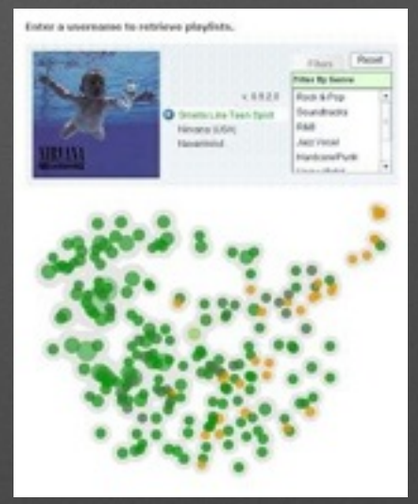
promhx

Programming Reactive Object Methods in Haxe

A bit about me…



Data Visualization + Data Science + Machine Learning + Advanced UI

I “Promhx” to explain…

```
var p = new Promise<Explanation>();

var s = new Stream<Examples>();

var d = new Deferred<Understanding>();
```

But first some background…

Asynchronous Programming

- *Client code is often single threaded. Excessive time delays due to computation causes rendering lag, and can block IO.*
  We often need to break up computation in multiple loops.

- *Client code often handles server responses through asynchronous callbacks.*
  We need to wait for the server to respond.

- **Managing error handling often involves careful use of try-catch in several locations. It is difficult to ensure all exceptions are caught.**
  We want a better way of handling generic errors.

- *Callbacks generally follow a nested pattern*
  Nested code indentation patterns can become difficult to read.

- *Server side platforms are finding success with asynchronous programming techniques, with the same opportunities and pitfalls (nodejs). However, server side implementations often have different methods of managing asynchronous callbacks.*
  We want to implement asynchronous code the same way on client/server.

Asynchronous Programming

How many potential problems in this code?

1. bash

0 jjd 1 : vim 2 vim — 91% | Wed May 21 16:04:03 PDT 2014

```
1  class Callback {
2      public static function main() {
3          try {
4              var obj = { x : 10 };
5              haxe.Timer.delay(function(){
6                  trace("inside method 1");
7                  haxe.Timer.delay(function(){
8                      trace('the value of obj.x is ${obj.x}');
9                      haxe.Timer.delay(function(){
10                         trace('oh no... we are quickly running out of useful horizontal space due to all the indentati>
11                     }, 0);
12                 }, 0);
13             }, 0);
14             obj = null;
15         } catch(e:Dynamic){
16             trace('error caught: $e');
17         }
18     }
19 }
~
~
~
~
```

Asynchronous Programming

Lost space due to indentation

Missed error handling

Not *really* 0 seconds

Duplicate error handling

Functional Reactive Programming to the Rescue

Wikipedia : *“In computing, reactive programming is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.”*

Reactive programming moves the unit of asynchronous programming from a callback to a special object. It is a variable over time.

