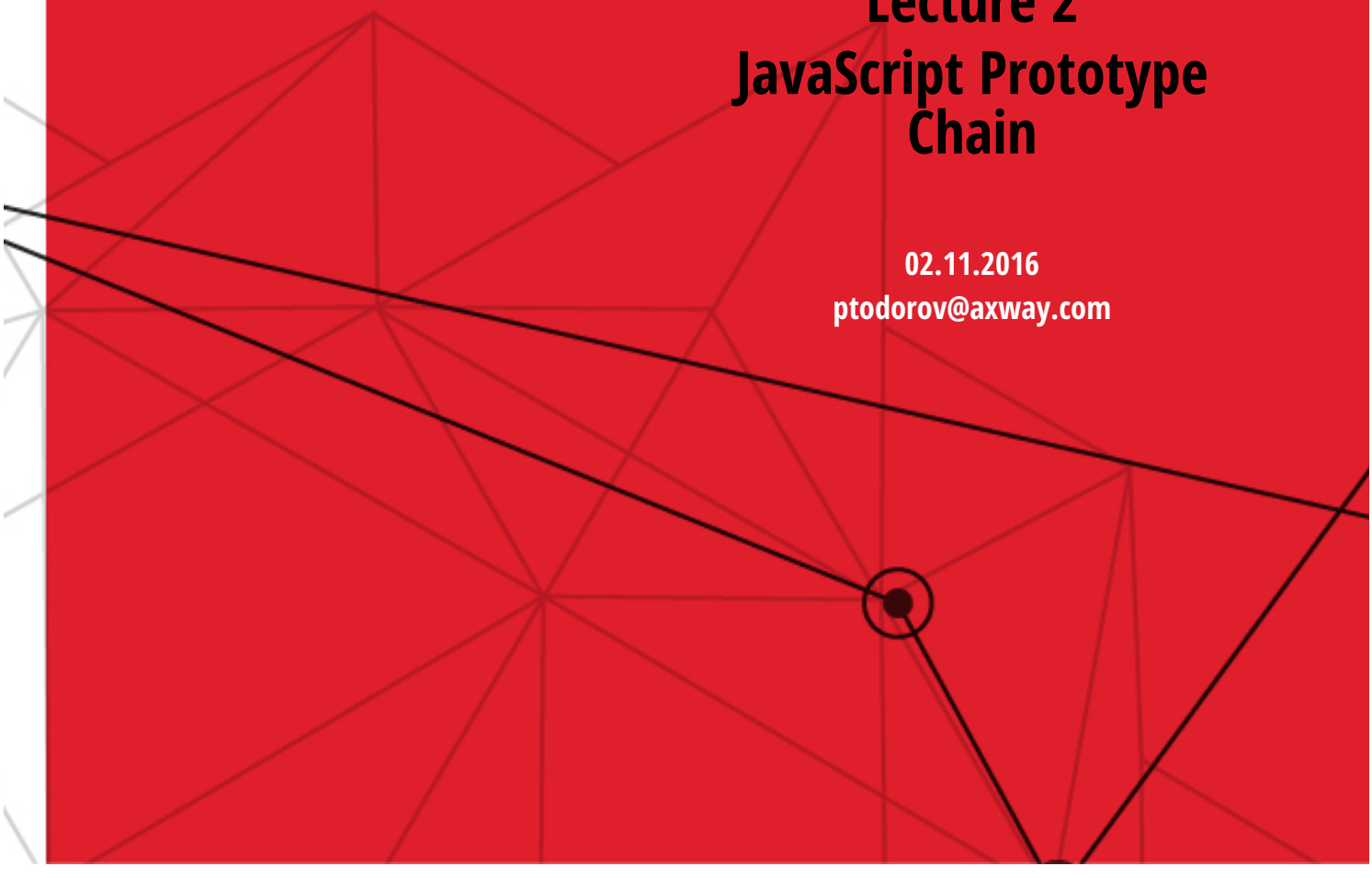
TRAINING

JAVASCRIPT
NODEJS
ARROW

Lecture 2
JavaScript Prototype Chain

02.11.2016
ptodorov@axway.com



QUIZ 1

```
debug(p); // ?

var p = 5;

debug(p); // ?

var p = 6;

debug(p); // ?
```

QUIZ 2

```
debug(typeof x); // ?

var x = 10;

debug(x); // ?

x = 20;

function x() {};

debug(x); //?
```

IN BOTH CASES 'P' DOES NOT EXIST BUT WE HAVE DIFFERENT RESULT. WHY?

- The engine resolves identifiers using internal method GetValue (p) -> if IsUnresolvableReference(p) throw a ReferenceError exception
- In case of 'typeof'' the engine runs special algorithm -> if IsUnresolvableReference(p) return "undefined"
- **Bottom Line: Consider both 1) ECMAScript Specification AND 2) JS Engine specific augmentations**

JavaScript is
1) untyped
2) interpreted
3) dynamic
4) multi-paradigm
programming language
standardized by
ECMAScript specification.

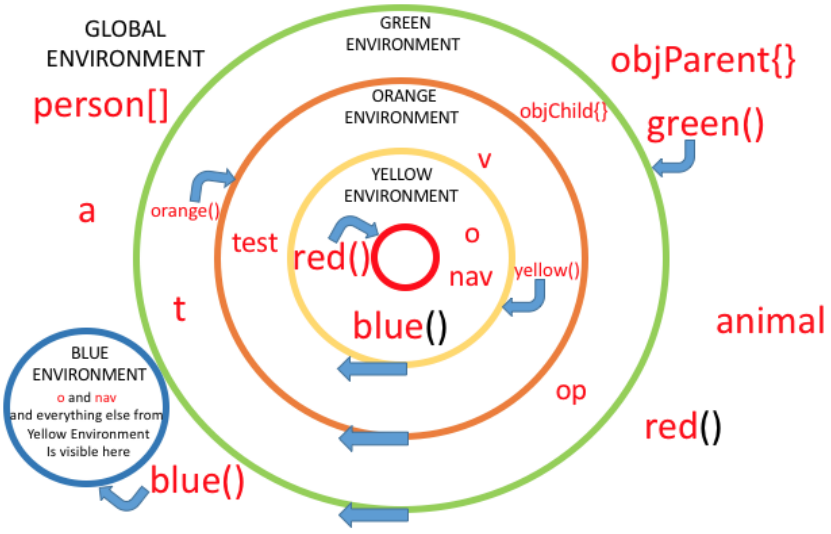
- Wikipedia

```
var a = 5; //global env
// we see only a here

function foo() {
    var a = 6; //foo env
    var b = 10; //foo env
    // we see a and b here
    // a is now 6

    function bar() {
        var c = 15; //bar env
        // we see a (=6), b, and c here
    }

    bar(); // function activation triggers
           // bar environment creation

}
// we see only a here

foo(); // function activation triggers
       // foo environment creation
```

JAVA SCRIPT PROGRAM

=

STACK OF ENVIRONMENTS


GLOBAL
ENVIRONMENT
person[]
a
t
BLUE
ENVIRONMENT
o and nav
and everything else from
Yellow Environment
Is visible here
blue()
GREEN
ENVIRONMENT
objParent{}
green()
ORANGE
ENVIRONMENT
objChild{}
orange()
test
YELLOW
ENVIRONMENT
v
o
red()
nav
yellow()
blue()
animal
op
red()

