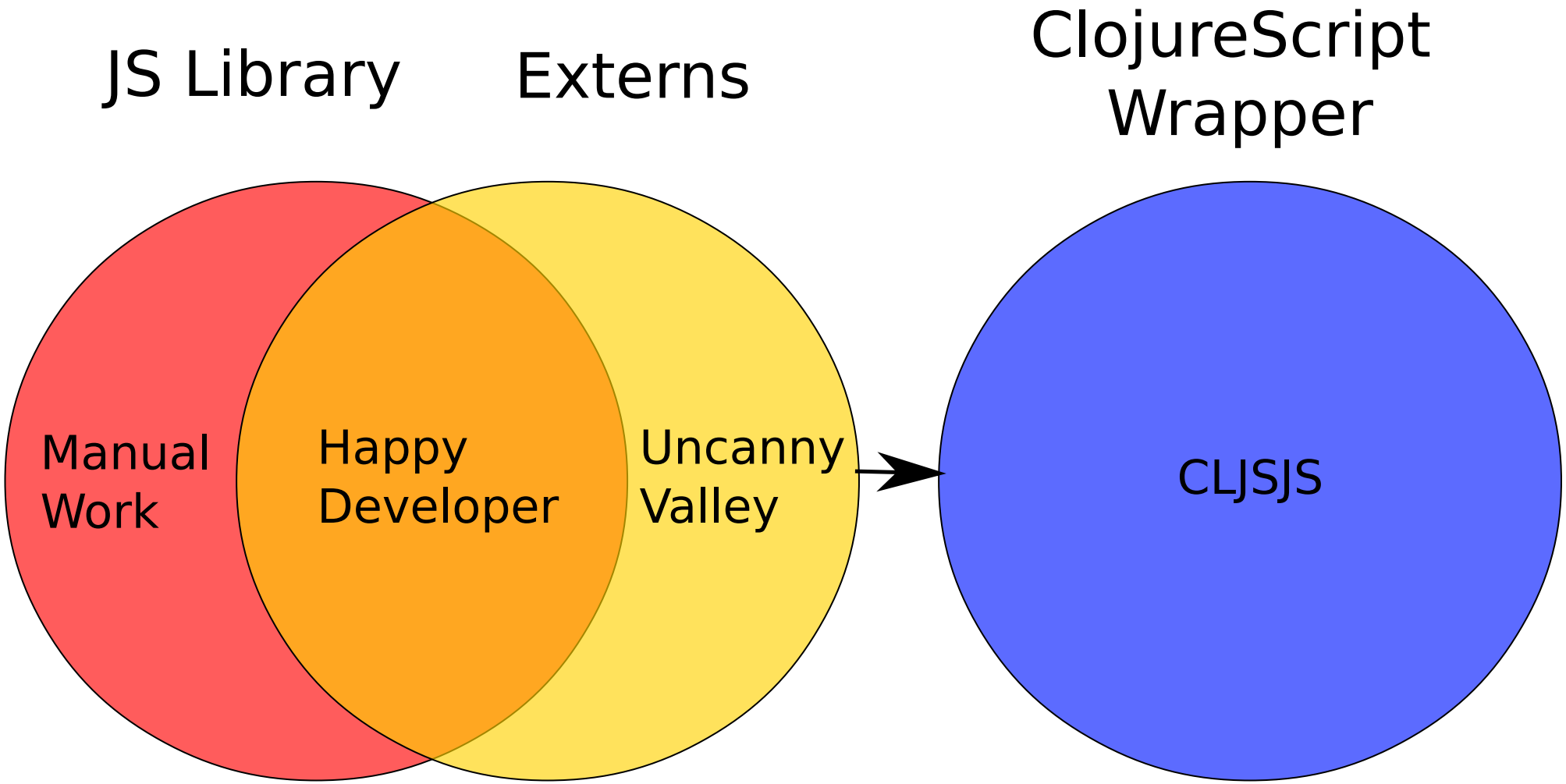
```
foo.bar("hello");
```

All that is needed is to write down all the exported names of the projects JavaScript libraries