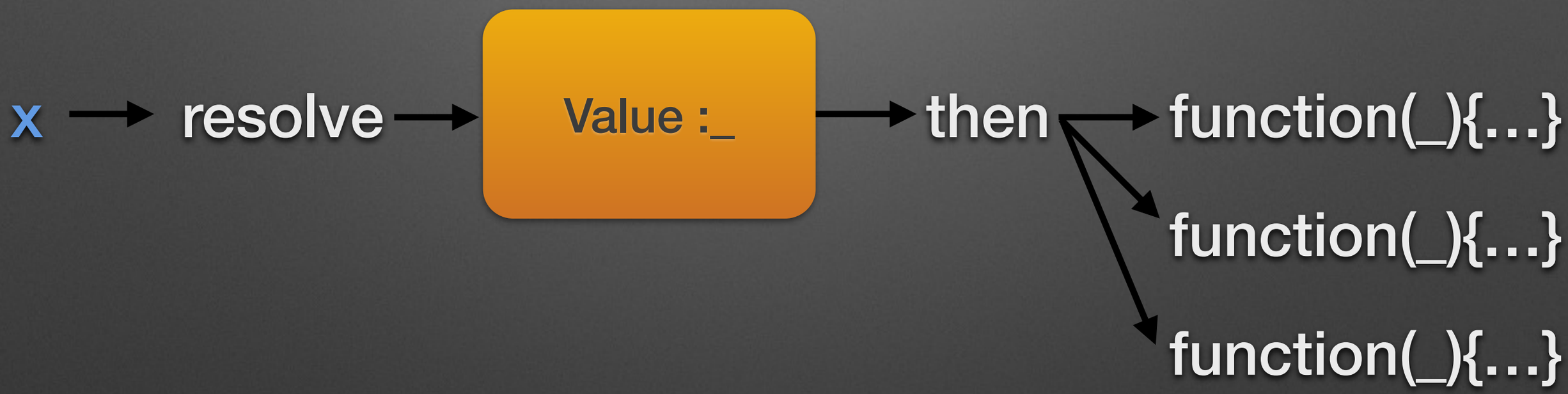
Promhx Object

Container Object

Value : _

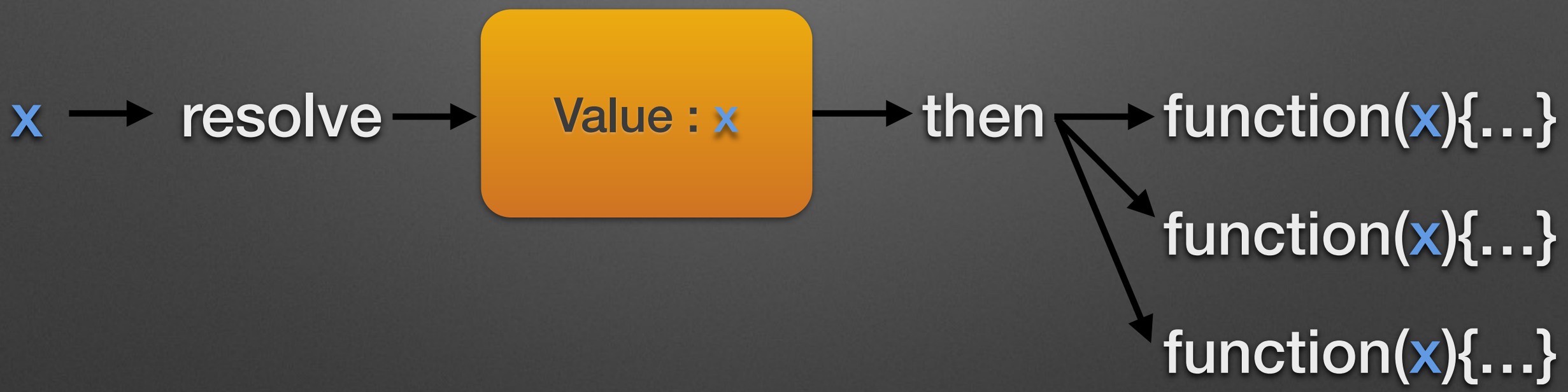
Promhx Object

Container Object

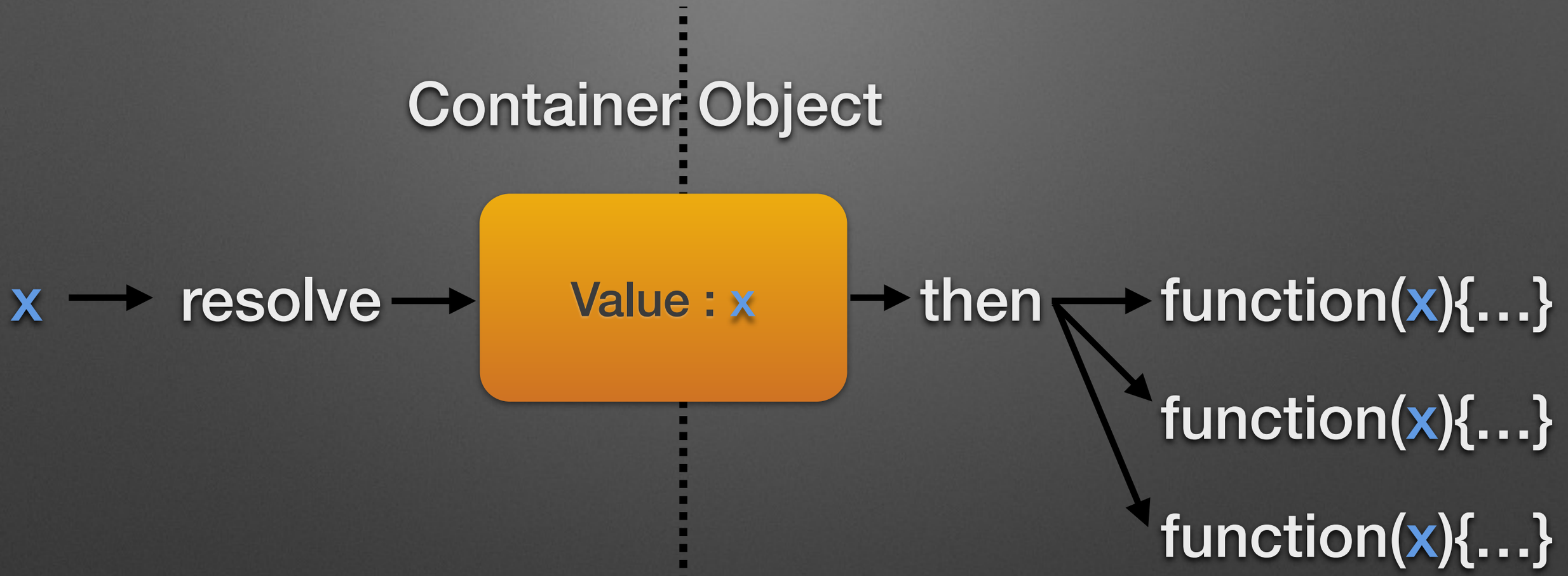


Promhx Object

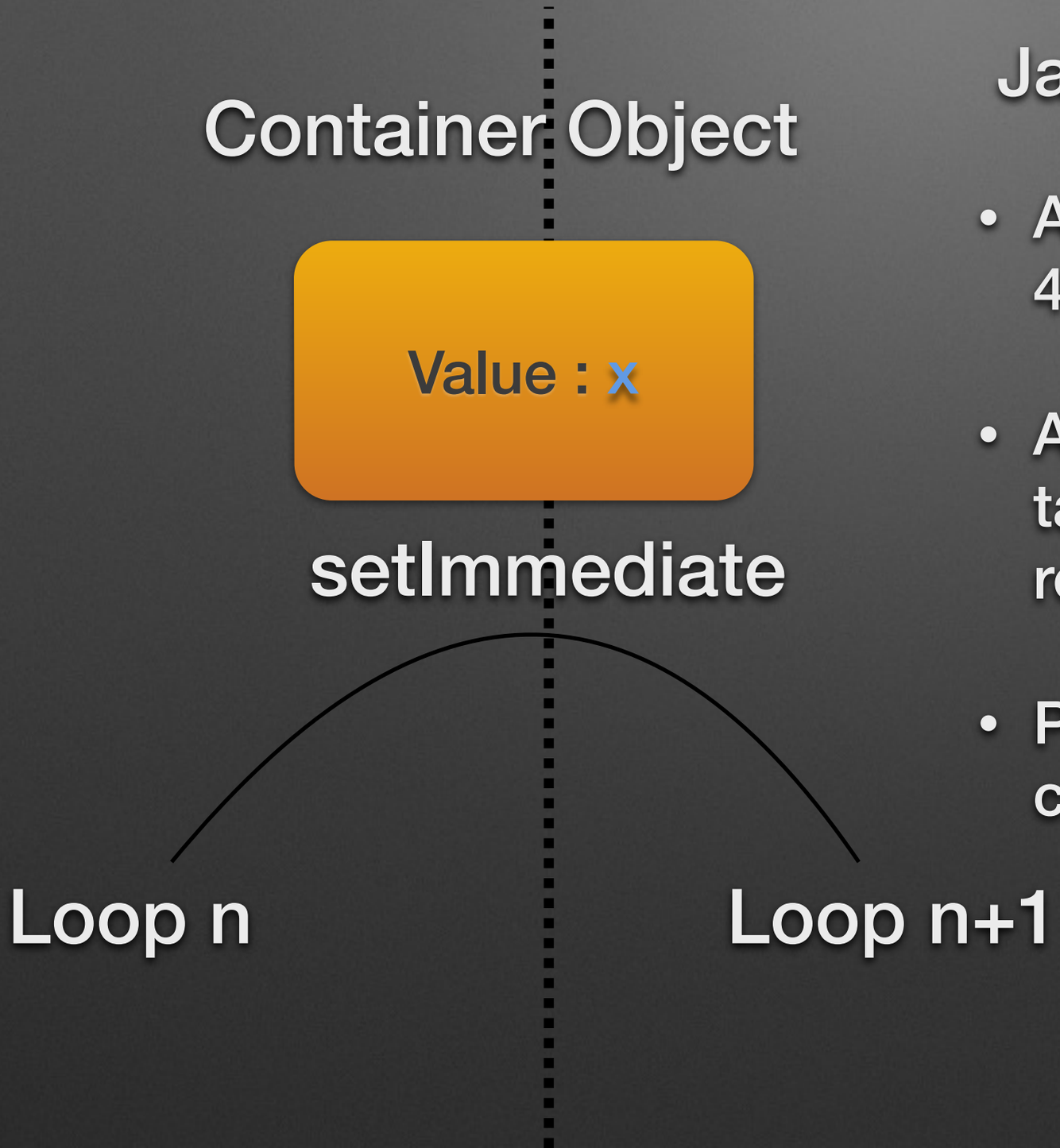
Container Object



Promhx Object



Promhx Looping



Javascript specific details

- Avoid the default delay of 4ms-10ms in setTimeout
- Avoid scheduling a new paint task in requestAnimationFrame
- Provide polyfill for backwards compatibility (via noble.js)

Promhx Looping

Container Object

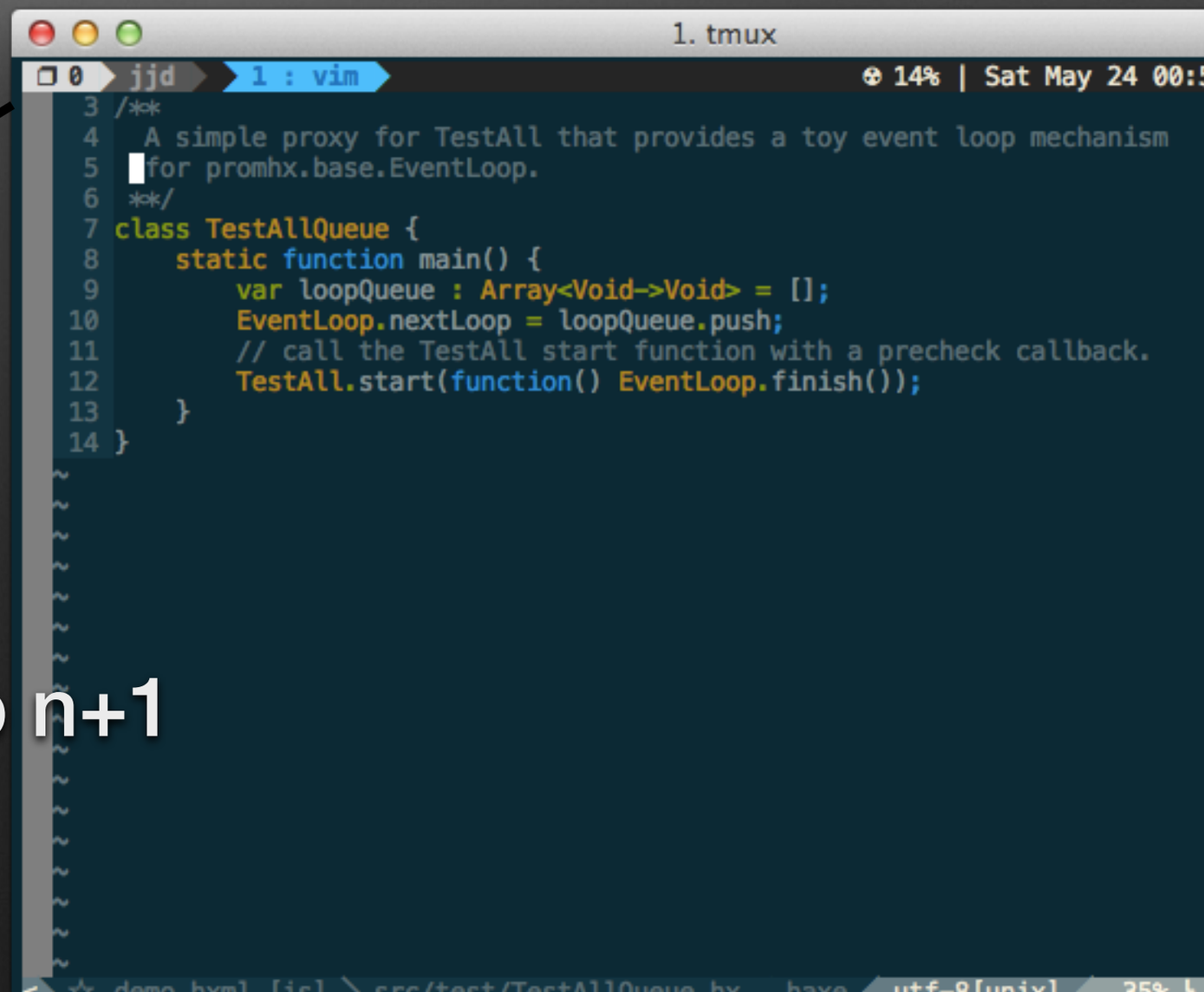
Value : x

???

Loop n

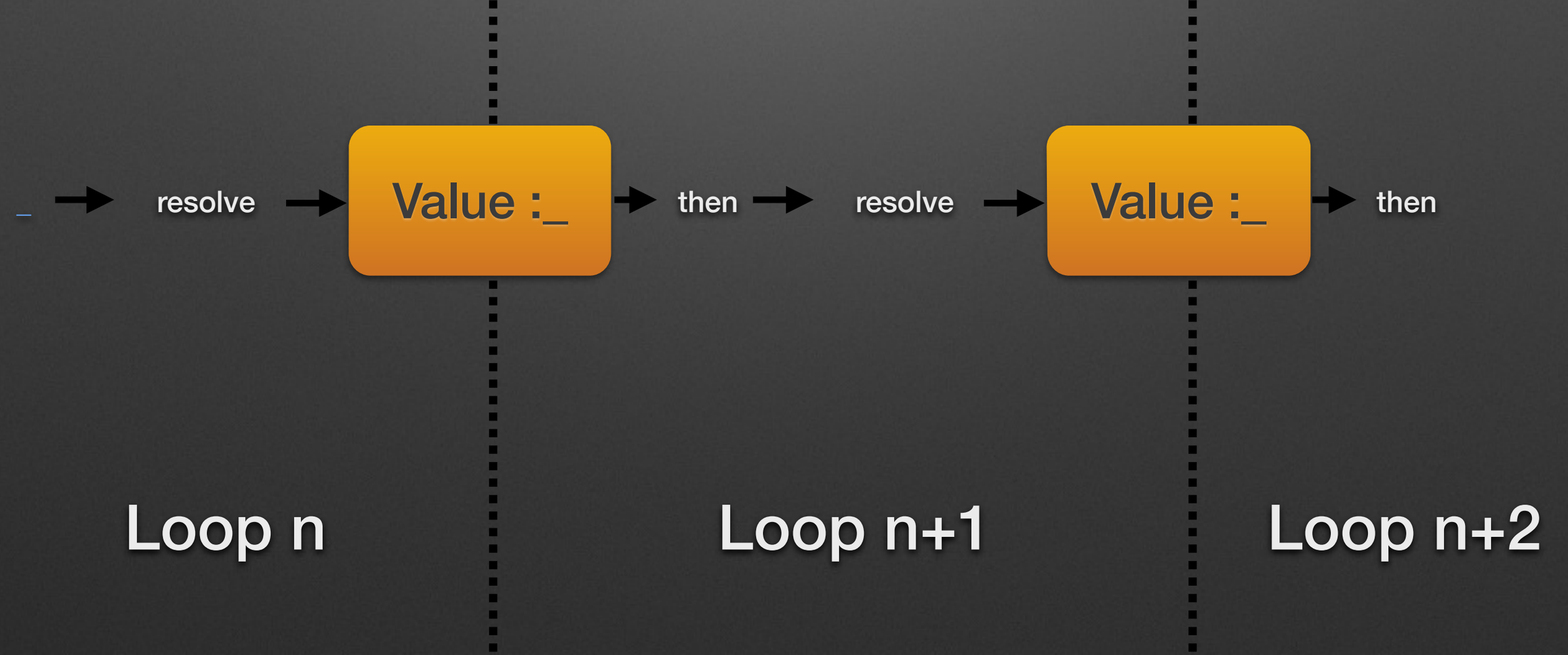
Loop n+1

Other platforms : BYOEL
(bring your own event loop)