LECTURE 1 TAKEWAYS (1/2)

- One Global and Many Local Enviroments with Identifiers
- Identifiers (states: NOT EXIST / NOT BOUND / BOUND, creation: Variable or Function Declaration / Function Parameters, types: built-in / manual
- Identifier to Value Binding-> Function Declaration during Environment Creation (Hoisting) / Function Expression and Variable Declaration during Environment Execution

LECTURE 1 TAKEWAYS (2/2)

- Values - Everything is Object Mantra: primitives are autoboxed (5 types, 7 falsy values, coercion) / functions (callable objects)
- Functions (the most powerful type of objects): Create Local Environments (on invokation) / Send data to other environments (closures) / Create Other Objects (when invoked with new)
- How do we create Function objects? -> Function Declaration or Expression / Function constructor
- How do we resolve values? -> Scope Chain Resolution

Using only Scope Chain Resolution to write apps in

1) Functional programming style

2) WITHOUT rigid object hierarchies

3) WITH single source of truth data store,

4) Immutable data structures, and

4) Streams-like control flow

weaving different single purpose granular functions into coherent application logic

- The True Nature of JavaScript

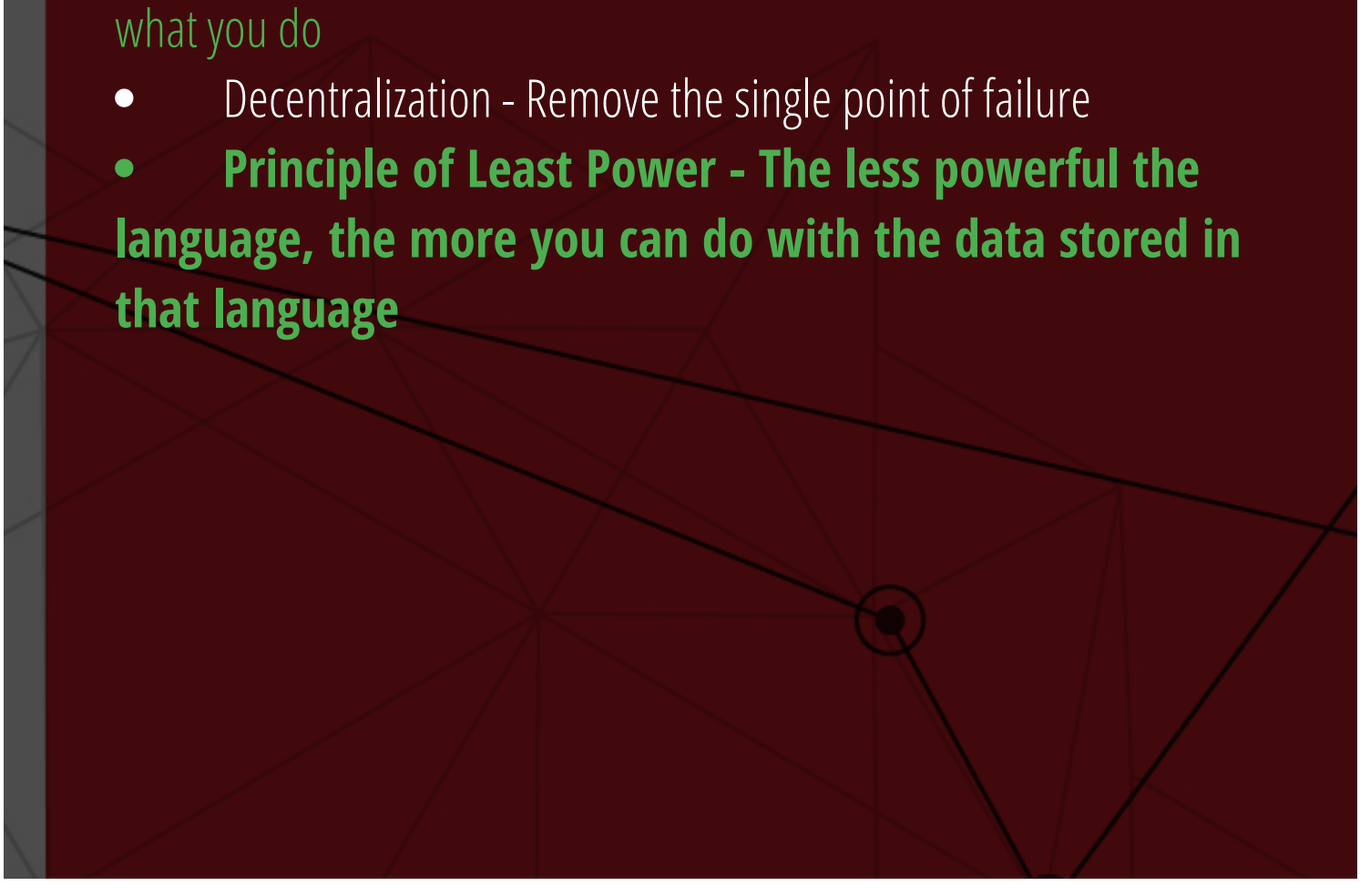
SO HOW WE ARE GOING TO PROCEED TODAY?

- Your Daily Dose Of Motivation
- MAIN GOAL: Understand Prototype Chain (in contrast to Scope Chain it allow us writing in OOP style, deal with object hierarchies and 'this' context etc.)
- OOP (in contrast to Functional Programming)
- Useful JavaScript Patterns (good reason to weave some software architecture advices in this lecture; as much as we can...will continue with them in Modules lecture)

PRINCIPLES OF SOFTWARE/SYSTEMS DESIGN

Tim Berners-Lee

- Simplicity - Keep it simple, stupid!
- Modularity - Granular reusable software agnostic to change
- Tolerance - Be liberal in what you require but conservative in what you do
- Decentralization - Remove the single point of failure
- **Principle of Least Power - The less powerful the language, the more you can do with the data stored in that language**