JS Library

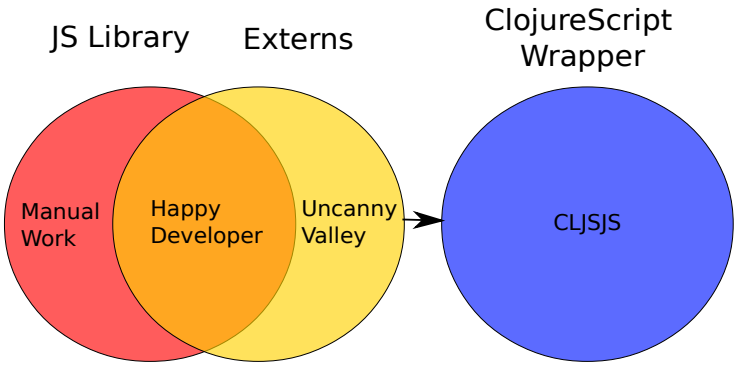



JS Library
Externs
Manual
Work
Happy
Developer
Uncanny
Valley

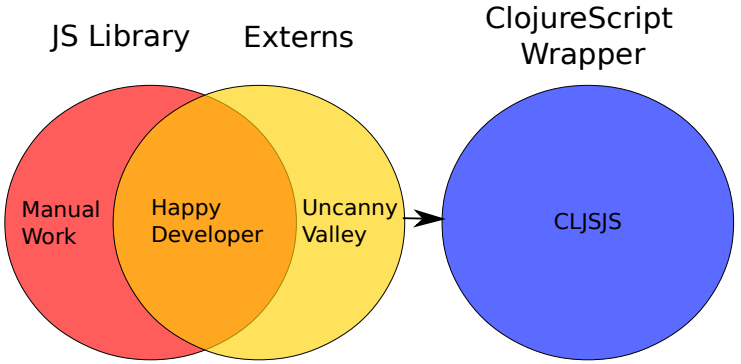

JS Library
Externs
ClojureScript
Wrapper
Manual
Work
Happy
Developer
Uncanny
Valley
CLJSJS

Repeat for every version

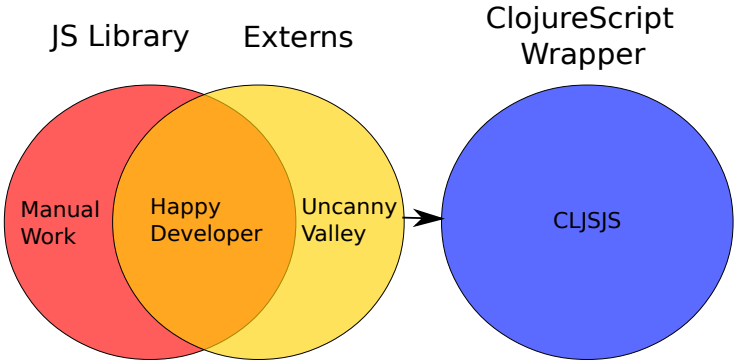
1.1.0



1.1.5



1.2.7



Caught in the Pit of Despair

Only most popular libraries are readily usable

We can escape the Pit Of Despair

- ClojureScript Externs Inference
- cljs-oops library

Externs Inference

ClojureScript can generate externs for us as of 1.9.456

Compiler Option

Turns on Externs Inference and writes an externs file inferred_externs.js

```
:infer-externs true
```

Infer Waring Flag

Turns on inference warnings

```
(set! *warn-on-infer* true)
```

Three types of inference warnings

- Use of an unknown JavaScript type
- Using Base JavaScript Object
- Calling an Unknown method or property on a known extern

```clojure
(defn cloudy [outside]
  (.getClouds outside))
```

"Cannot infer target type for ..."

Unknown Object → outside

Weather → outside

Type Hint outside with js/Weather

```
(defn cloudy [^js/Weather outside]
  (.getClouds outside))
```

```
(defn cloudy [^js/Weather outside]
  (let [clouds (.getClouds outside)]
    (.getType clouds)))
```

"Adding extern to Object for ..."

outside.getClouds() → Object JS

outside.getClouds() → Clouds 

Wrap getClouds function in ClojureScript

```
(defn ^js/Clouds get-clouds [^js/Weather outside]
  (.getClouds outside))

(defn cloudy [outside]
  (let [clouds (get-clouds outside)]
    (.getType clouds)))
```

Add return type to getClouds and add Clouds externs to inferred_externs.js

```
/*
 * @return {Clouds}
 */
Weather.prototype.getClouds;

var Clouds;
Clouds.prototype.getType = function() {};
```

```
(defn cloudy [^js/Weather outside]
  (let [clouds (.getClouds outside)]
    (.frog clouds)))
```

"Cannot resolve property ..."