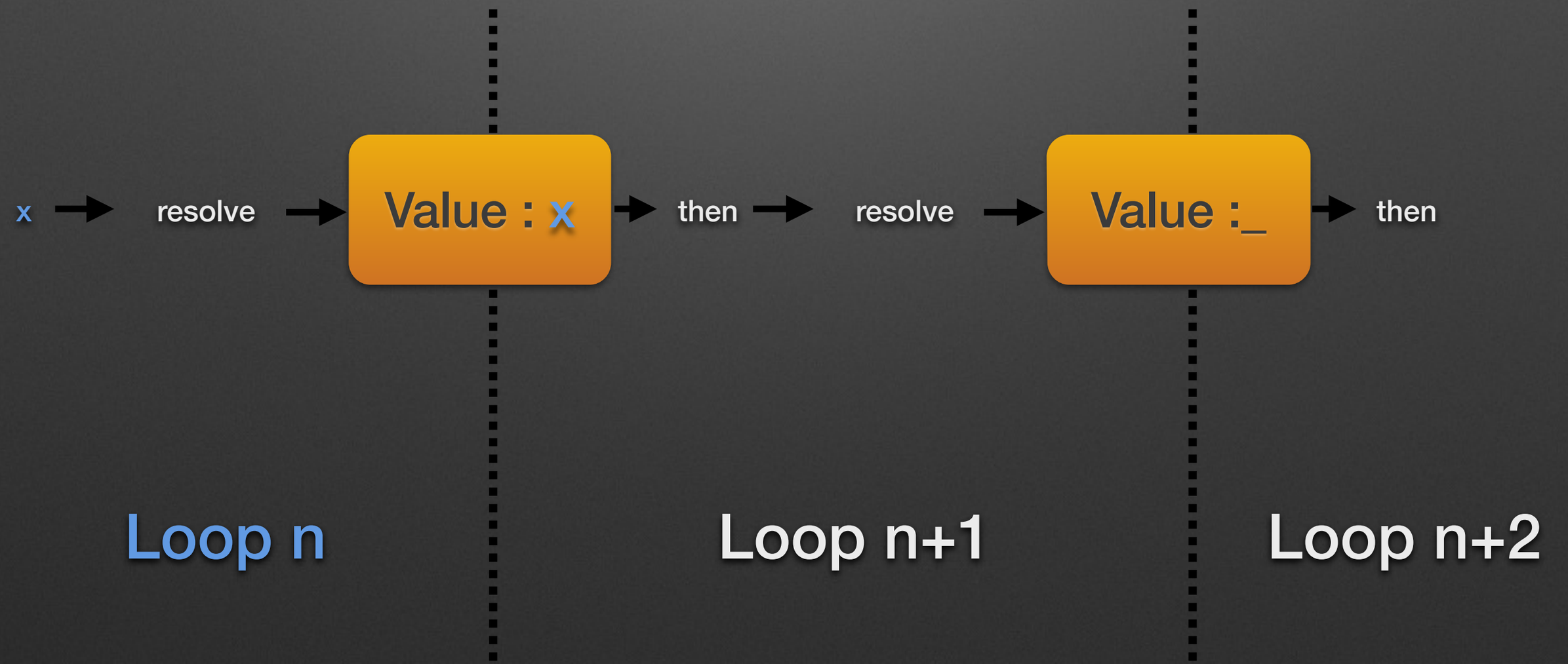
Promhx Object

Chaining



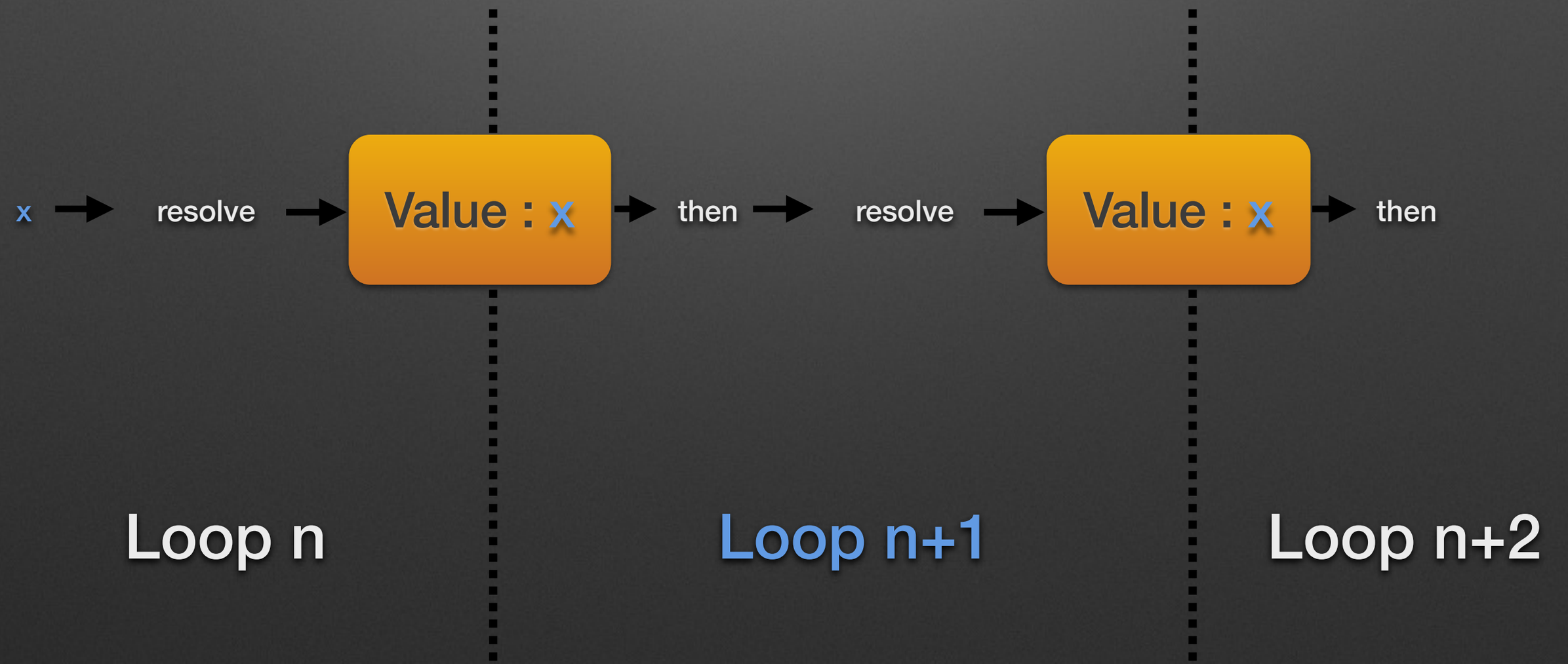
Promhx Object

Chaining



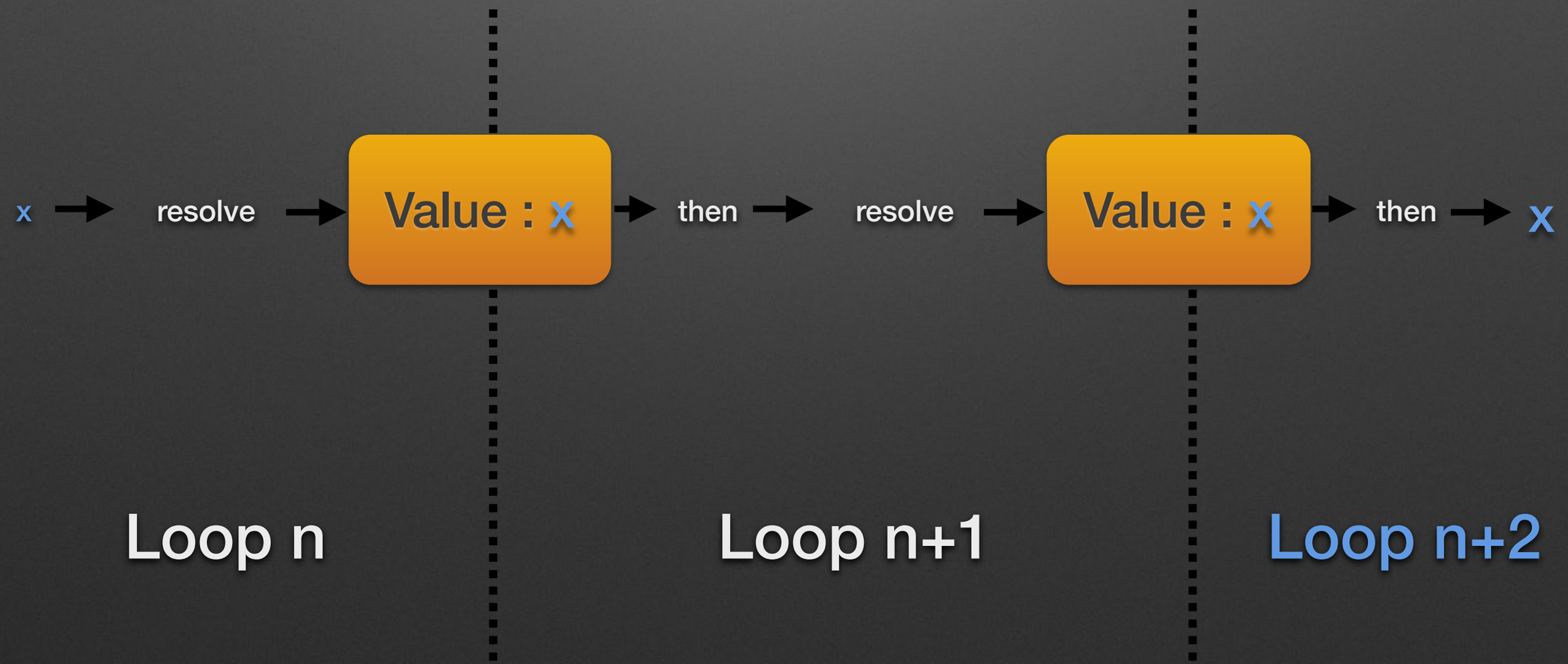
Promhx Object

Chaining



Promhx Object

Chaining



What is a Promise?

Wikipedia : “*In computer science, future, promise, and delay refer to constructs used for synchronizing in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete.*”

What is a (Reactive) Stream?

Wikipedia : “In computer science, a stream is a sequence of data elements made available over time. A stream can be thought of as a conveyor belt that allows items to be processed one at a time rather than in large batches.”