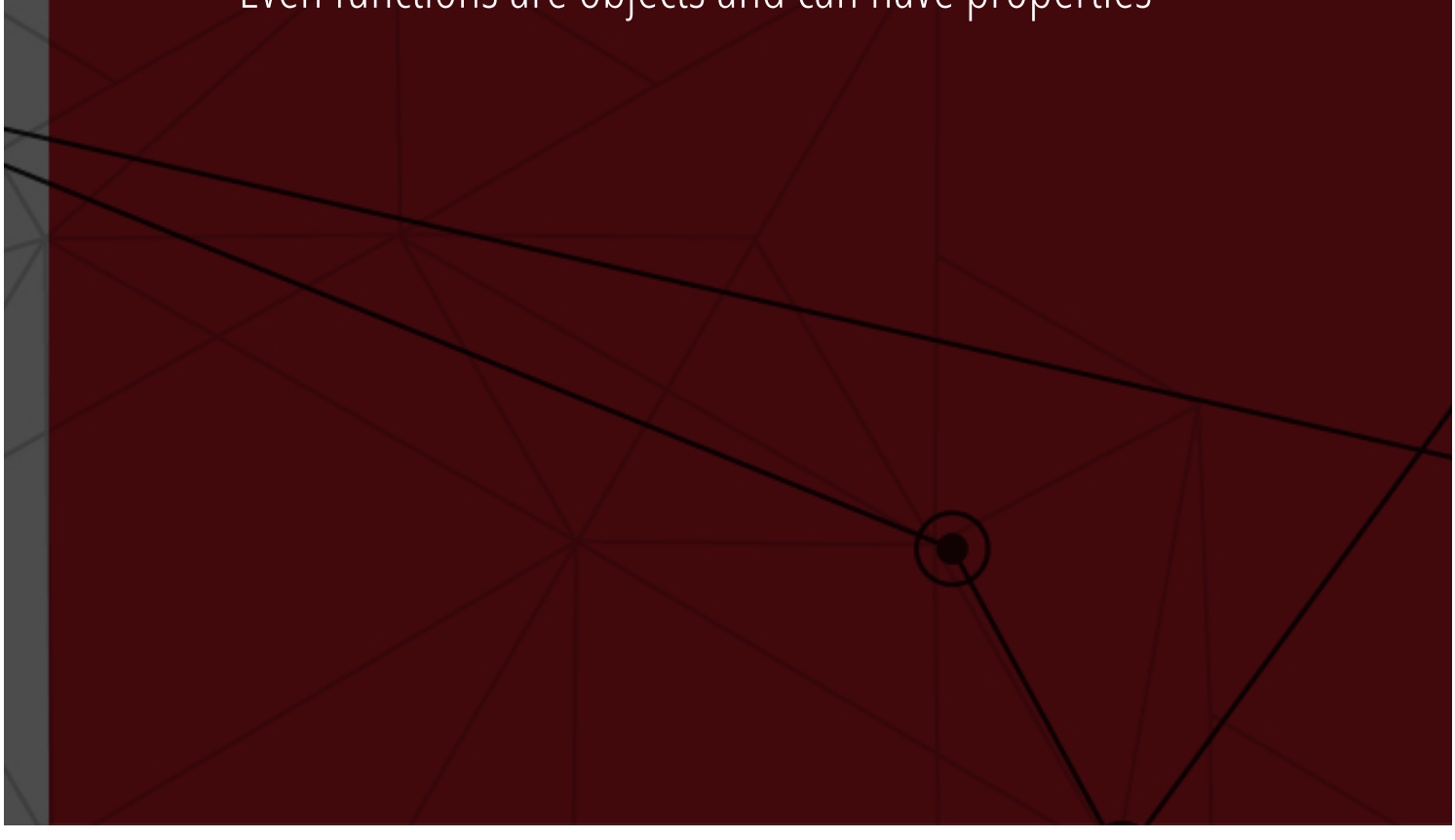
Atwood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript.

- 2007, Jeff Atwood (founder of stackoverflow)

1. EVERYTHING IS OBJECT

everything-is-object.js

- Primitives are autoboxed
- Even functions are objects and can have properties



2. HOW DO WE CREATE REAL OBJECTS?

object-creation.js

- Object Literal
- Object.create
- Constructor Functions: built-in for reference types
- Constructor Functions: custom

3. FUNCTIONS ROLES - SUMMARY

function-roles.js

- Function used as Function => test()
- Function used as Method => obj.test()
- Function used as Constructor => new test()
- Constructor Functions are implicit - protect your code of cases where 'new' is missing
- Functions are Objects - they can have properties and methods

4. IDENTIFIERS VS. PROPERTIES

properties.js

- Identifiers are evaluated via Scope Chain
- Properties are evaluated via Prototype Chain
- Properties are deletable by default

5. OBJECT MODIFICATION

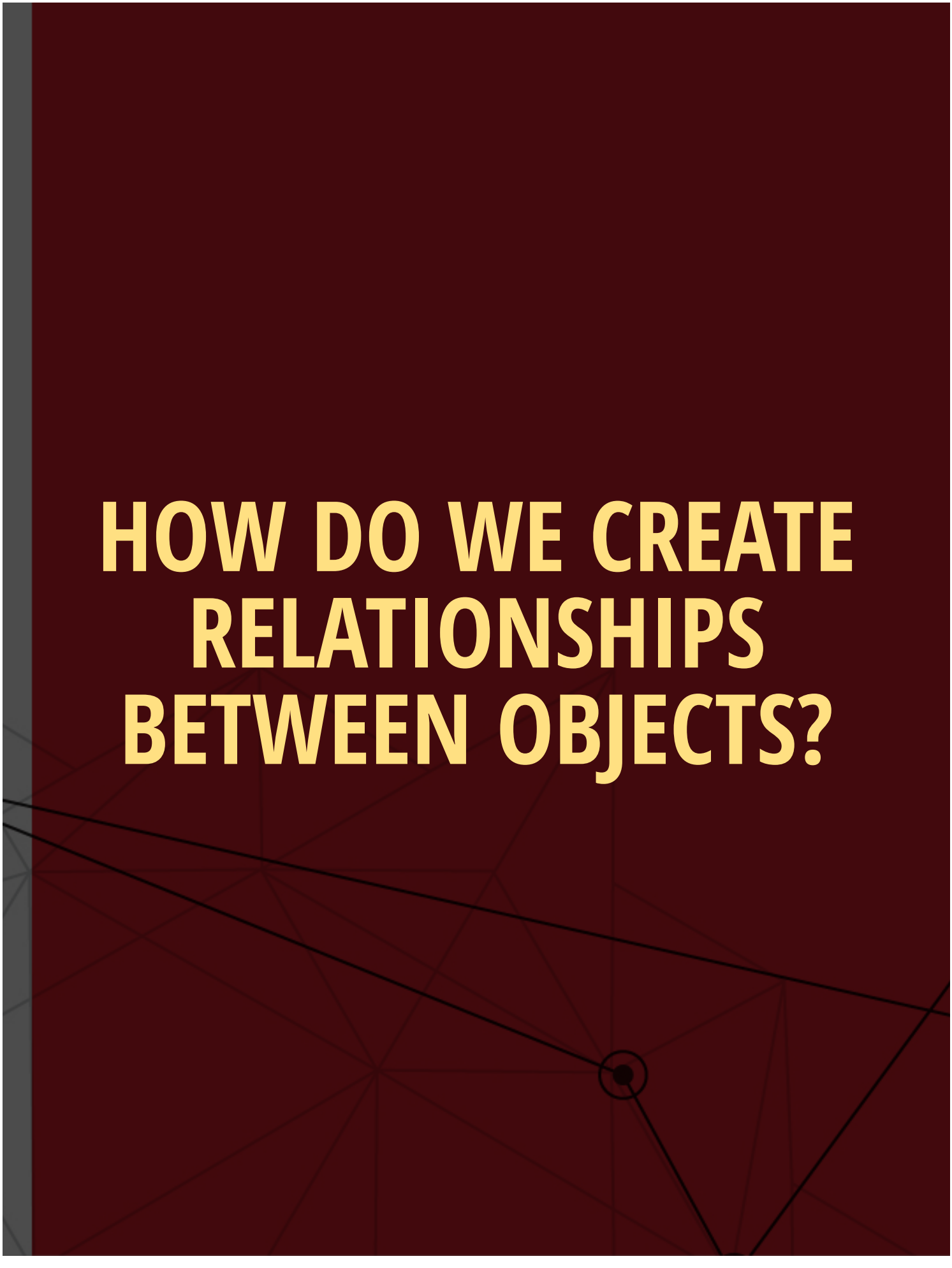
object-modification.js

- Remove the possibility to add properties - isExtensible
- Remove the possibility to add and remove properties - isSealed
- Remove the possibility to add, remove, and update properties - isFrozen

OUR PROGRAMS SO FAR

1. A SET OF ENVIRONMENTS FULL OF UNRELATED OBJECTS
2. SOME OBJECTS ARE SIMPLE DATA HOLDERS WHERE DATA VISIBILITY OBEY SCOPE CHAIN RULES
3. OTHER OBJECTS ARE CALLABLE - COULD RECIEVE DATA, DO SOME PROCESSING, AND PASS DATA ALONG

HOW DO WE CREATE RELATIONSHIPS BETWEEN OBJECTS?