Clouds.prototype.frog does not exist in externs

Change (.frog clouds) to (.getType clouds)

Add frog to inferred_externs.js

```
Clouds.prototype.frog = function() {};
```

cljs-oops

Sidestep externs entirely using string names

Access string names via goog.object/get or aget
Advanced optimization does not rename String names

```
(ns my.app
  (:require [goog.object :as gobj]))

(defn cloudy [outside]
  (.call (gobj/get outside "getClouds") outside))
```

cljs-oops provides macros for automation

oget

Retrieve JavaScript Object properties

```
(def home #js {"floor" #js {"living-room" "500 sqft"}})
(oget home "floor" "living-room")
;; => "500 sqft"
```

oset!

Set JavaScript Object properties

```
(def home #js {"floor" #js {"living-room" "500 sqft"}})
(oset! home "floor" "living-room" "300 sqft")
;; => #js {"floor" #js {"living-room" "300 sqft"}}
```

ocall

Call JavaScript methods with fixed arguments

```
(def car #js {"ispy" (fn [desc item] (str "I see a " desc " " item))})
(ocall car ["ispy"] "red" "barn")
;; => "I see a red car"
```

oapply

Call JavaScript methods with variadic arguments

```
(def bill #js {"total" (fn [& items] (reduce + items))})
(oapply bill "total" [1 2 3])
;; => 6
```

cljs-oops not just automation

extensive validation during development
emits optimized code during advanced compilation

Navigating JavaScript Objects
Access Modifiers

- ? soft access, returns nil for non existent key
  - Change key to ?key
- ! punching, creates key when it does not exist
  - Change key to !key

? soft access
Like get-in

```
(def home #js {"house" #js {"bedroom" #js {:color "red"}}})
(oget home "house" "?livingroom")
;; => nil
```

```
(def home {:house {:bedroom {:color "red"}}})
(get-in home [:house :living-room])
;; => nil
```

! punching
Like assoc-in

```
(def home #js {})
(oset! home "!house" "!livingroom" "!color" "green")
;; => #js {"house" #js {"livingroom" #js {"color" "green"}}}
```

```
(def home {})
(assoc-in home [:house :livingroom :color] "green")
;; => {:house {:livingroom {:color "green"}}}
```

Use in small doses

String Names defeats Google Closure Advanced Optimization

Opens up the JavaScript ecosystem

Selectively use library features with

- ClojureScript externs inference
- cljs-oops

CLJSJS library is the best option

Externs file next best options

When neither exist use

- ClojureScript externs inference
- cljs-oops

Compiling Node Modules

Miracle Pill

Google Closure Compiler can compile node modules

- Facebook React and React DOM 53k
- Google Closure React and React DOM 32k

New compiler option :npm-deps

```
{:npm-deps {:react "15.4.2"
            :react-dom "15.4.2"}}
```

react and react-dom are now just libraries

```
(ns my.app
  (:require [react :as React]
            [react-dom :as ReactDOM]))

(def app (React/createElement "h1" nil "Hello World!"))
(ReactDOM/render app (.getElementById js/document "app"))
```