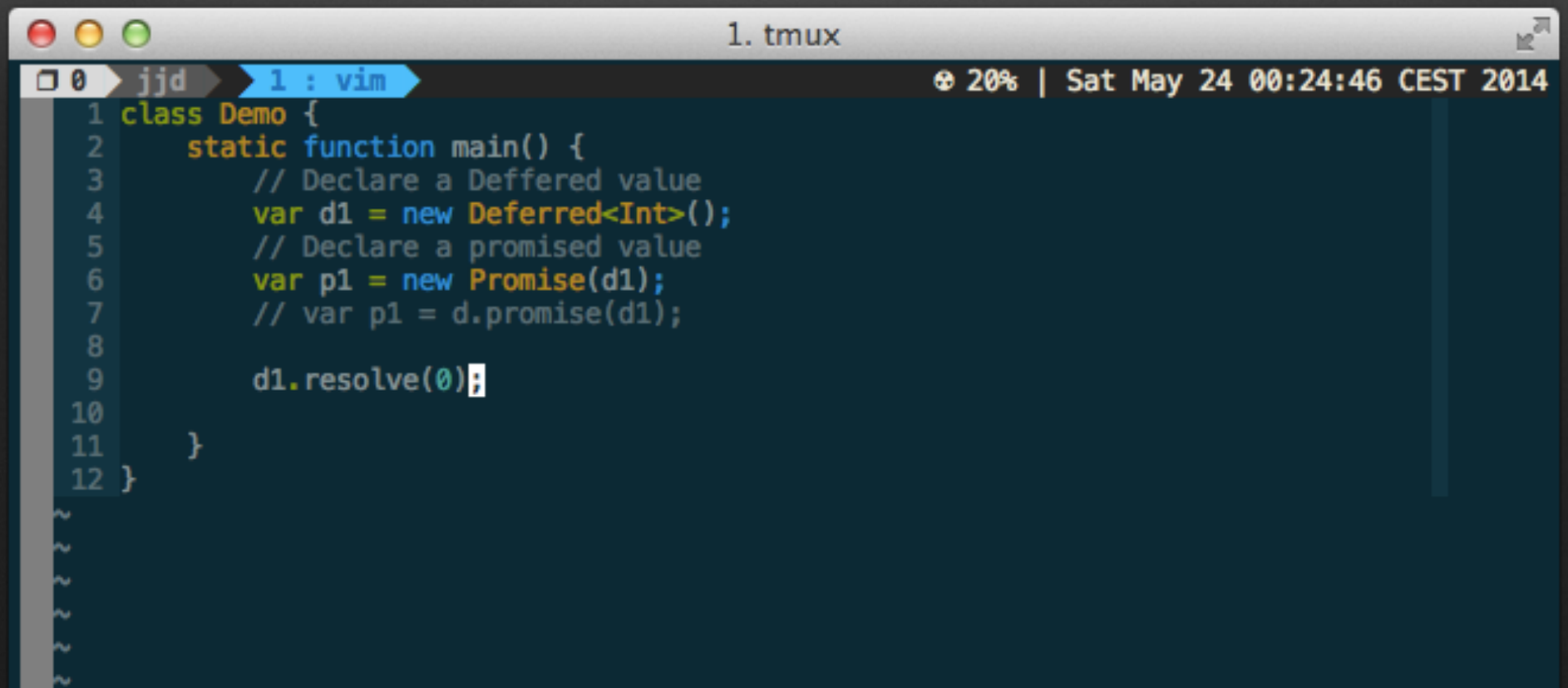
Promises vs. Streams

- Promises
  - Good for providing initial configuration, and ensuring that an asynchronous workflow progresses “forward”.
  - Can only resolve once!
- Streams
  - Good for providing general-purpose reactive programming.
  - Resolves more than once.
- Promises and Streams
  - Generally, promises can be substituted where streams are expected, but not vice-versa.
  - Both provide the same error semantics, and optimized asynchronous event loop management.
  - Both use Deferreds as a writable interface…

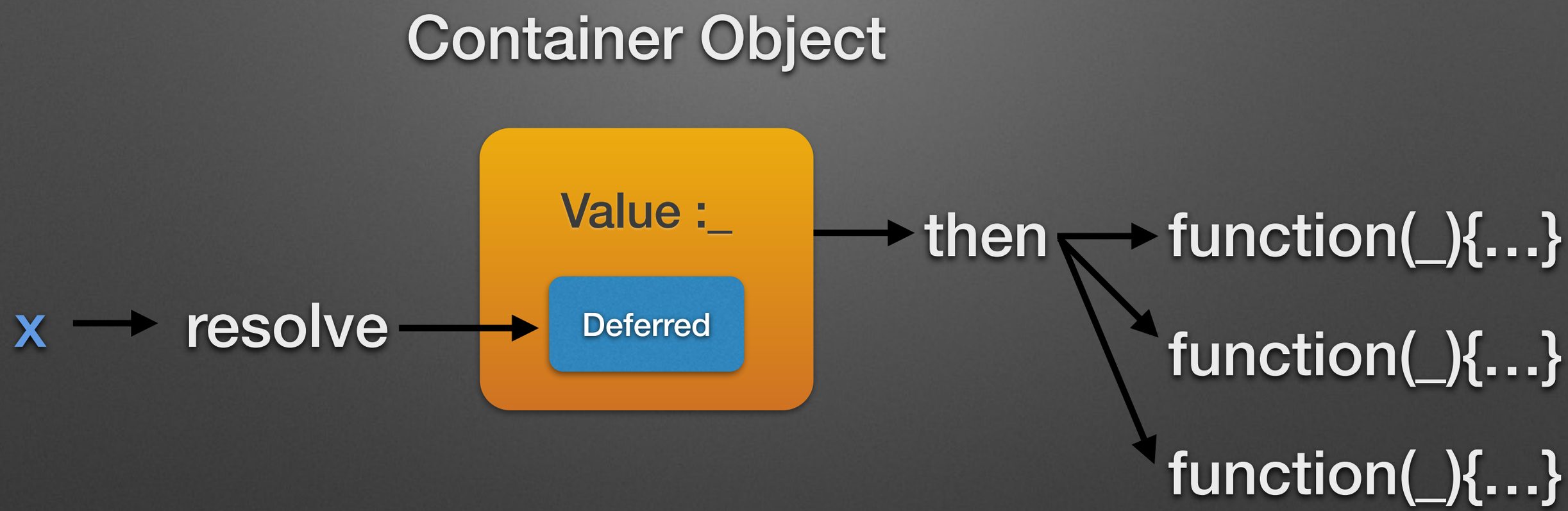
What is a Deferred?

Deferreds are a writable interface for reactive objects. Use them to prevent other developers from mistakenly resolving a Promise or Stream that they should not have access to.*

*The notable exception is PublicStream, which is globally write-accessible.

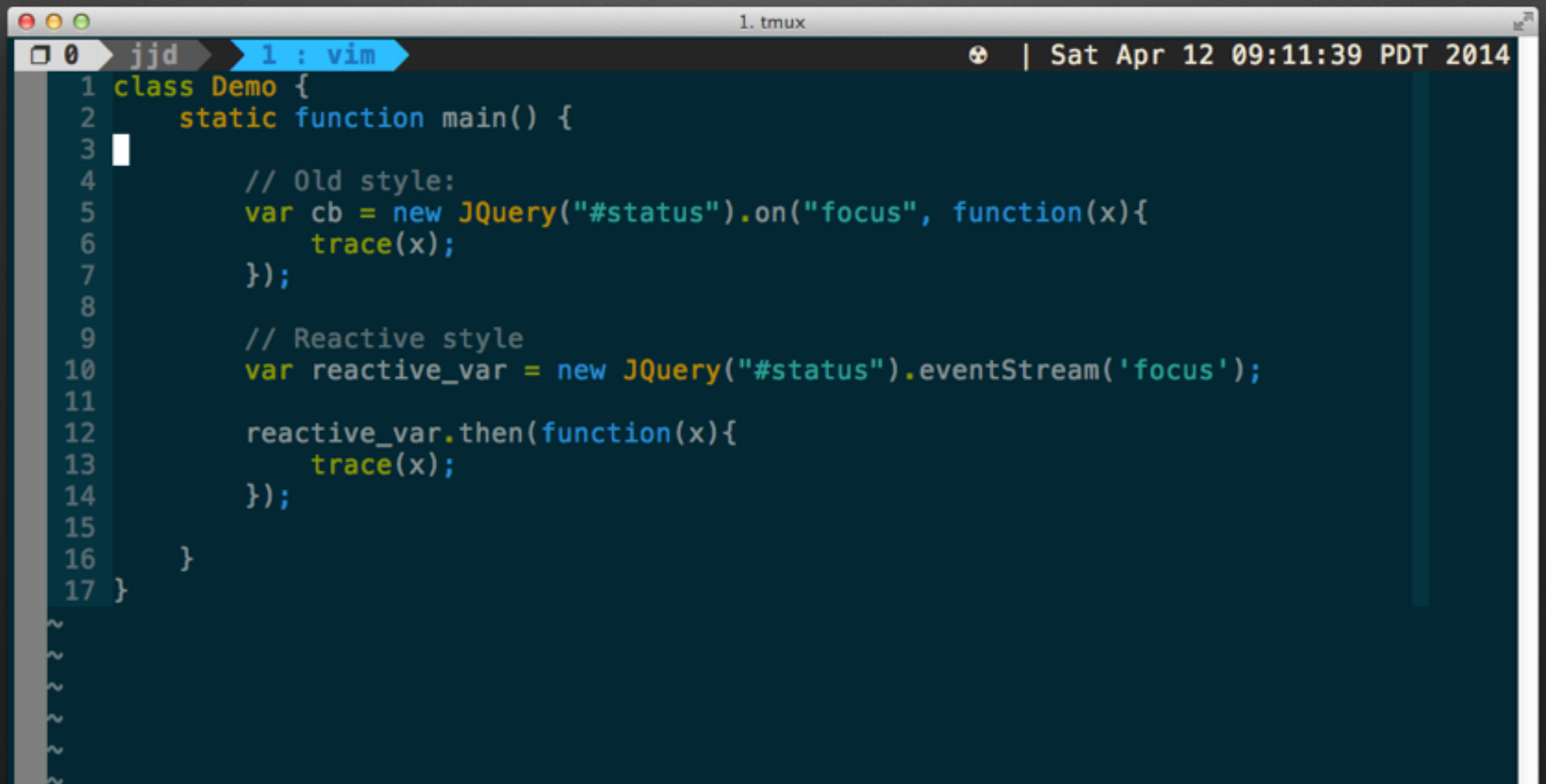


What is a Deferred?



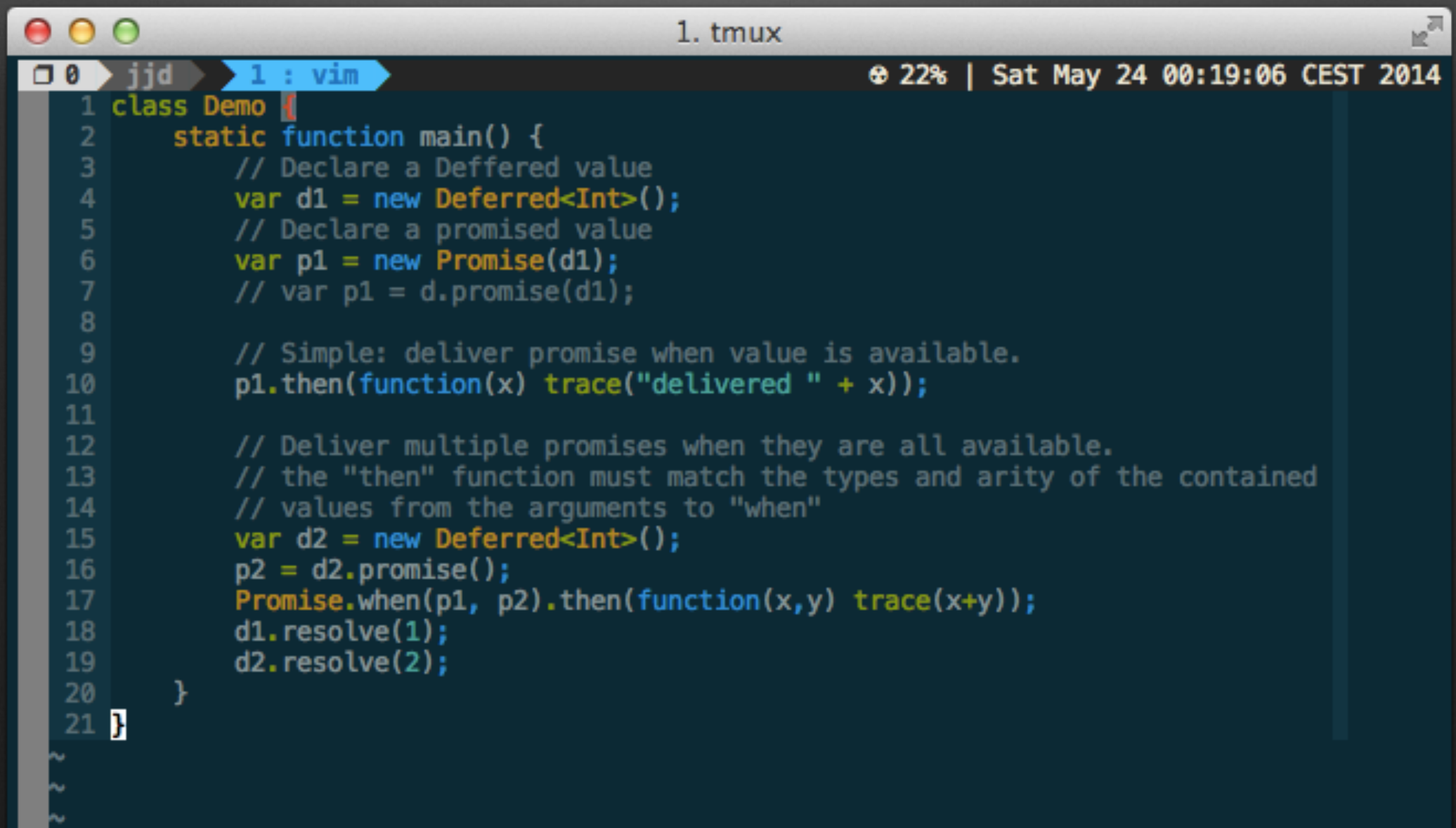
Basics of Reactive Programming in Promhx

Move the callback to the proxy object



Basics of working with Promises

then vs. when : instance vs. static usage



Error handling

- Errors are propagated through the reactive chain
- catchError() functions like a try/catch, preventing updates to other reactive variables.
- Use Haxe's try/catch semantics to catch typed errors.