HELLO, OBJECT ORIENTED PROGRAMMING!

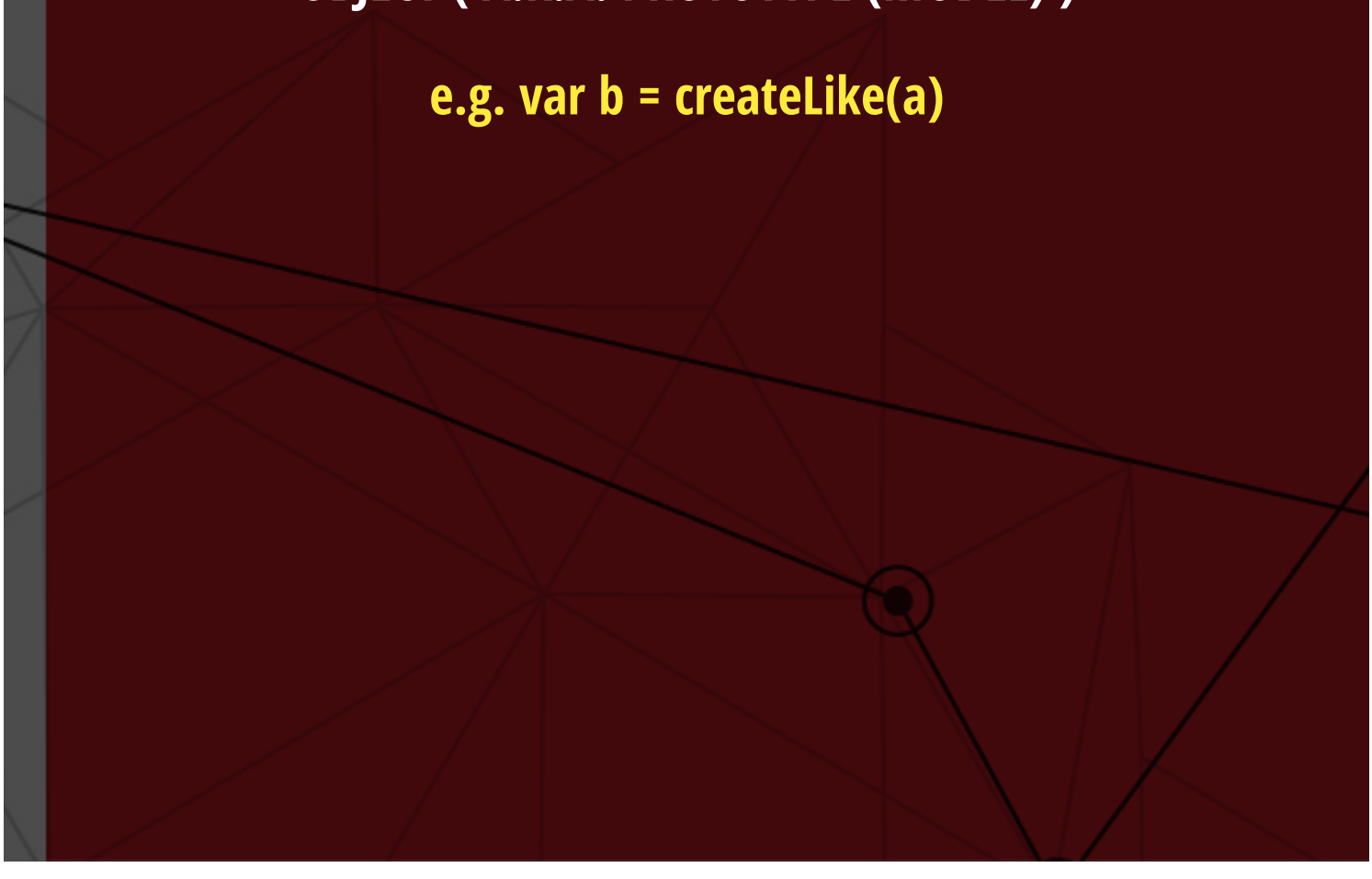
CLASSICAL OOP

1) CREATE CLASS AS A BLUEPRINT
2) ONE CLASS COULD INHERIT ANOTHER
3) CREATE OBJECT INSTANCES OUT OF THE CLASS

PROTOTYPAL OOP

CREATE OBJECT BASED ON (INHERITING) ANOTHER OBJECT (A.K.A. PROTOTYPE (MODEL))

e.g. var b = createLike(a)



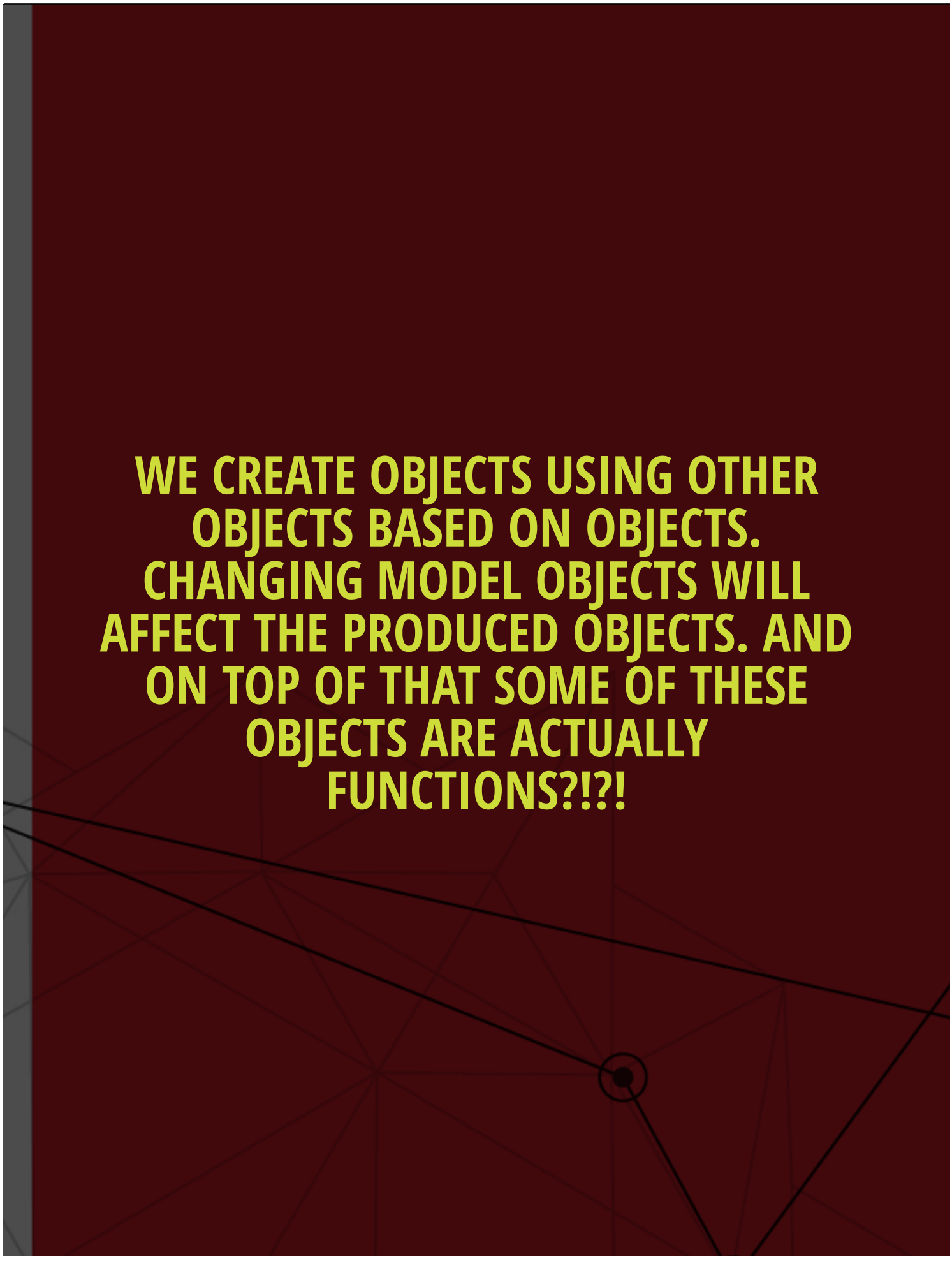
6. HOW DO WE CREATE REAL OBJECTS (REVISITED)