React and ReactDom are namespaces Use like this

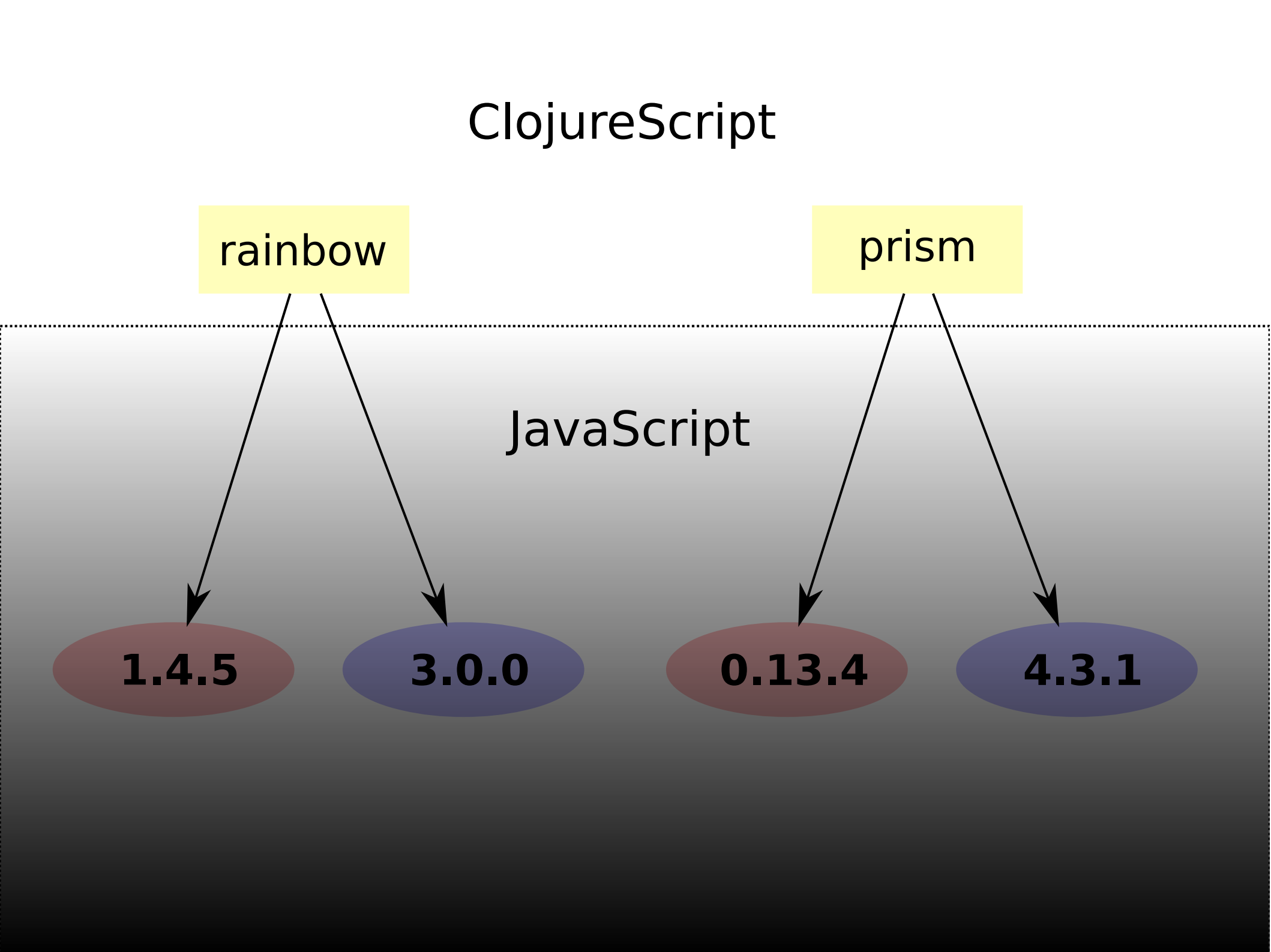
```
(React/createElement "hi" nil "Hello World")
```

Not CLJSJS Style

```
(.createElement js/React "hi" nil "Hello World")
```

Now for the caveats, addendum's, and hoop jumping

ClojureScript does not manage JavaScript dependency graph


ClojureScript
rainbow
prism
JavaScript
1.4.5
3.0.0
0.13.4
4.3.1

Application development is the big winner

- npm or yarn can manage JavaScript dependencies
- Minimal code size with Advanced optimizations

module-deps JavaScript library needed

- npm install --save-dev module-deps
- yarn add --dev module-deps

Externs for node needed

JavaScript uses node to compile JavaScript

https://github.com/dcodeIO/node.js-closure-compiler-externs

CLJSJS libraries for externs

Tells Closure about dynamic (meta programmed) names

```
[cljsjs/react "15.4.2-2"]
[cljsjs/react-dom "15.4.2-2"]
```

Making the miracle pill from scratch

Useful for debugging purposes

To create a miracle pill from scratch first you must create the universe.