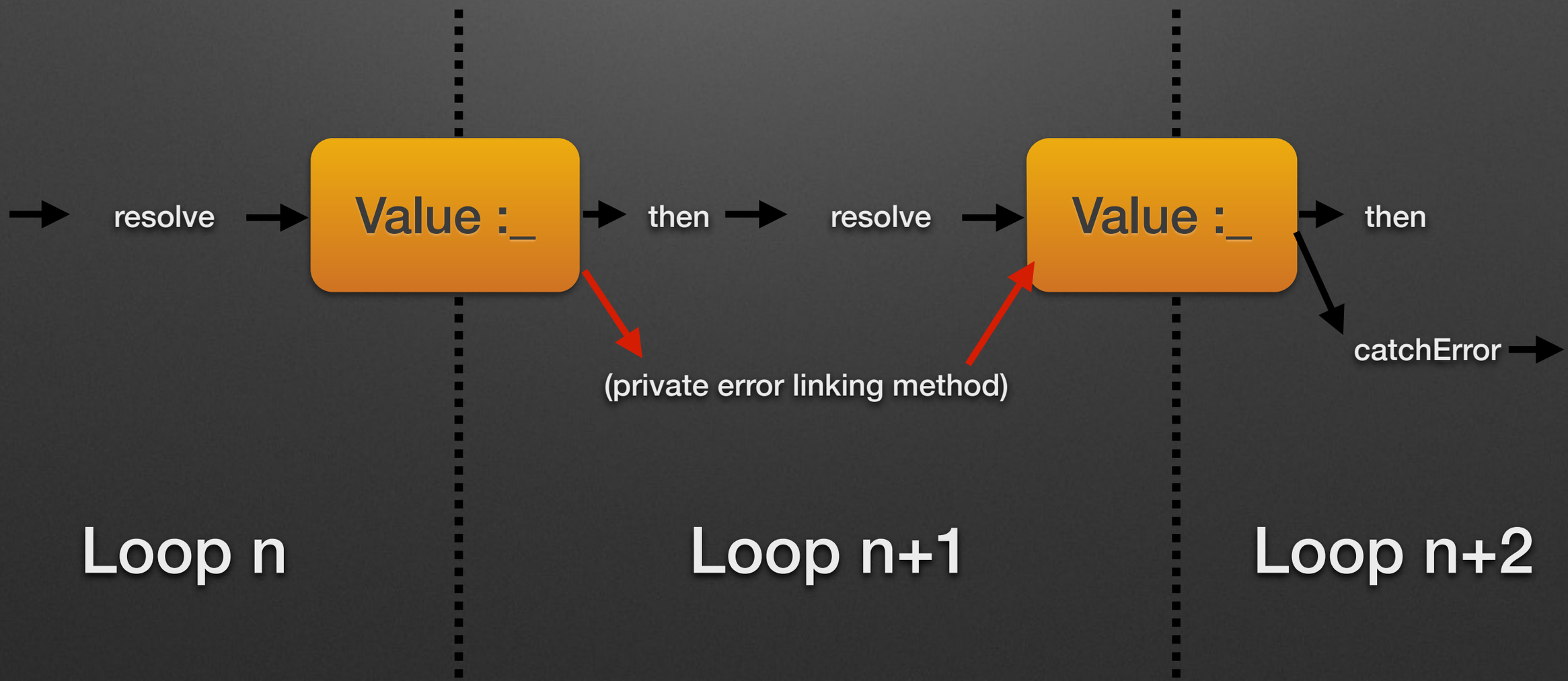
1. tmux

0 jjd 1 : vim | Sat Apr 12 09:33:38 PDT 2014

```
1  class Demo {
2     static function main() {
3        // Errors are propagated through the promise chain.
4        // You can rethrow errors to use Haxe's try/catch feature.
5        // Stream works the same here too.
6        Promise.when(p1,p2).then(function(x,y) {throw('an error'); return 'hi';})
7           .then(function(x) return 'a value')
8           .catchError(function(x) {
9              try {
10                throw(x); // rethrow the error value to do standard error handling
11             } catch(e:String){
12                trace('caught a string: ' + e);
13             } catch(e:Dynamic){
14                trace('caught something unknown:' + e);
15             }
16          });
17    }
18 }
```