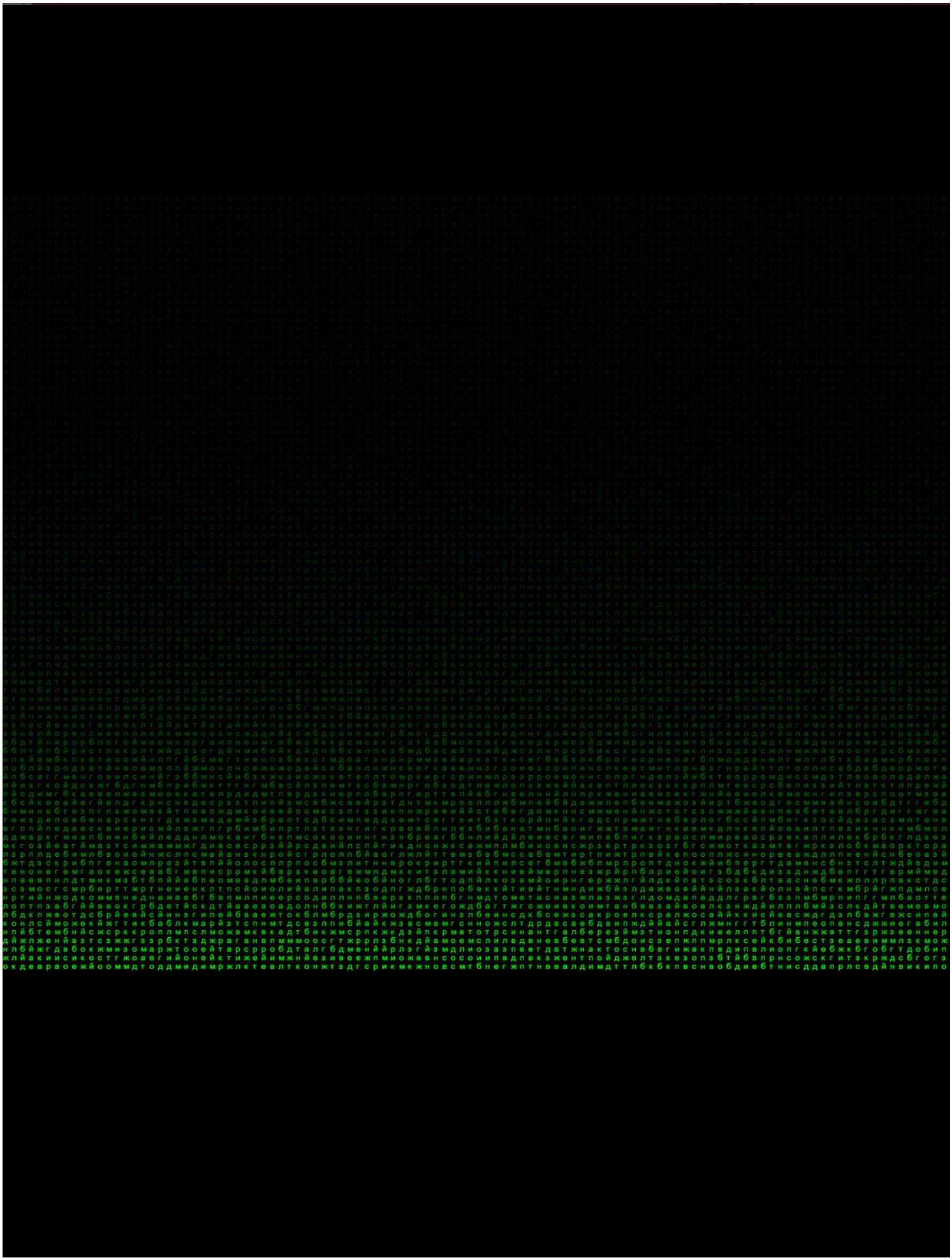
object-creation-default-proto.js

- new Object() => default Model === Object.prototype
- Object literal => default Model === Object.prototype
- Object.create => no default Model, must be provided

Lets bring some confussion here...

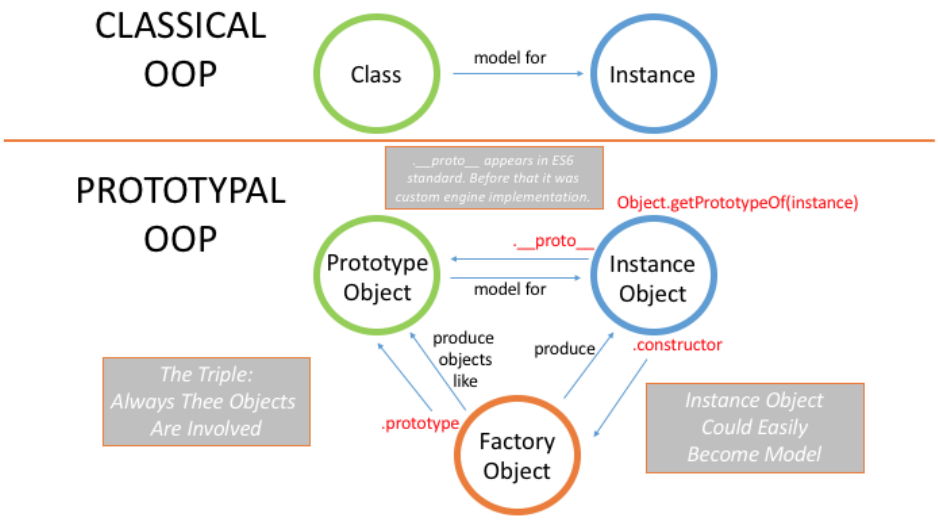
```
1. test('### Confusion ###', function(t) {
2.
3.     var obj = new Array();
4.     t.ok(typeof Array === "function");
5.     t.ok(Array instanceof Function);
6.
7.     t.ok(Object.getPrototypeOf(obj) === Array.prototype);
8.     t.ok(Array.prototype instanceof Object);
9.
10.    Array.data = 5;
11.    t.equal(5, Array.data);
12.
13.    t.ok(Object.getPrototypeOf(Array) ===
Function.prototype);
```


WE CREATE OBJECTS USING OTHER OBJECTS BASED ON OBJECTS. CHANGING MODEL OBJECTS WILL AFFECT THE PRODUCED OBJECTS. AND ON TOP OF THAT SOME OF THESE OBJECTS ARE ACTUALLY FUNCTIONS?!?!