Install react and react-dom

npm

- npm install --save react@15.4.2
- npm install --save react-dom@15.4.2

yarn

- yarn add react@15.4.2
- yarn add react-dom@15.4.2

Create the recipe for the miracle pill

Setup dependencies and exports

```
var React = require("react");
var ReactDOMServer = require("react-dom");

module.exports = {
  React: React,
  ReactDOM: ReactDOM
};
```

Collect the ingredients for the miracle pill

Pass in file as a foreign-lib to cljs.closure/node-inupts

```
(require '[clojure.java.io :as io])
(require 'cljs.closure)

(def root-js-deps
  {:file (.getAbsolutePath (io/file "path/to/npm_deps.js"))
   :provides ["libs.npm-deps"]
   :module-type :commonjs})

(def node-libs
  (into [entry] (cljs.closure/node-inputs [root-js-deps])))
```

Finally ready to make the miracle pill

Pass cljs.closure/node-inputs result to :foreign-libs

```
(require 'cljs.build.api)

(cljs.build.api/build "src"
  {:optimizations :advanced
   :output-to "out/app.js"
   :foreign-libs node-libs})
```

Same result as :npm-deps

react and react-dom are in lib.npm-deps

```
(ns my.app
  (:require [lib.npm-deps :as npm-deps]))

(def app (npm-deps/React.createElement "h1" nil "Hello World!"))
(npm-deps/ReactDOM.render app (.getElementById js/document "app"))
```

Still very alpha

It's not just ClojureScript working on this

Major players are integrating Google Closure

React - Fiber Build

https://github.com/facebook/react/issues/7925

Angular - Offline Template Compilation

https://github.com/angular/angular/issues/8550

Typescript - tsickle

https://github.com/angular/tsickle

Dynamically Loading ClojureScript Modules

ClojureScript Modules pretty straight forward

```
:modules {:extra {:output-to "resources/modules/extra.js"
                  :entries #{"my.module.core"}}
          :dev   {:output-to "resources/modules/dev.js"
                  :entries #{"my.module.root"}
                  :depends-on #{:extra}}}
```

Google Closure does the hard work
Cross Module Code Motion

Dynamically Loading ClojureScript Modules

Morass of OOP boiler plate

Module Management OO Style

```
;; Singleton Module Manager
(def manager (.getInstance goog.module.ModuleManager))

(def loader (goog.module.ModuleLoader.))
(.setLoader manager loader)

(def modules
  ;; id -> urls
  #js {"extra" "resources/modules/extra.js"})

(def module-info
  ;; id-> dependencies
  #js {"extra" #js []})

(.setAllModuleInfo manager module-info)
(.setModuleUris manager modules)
```

Mark Module as Loaded

```
(ns my.module.name)

(.setLoaded (.getInstance goog.module.ModuleManager) "my.module.name")
```

Just for Module Manager bookkeeping

Modules still need to get dynamically loaded

Desired End Result

```
(ns my.module.root)

(load-module "extra" (fn [] (.log js/console "extra loaded")))
```

Development

```
:optimizations :none
```

:modules is not available
All namespaces are auto loaded

Put all module namespaces in :preloads

```
:preloads '[my.module.extra]
```

Loading Modules in Development

Check every 100ms if module has been auto loaded

```
(defn load-module-dev [id callback]
  (let [interval (goog.Timer. 100)
        manager (.getInstance goog.module.ModuleManager)
        loaded? (fn []
                  (if-let [module (.getModuleInfo manager id)]
                    (.isLoaded module) false))
        tick-fn (fn [_]
                  (when (loaded? id)
                    (.stop interval)
                    (goog.events/removeAll interval)
                    (callback)))]
    (goog.events/listen interval "tick" tick-fn)
    (.start interval)))
```

Loading Modules in Production

```
(defn load-module-prod [id callback]
  (.execOnLoad (.getInstance goog.module.ModuleManager) id callback))
```

Choose module loader with goog-define

```
(goog-define PRODUCTION false)

(def load-module
  (if ^boolean PRODUCTION
    load-module-prod
    load-module-dev))
```

It would be great ClojureScript library for this

There is
https://github.com/bendyworks/conwip-modules
[conwip.modules 0.1.0]

We've gone though ClojureScript's fire swamps and come out alive

People see only one **Dread Pirate Roberts**

There are many **Dread Pirate Roberts**

Thanks

- Bendyworks for supporting my presentation
- Antonin Hildebrand for cljs-oops and helping me go over it
- António Monteiro for presentation help and node modules reference
- David Nolen for helping with presentation and ClojureScript details
- Allen Rohner for blog post on "Dynamic ClojureScript Module Loading"
- Martin Klepsch for blog post on "Parameterizing ClojureScript Builds"

Additional Topics

Profiling ClojureScript

JavaScript Just In Time (JIT) Profiling

Questions?