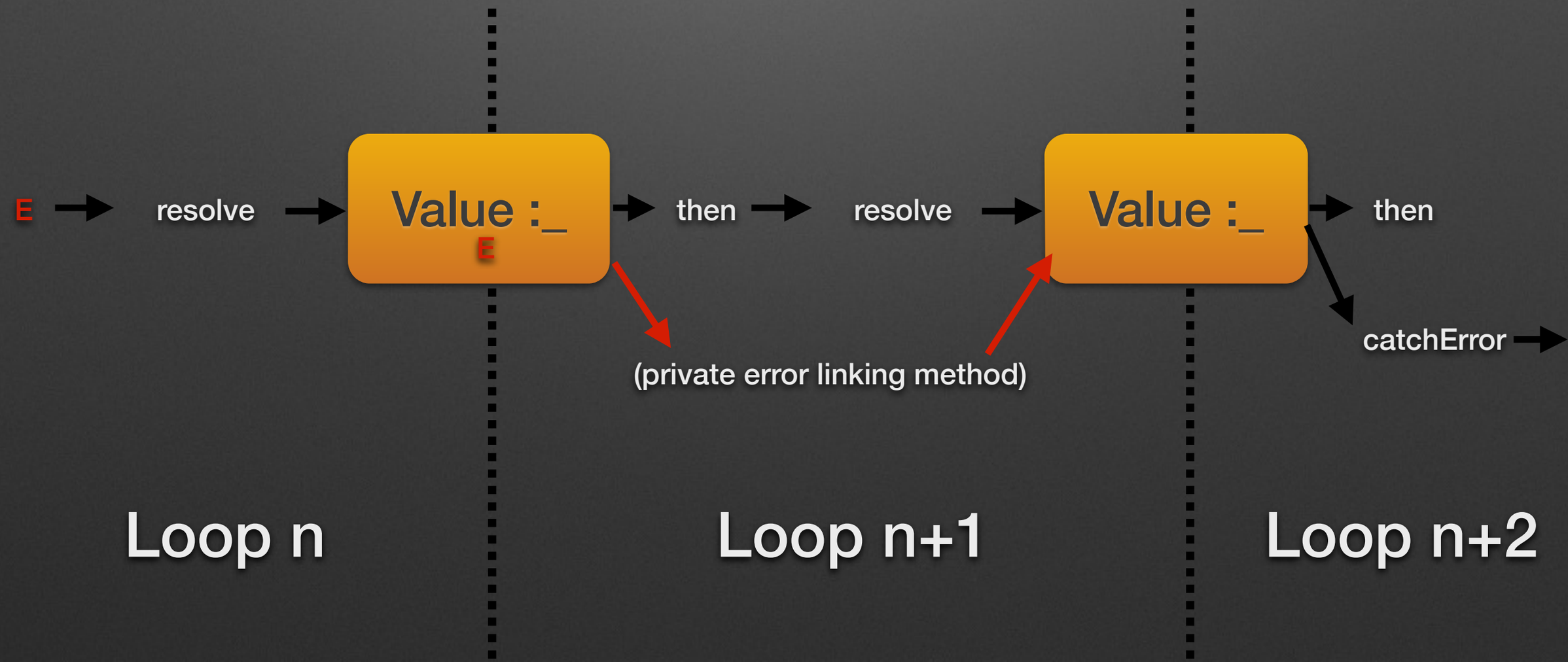
Promhx Object

Error Handling



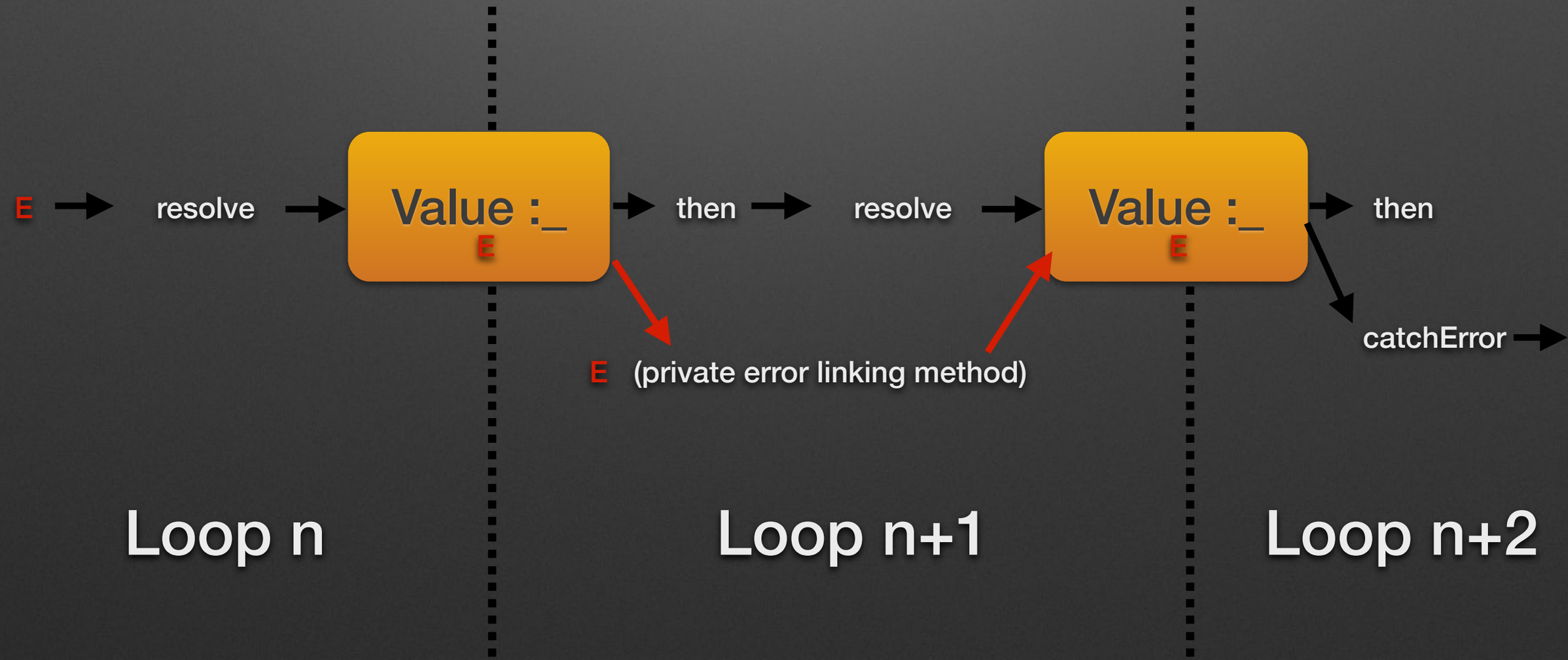
Promhx Object

Error Handling



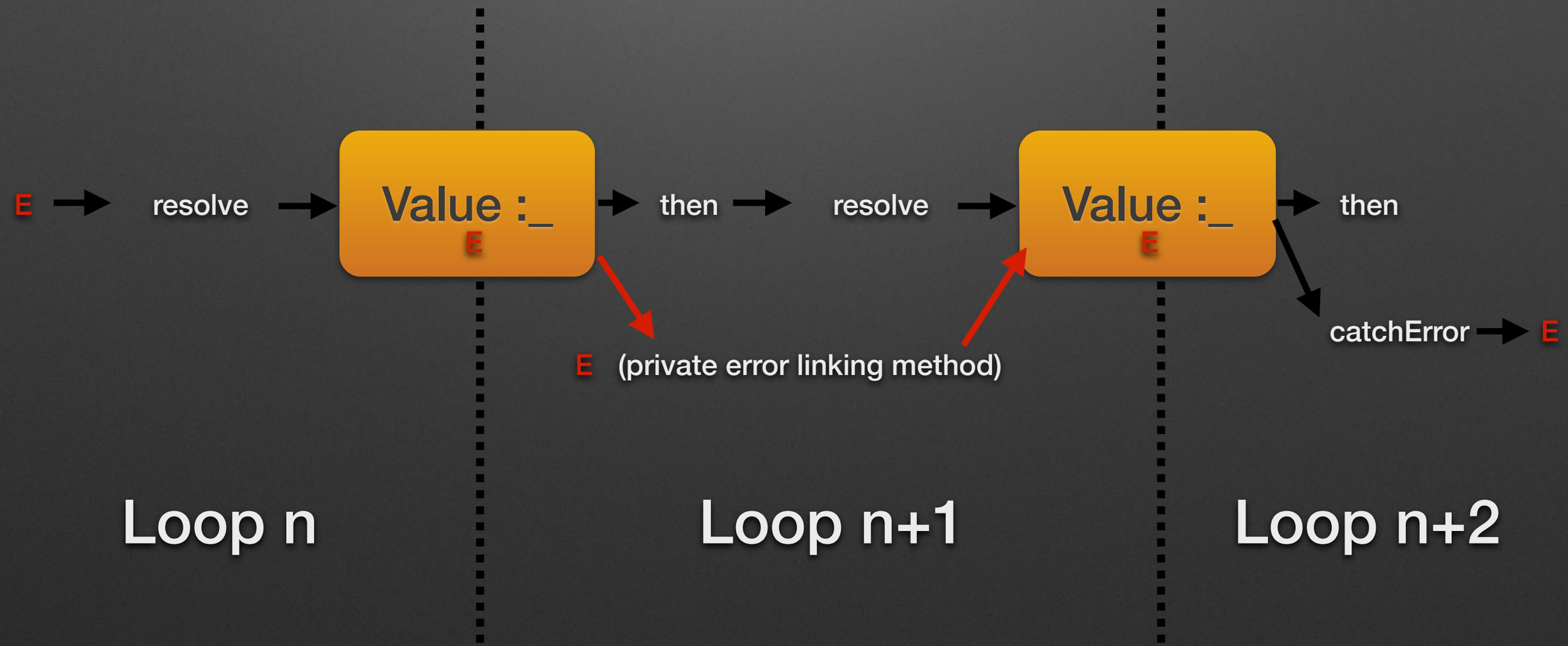
Promhx Object

Error Handling



Promhx Object

Error Handling



Error handling

- errorThen() allows you to convert an error back into the appropriate variable type.
- This variable can be propagated as if it were resolved normally.

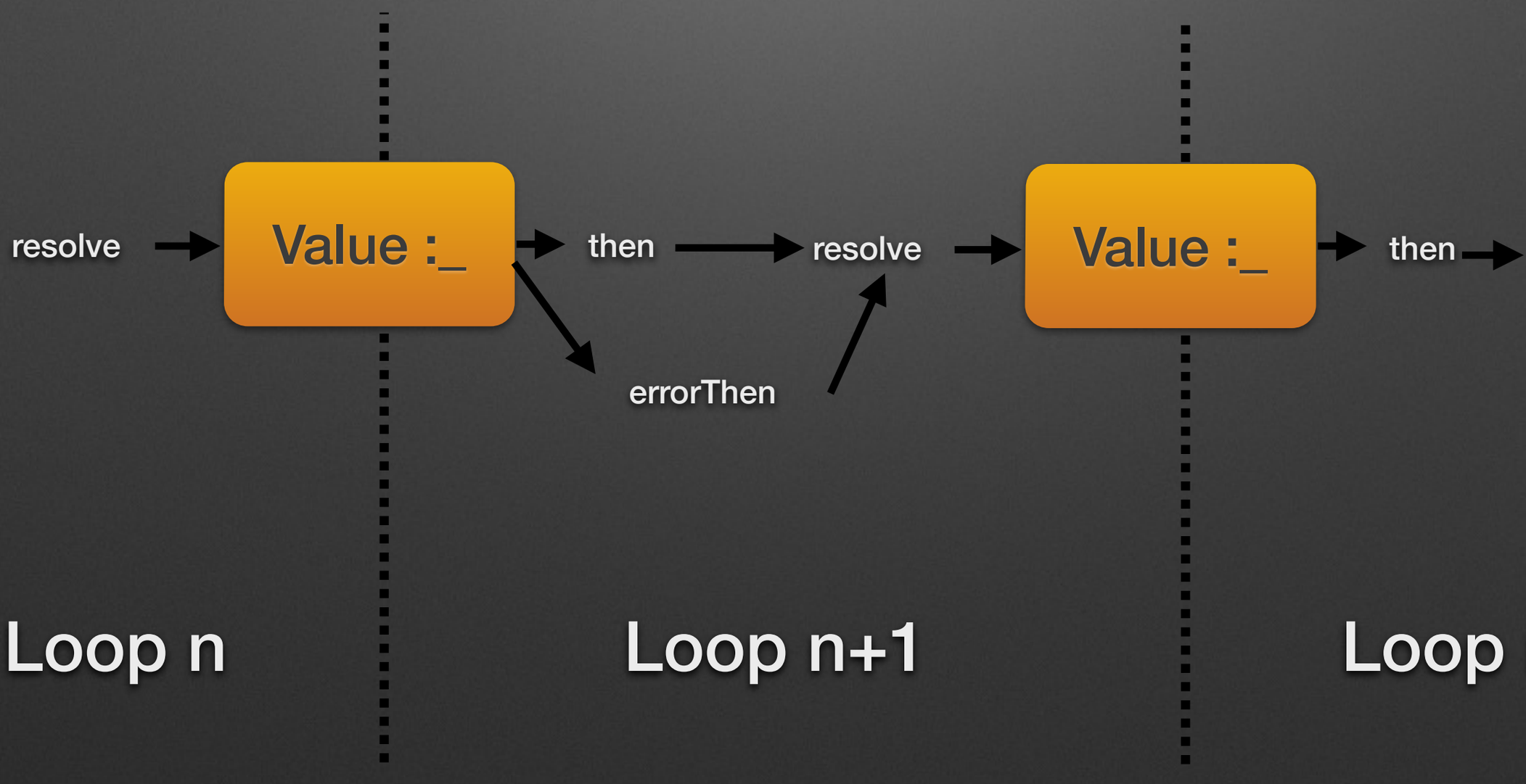
1. tmux

0 jjd 1 : vim | Sat Apr 12 09:41:26 PDT 2014

```
1  class Demo {
2     static function main() {
3        // Errors are propagated through the promise chain.
4        // You can rethrow errors to use Haxe's try/catch feature.
5        // Stream works the same here too.
6        Promise.when(p1,p2).then(function(x,y) {throw('an error'); return 'hi';})
7           .errorThen(function(x) {
8              return 'hi';
9           }).then(function(x) return 'a value')
10    }
11 }
```