SO WHO IS THE ARCHITECT?

Object.prototoype


CLASSICAL
OOP
Class
model for
Instance
PROTOTYPAL
OOP
.__proto__ appears in ES6 standard. Before that it was custom engine implementation.
Object.getPrototypeOf(instance)
Prototype Object
.__proto__
model for
Instance Object
produce objects like
produce
.constructor
The Triple: Always Thee Objects Are Involved
.prototype
Factory Object
Instance Object Could Easily Become Model

7. PROTOTYPAL INHERITANCE - THE TREE OBJECTS

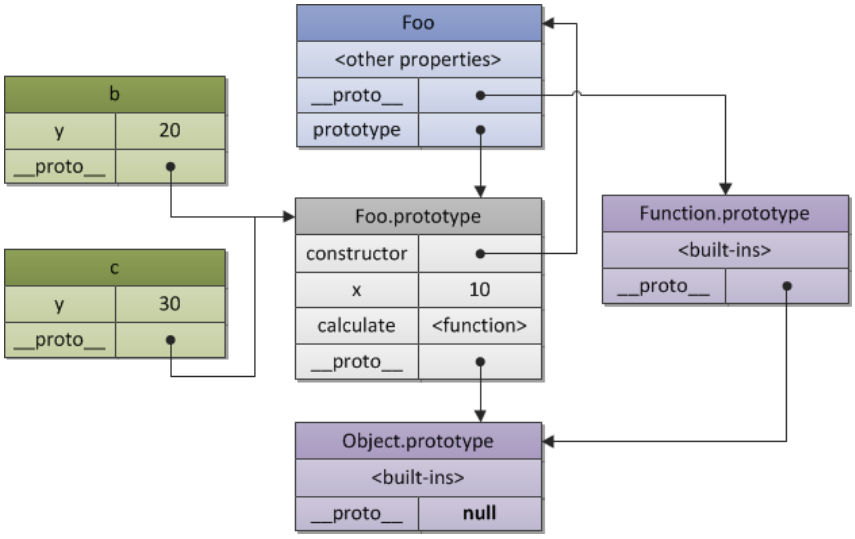
model-factory-instance.js

- Instance Objects (IO) - created with Constructor Function, Object Literal, Object.create
- Factory Object (FO) - accessed with IO.constructor
- Prototype(Model) Object => accessed with IO.__proto__ and FO.prototype
- NOTE: If we change __proto__ or prototype we will break the hierarchy unless we synchronize both of them again

8. OBJECT CREATION EVOLUTION: ES3, ES5

real-world.js

- ES3: SHIM for Prototypal inheritance - difficult but ok if you have it in library
- ES5: Native Prototypal inheritance with Object.create



Source: http://dmitrysoshnikov.com/ecmascript/javascript-the-core/

HONESTLY, IS THIS SYNTAX FAMILIAR TO YOU?

```
class Address {
    constructor(town, country) {
        this.town = town;
        this.country = country;
    }

    getTown() {
        return this.town;
    }

    getCountry() {
        return this.country;
    }
}

class ExtendedAddress extends Address {
    constructor(town, country, zipCode) {
        super(town, country)
        this.zipCode = zipCode;
    }

    getZipCode() {
        return this.zipCode;
    }
}

var extAddr = new ExtendedAddress('Chicago', 'US', '555');
```

HELLO, ECMASCRIPT 6!

9. OBJECT CREATION EVOLUTION: ES6 AND BEYOND

real-world.js

- ES6: Native Classical inheritance with "class" construct - Beware of huge amount of syntactic sugar. Prototypal inheritance is implementation detail.
- Crockford Object Creational Pattern: Neither of above. It use the power of Node modules and closures to create and export immutable objects.

10. USING 'NEW' WITH CONSTRUCTOR FUNCTIONS - THE TWO OBJECTS

function-roles.js

- JS engine creates 1) empty object and set it to FO.prototype
- JS engine creates 2) instance object and set it to "this"

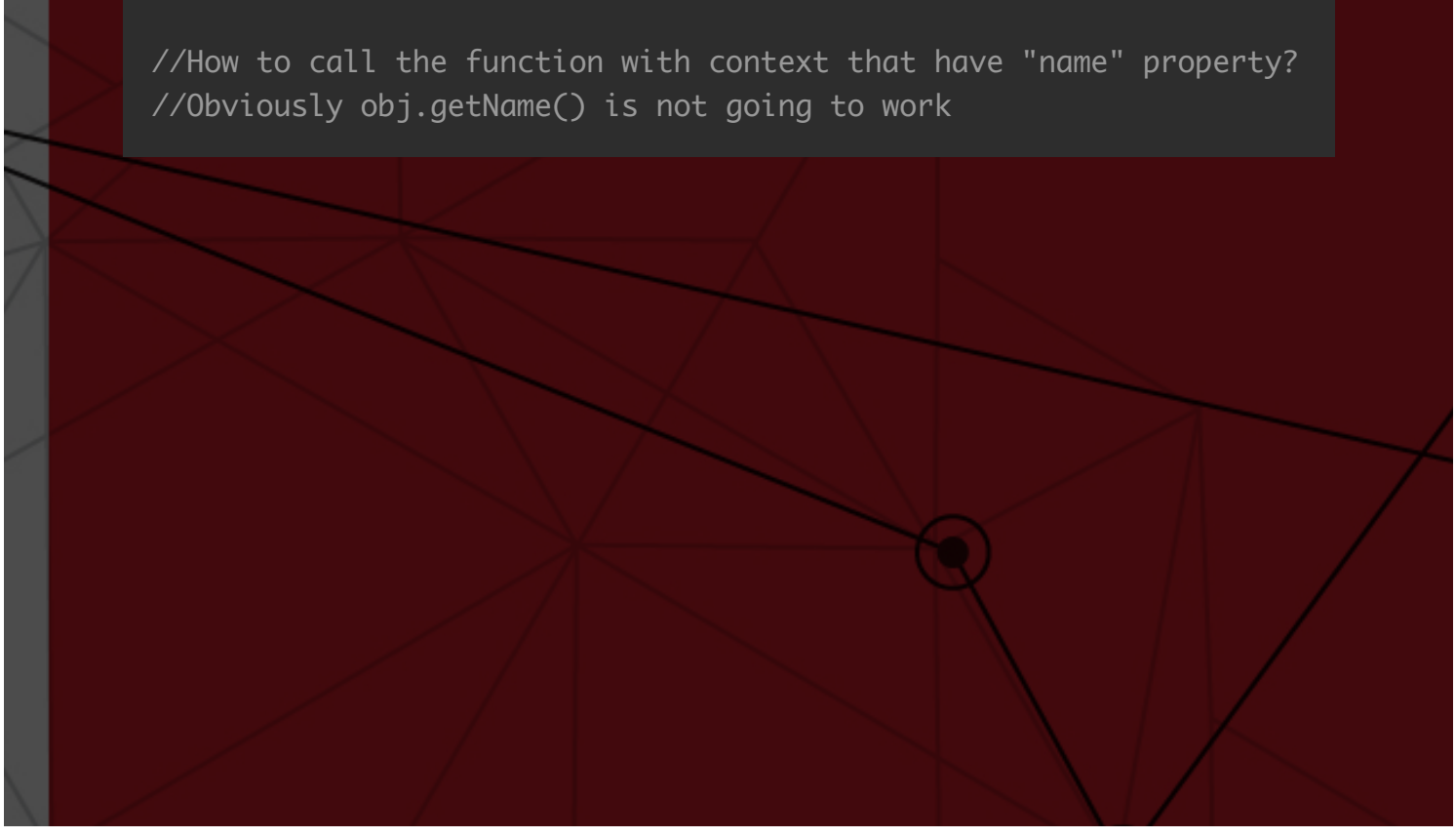
11. THE PROBLEM OF 'THIS'

this.js

- 'this' is a.k.a. the context in which the function is running
- the value of 'this' depends on the way the function is invoked
- to understand what is the value of 'this' look before the '.'

QUIZ

```
var obj = {
    name: "MJ"
};

function getName() {
    return this.name
}

//How to call the function with context that have "name" property?
//Obviously obj.getName() is not going to work
```



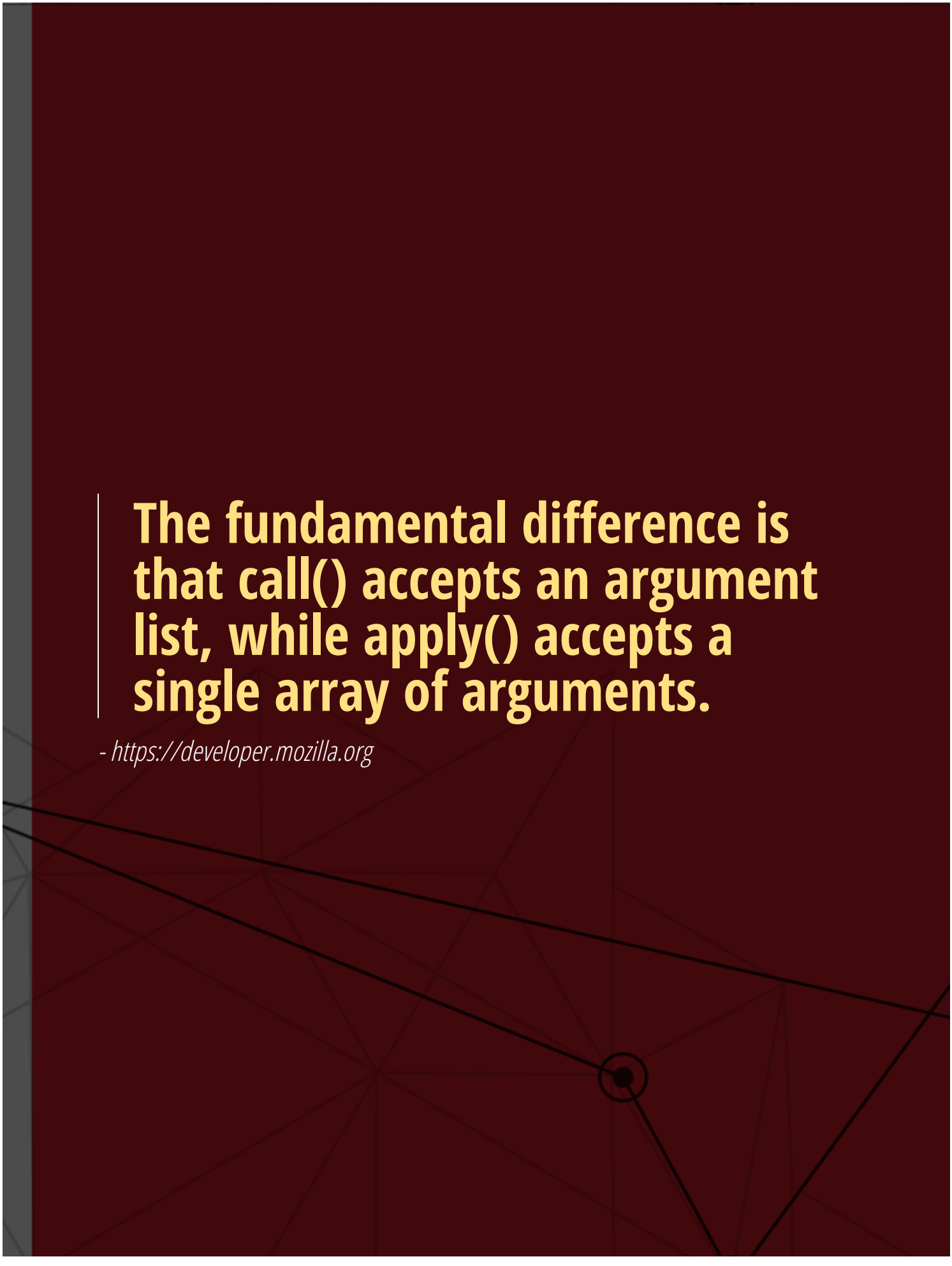
12. BINDING 'THIS' CONTEXT - BIND / CALL / APPLY

this.js

- 'this' is a.k.a. the context in which the function is running
- the value of 'this' depends on the way the function is invoked
- to understand what is the value of 'this' look before the '.'
- use bind / call / apply to control the 'this' context

The fundamental difference is that call() accepts an argument list, while apply() accepts a single array of arguments.

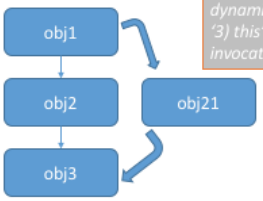
- https://developer.mozilla.org



SCOPE CHAIN RESOLUTION

Developers know the data just when scanning through the code

PROTOTYPE CHAIN RESOLUTION



1) Object hierarchies are rigid
2) Object hierarchies could change dynamically
'3) this' context is dynamic depends on invocation