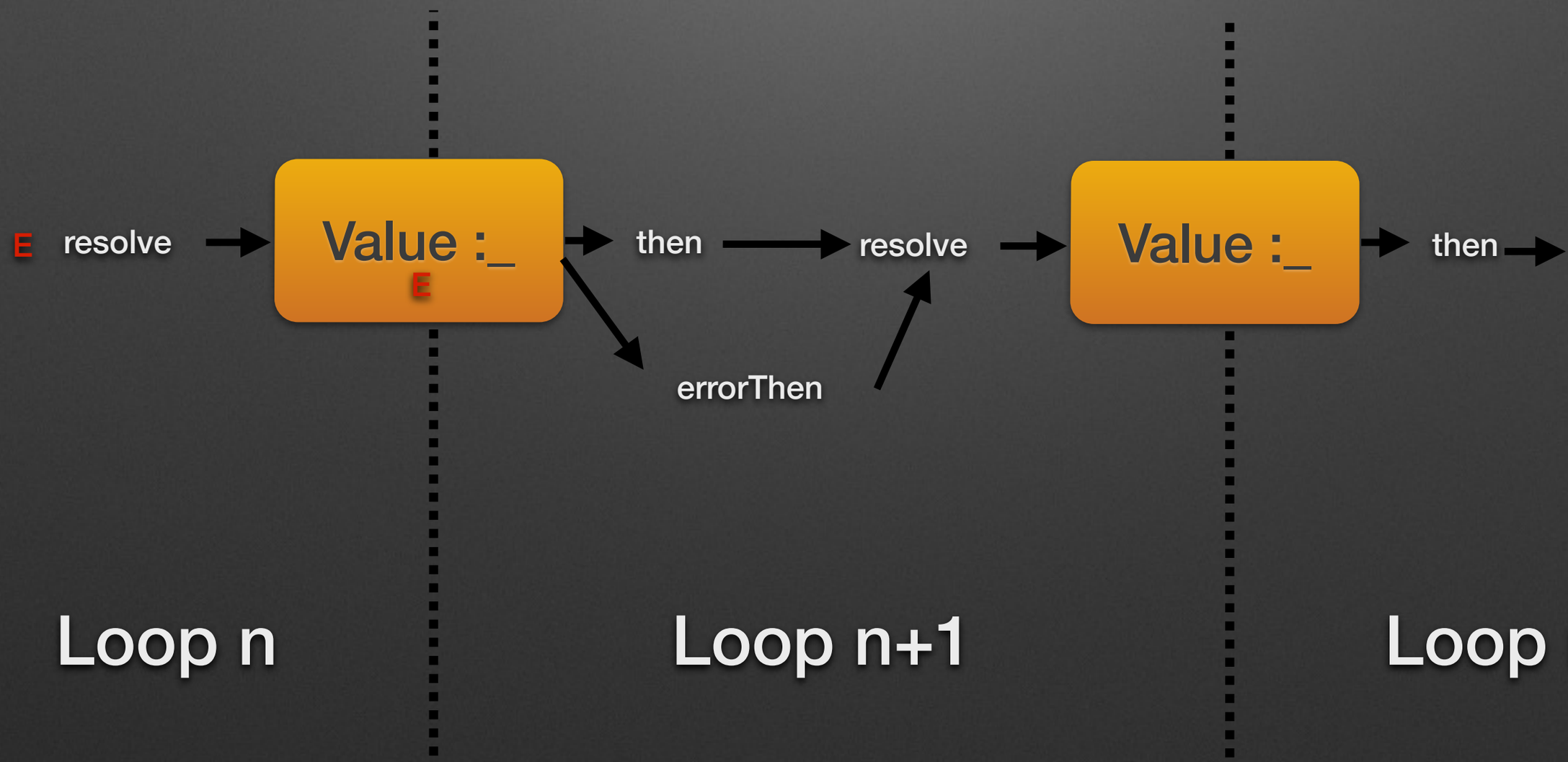
Promhx Object

Recovering from an error



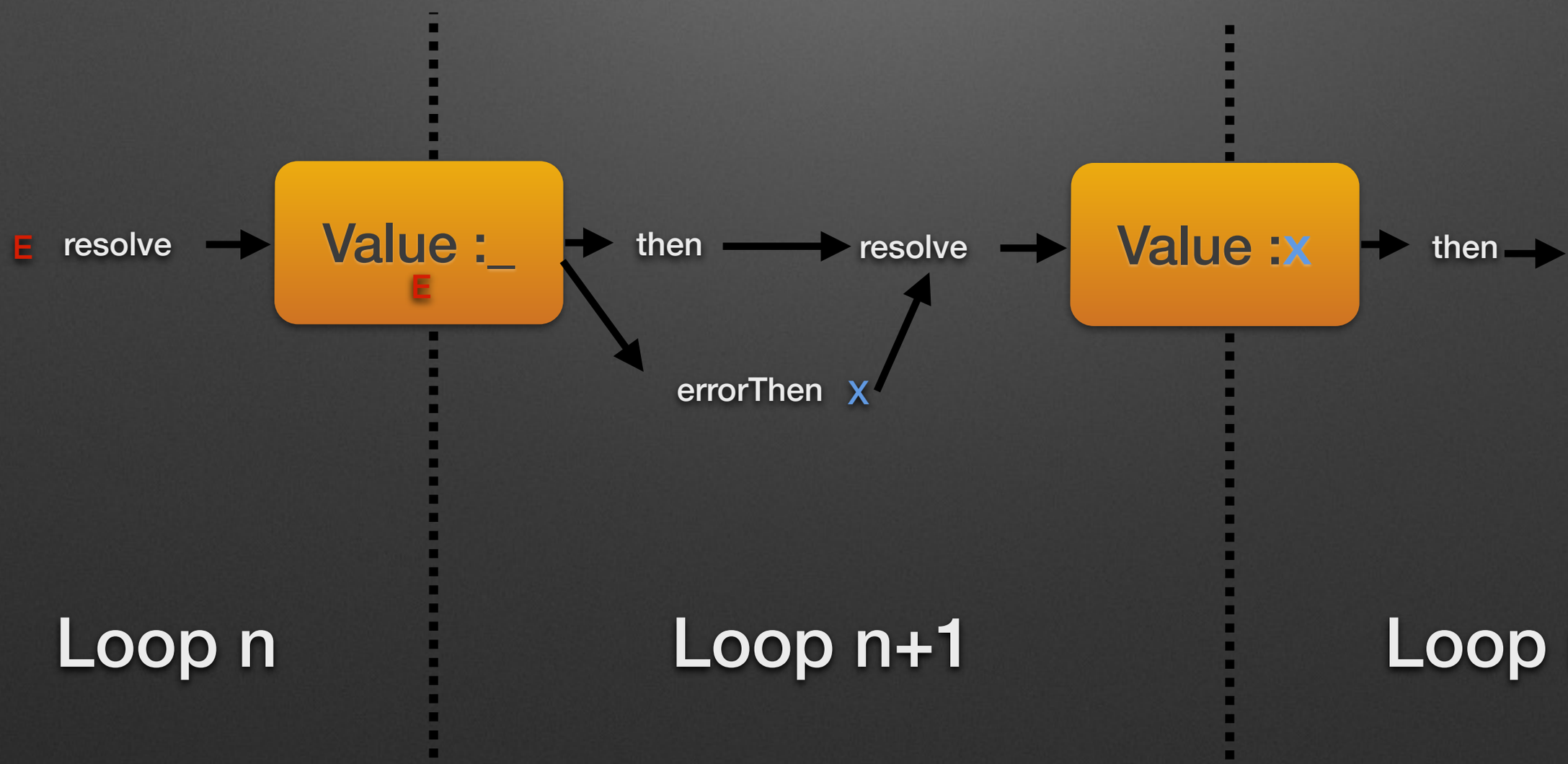
Promhx Object

Recovering from an error



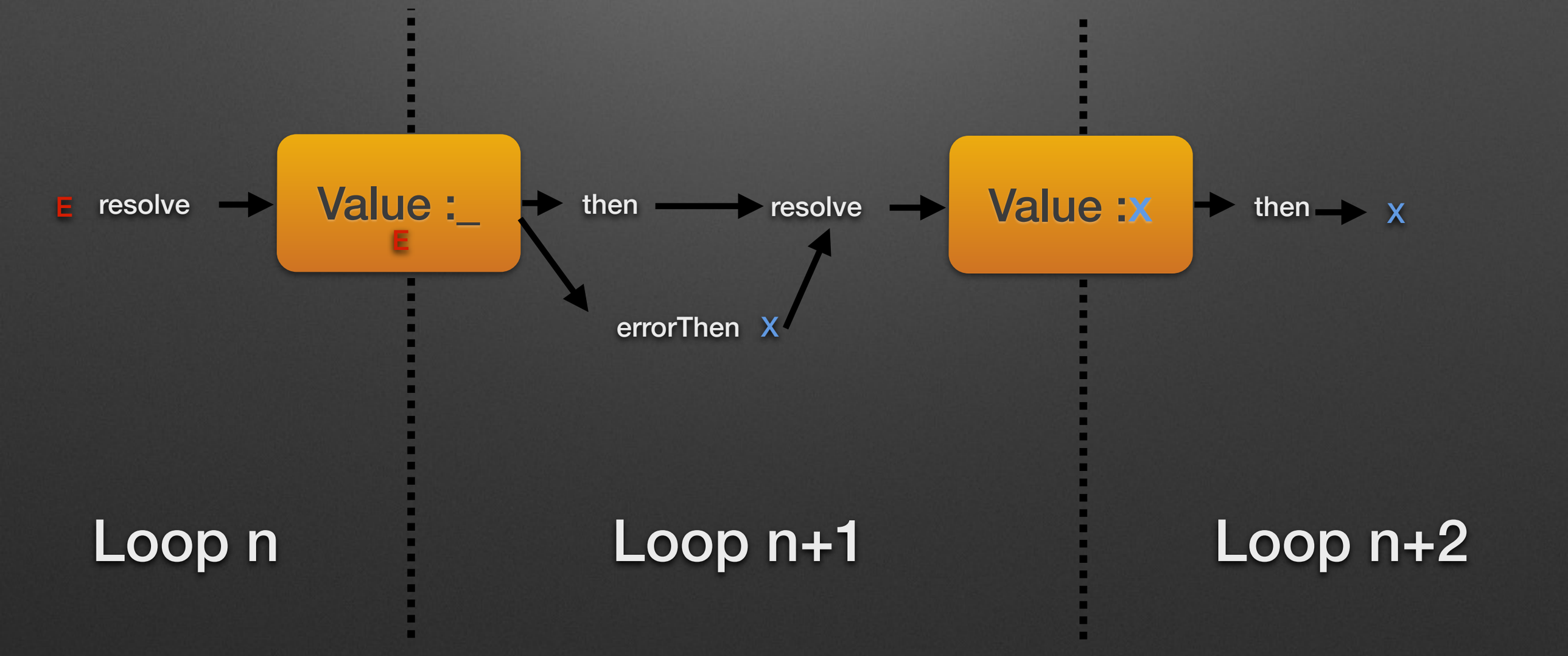
Promhx Object

Recovering from an error



Promhx Object

Recovering from an error



Are promhx objects Monads?

Three requirements for Monads:

- Type constructor : Create a monadic type for the underlying type. For example it defines type Promise<Int> for the underlying type Int. I.e. the class constructor
- unit function : Wraps a value of underlying type into a monad. For Promise monad it wraps value 2 of type number into value Promise(2) of type Promise<Int>. I.e. Promise.promise.
- bind function : Chains operations on monadic values. I.e. the “then” function.

…So yes, but why should we care?

Monads are useful

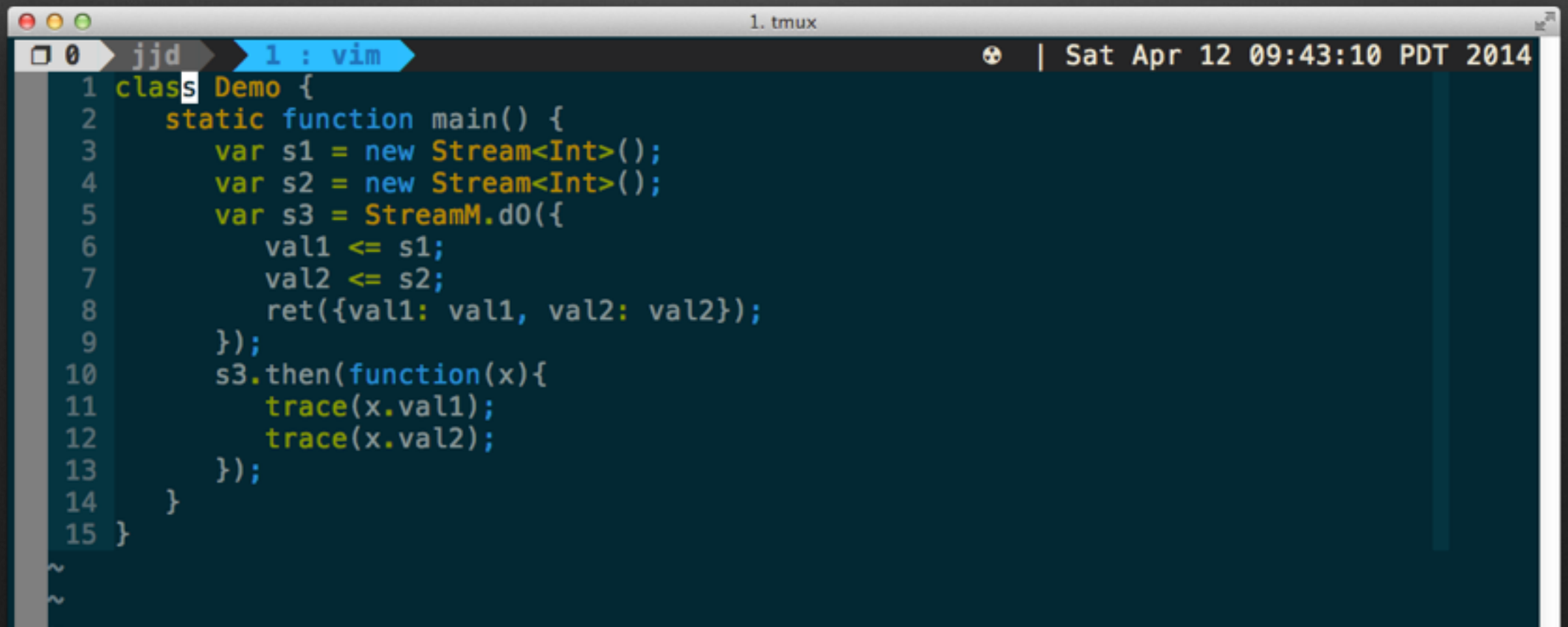
- Allows for new capabilities for a given type, and allows the original functions to use the new capabilities.