4) call and bind everywhere in your code to ensure the right context

13. INHERITANCE SUMMARY

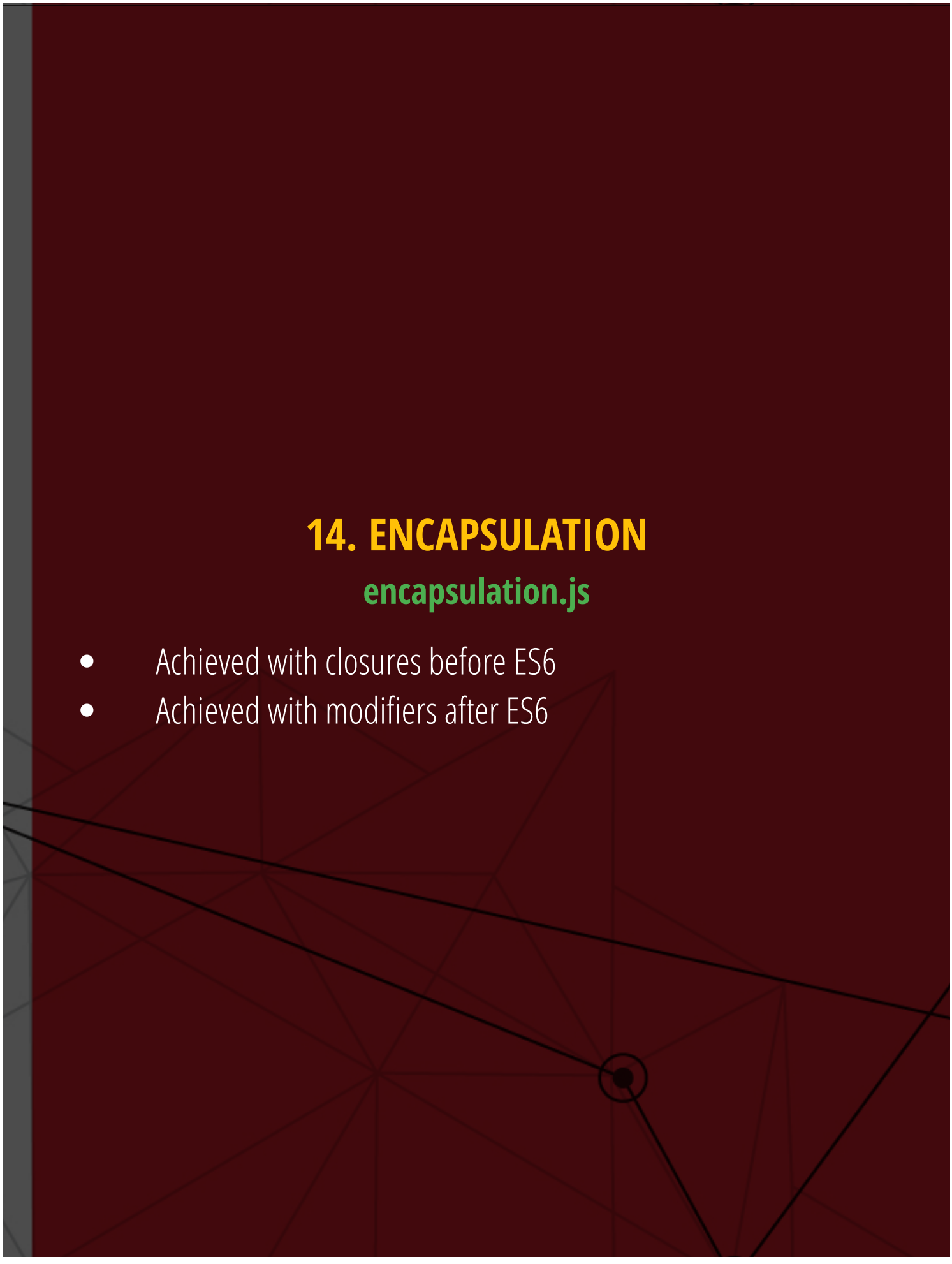
inheritance.js

- Classical: ES3/5 - Shim / ES6 - native support with Class construct
- Prototypal: ES3 - Shim / ES5 and beyond - native support with Object.create
- Pay attention to - __proto__ / prototype / constructor - properties

14. ENCAPSULATION

encapsulation.js

- Achieved with closures before ES6
- Achieved with modifiers after ES6



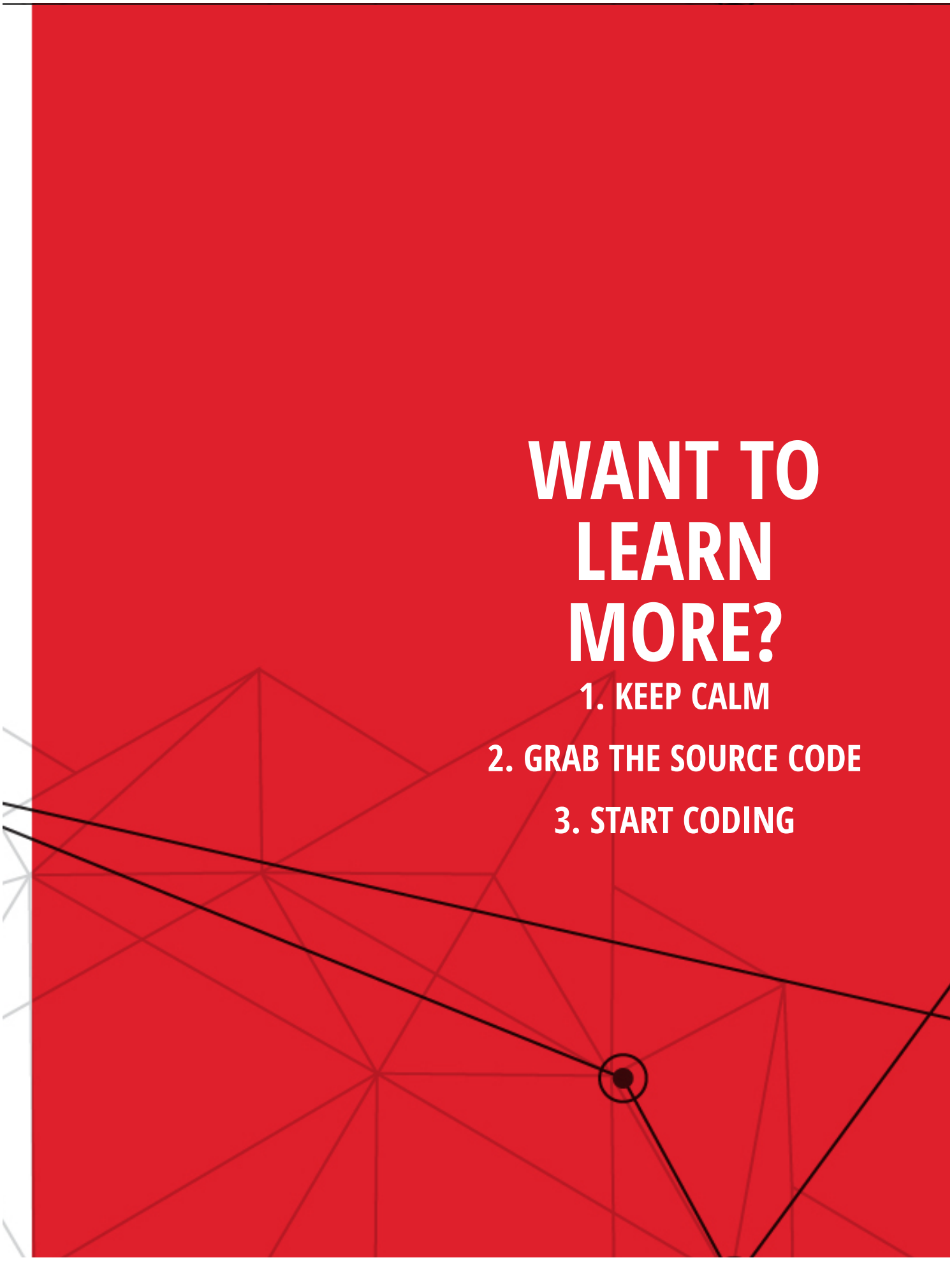
15. COMPOSITION AND MIXINS

this.js

- Shims before ES5
- Object.assign from ES5 and beyond

WANT TO LEARN MORE?

1. KEEP CALM
2. GRAB THE SOURCE CODE
3. START CODING



THANK YOU!

ptodorov@axway.com