- Captures operations on types, making it easier to build complex compositions that all behave by the same simple rules.

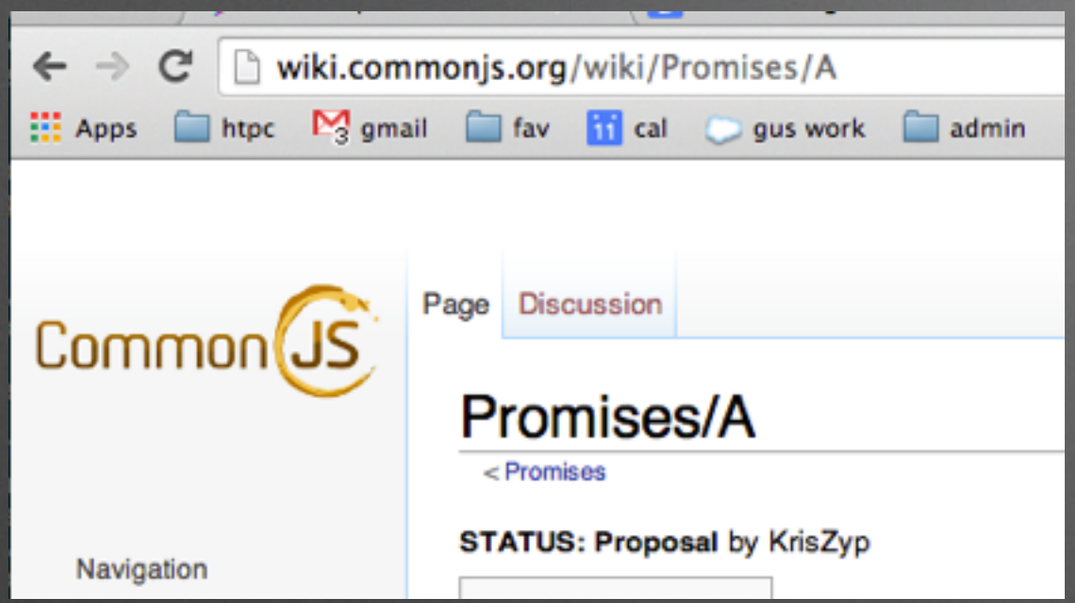
- Represents side-effecting operations cleanly in languages where this is problematic

Monad Composability

- Do-notation provided courtesy of Monax (Stephane Ledorze)
- The “<=“ operator binds the result returned via then()
- We can drop the chain syntax, and write out promises as if they were a simple series of expressions.



Composability vs. Standards



- JS: callback usage is predominant, especially in platforms like node.
- Should we follow standards? Standards are good right?

Promises A+ Spec

Promise<Promise<T>>?

2.3.1. If `promise` and `x` refer to the same object, reject `promise` with a `TypeError` as the reason.

2.3.2. If `x` is a promise, adopt its state [[3.4](#notes)]:

- 2.3.2.1. If `x` is pending, `promise` must remain pending until `x` is fulfilled or rejected.
- 2.3.2.2. If/when `x` is fulfilled, fulfill `promise` with the same value.
- 2.3.2.3. If/when `x` is rejected, reject `promise` with the same reason.

2.3.3. Otherwise, if `x` is an object or function,

- 2.3.3.1. Let `then` be `x.then`. [[3.5](#notes)]
- 2.3.3.2. If retrieving the property `x.then` results in a thrown exception `e`, reject `promise` with `e` as the reason.
- 2.3.3.3. If `then` is a function, call it with `x` as `this`, first argument `resolvePromise`, and second argument `rejectPromise`, where:
  - 2.3.3.3.1. If/when `resolvePromise` is called with a value `y`, run `[[Resolve]](promise, y)`.
  - 2.3.3.3.2. If/when `rejectPromise` is called with a reason `r`, reject `promise` with `r`.
  - 2.3.3.3.3. If both `resolvePromise` and `rejectPromise` are called, or multiple calls to the same argument are made, the first call takes precedence, and any further calls are ignored.
  - 2.3.3.3.4. If calling `then` throws an exception `e`,
    - 2.3.3.3.4.1. If `resolvePromise` or `rejectPromise` have been called, ignore it.
    - 2.3.3.3.4.2. Otherwise, reject `promise` with `e` as the reason.
- 2.3.3.4. If `then` is not a function, fulfill `promise` with `x`.

2.3.4. If `x` is not an object or function, fulfill `promise` with `x`.

~~pipe/flatMap~~

~~then /map~~

No composability

Well Tested

jdonaldson/promhx

build passing

A promise and functional reactive programming library for Haxe

Current | Build History | Pull Requests | Branch Summary

master - add some haxedoc for deferred

#106 passed

ran for 4 min 23 sec
2 days ago

Justin Donaldson authored and committed

Commit 652efd8 Compare 0226d01..652efd8

Build Matrix

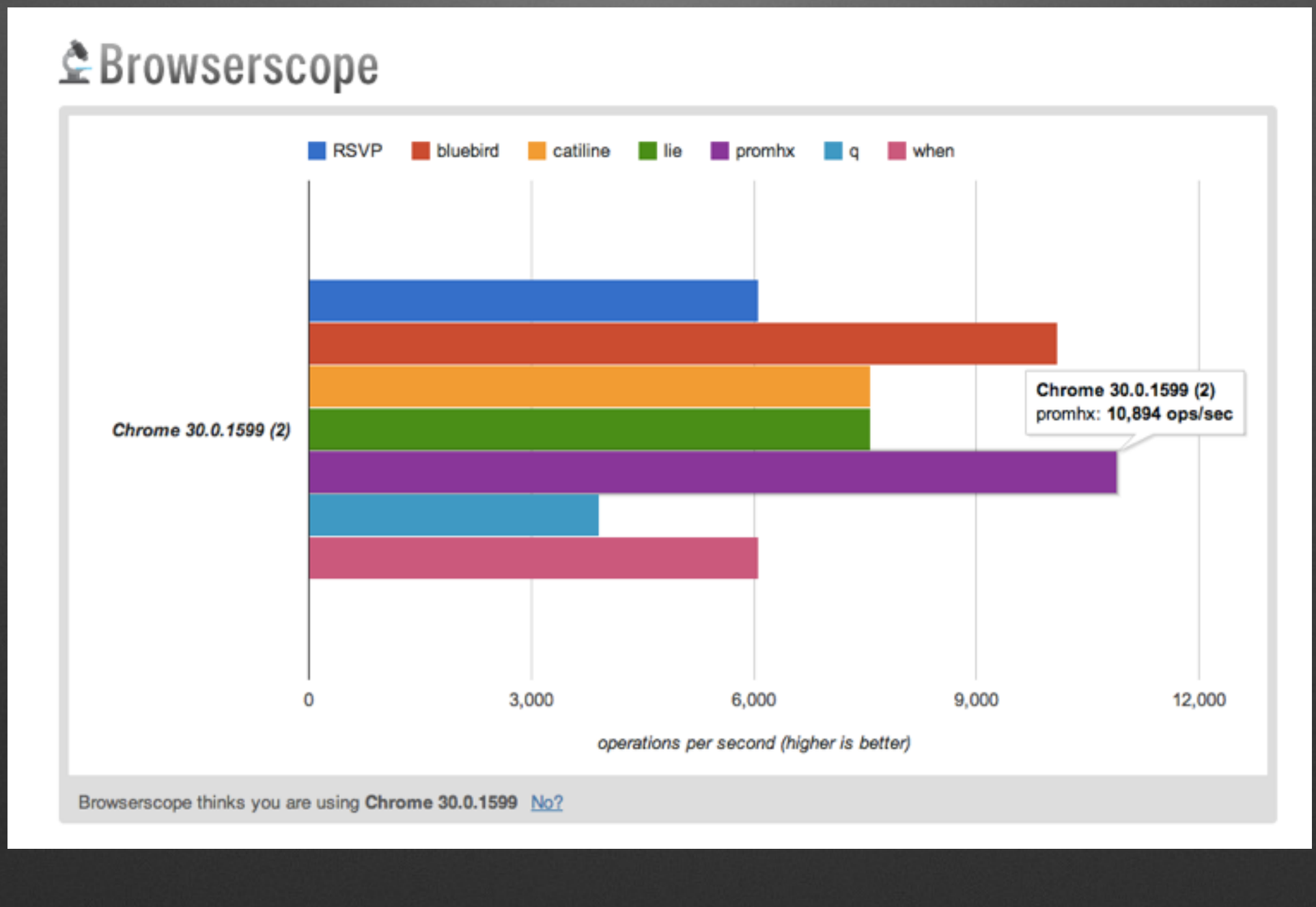
Job	Duration	Finished	ENV
106.1	1 min 14 sec	2 days ago	TARGET=cpp
106.3	47 sec	2 days ago	TARGET=java
106.4	40 sec	2 days ago	TARGET=neko
106.5	33 sec	2 days ago	TARGET=node

Allowed Failures

Job	Duration	Finished	ENV
106.2	31 sec	2 days ago	TARGET=cs
106.6	38 sec	2 days ago	TARGET=php

~30 f tests
x 7 platforms
(2 have problems)

Promhx Speed



That's it!

- I talked about why we need better asynchronous programming methods
- I talked about the basics of functional reactive programming
- I talked about the basic types of FRP that promhx supports, Promise, Stream, and PublicStream. I also talked about what each one is designed for, and how they use Deferred where appropriate.
- I talked about the way promhx handles errors and loop behavior
- I talked about some cool tricks that promhx can do since it behaves as a monad
- I talked about how fast and well tested the library is

QUESTIONS

?

We're Hiring!

- UI Dev Positions - Search and other teams
- Offices in Paris/Grenoble/Seattle/(etc.)
  (San Francisco HQ)
- Not much Haxe… yet!

AURA