KYLE SIMPSON GETIFY@GMAIL.COM

DEEP JS FOUNDATIONS

Motivations?

```
var x = 40;

x++;              // 40
x;                // 41

++x;              // 42
x;                // 42
```

```
x++;
++x;

// as...

x = x + 1;
```

```
var x = "5";
x = x + 1;    // "51"


var y = "5";

y++;          // ??
y;            // ??
```

```
var x = "5";
x = x + 1;    // "51"


var y = "5";

y++;          // 5
y;            // 6
```

Have you ever read any part of the JS specification?

12.4.4.1 Runtime Semantics: Evaluation

UpdateExpression **:** *LeftHandSideExpression* **++**

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be ? ToNumber(? GetValue(*lhs*)).
3. Let *newValue* be the result of adding the value 1 to *oldValue*, using the same rules as for the **+** operator (see 12.8.5).
4. Perform ? PutValue(*lhs*, *newValue*).
5. Return *oldValue*.

```
// x++ means:

function plusPlus(orig_x) {
    var orig_x_coerced = Number(orig_x);
    x = orig_x_coerced + 1;
    return orig_x_coerced;
}

var x = "5";
plusPlus(x);        // 5
x;                  // 6
```

https://twitter.com/YDKJS/status/1099716798088400899

Whenever there's a divergence between what your brain thinks is happening, and what the computer does, that's where bugs enter the code.

--getify's law #17

Course Overview

Types

- Primitive Types
- Abstract Operations
- Coercion
- Equality
- TypeScript, Flow, etc.

Scope

- Nested Scope
- Hoisting
- Closure
- Modules

Objects (Oriented)

- this
- class { }
- Prototypes
- OO vs. OLOO

...but before we begin...

Types

- Primitive Types
- Abstract Operations
- Coercion
- Equality
- TypeScript, Flow, etc.

"In JavaScript, everything is an object."

false

6.1 ECMAScript Language Types

An *ECMAScript language type* corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Symbol, Number, and Object. An *ECMAScript language value* is a value that is characterized by an ECMAScript language type.

Primitive Types

- undefined
- string
- number
- boolean
- object
- symbol

- undeclared?
- null?
- function?
- array?
- bigint?

Primitive Types

- undefined
- string
- number
- boolean
- object
- symbol
- null
- bigint (future)

Not

- object
- function
- array

Objects

Primitive Types

In JavaScript, variables don't have types, values do.

```
var v;
typeof v;                  // "undefined"
v = "1";
typeof v;                  // "string"
v = 2;
typeof v;                  // "number"
v = true;
typeof v;                  // "boolean"
v = {};
typeof v;                  // "object"
v = Symbol();
typeof v;                  // "symbol"
```

Primitive Types: typeof

```
typeof doesntExist;          // "undefined"

var v = null;
typeof v;                    // "object"   OOPS!

v = function(){};
typeof v;                    // "function"   hmmm?

v = [1,2,3];
typeof v;                    // "object"   hmmm?
```

```
// coming soon!
var v = 42n;
// or:  BigInt(42)
typeof v;                    // "bigint"
```

Primitive Types: typeof

undefined

vs.

undeclared

vs.

uninitialized (aka TDZ)

Special Values

NaN (“~~not a number~~”)

Special Values

```
var myAge = Number("0o46");           // 38
var myNextAge = Number("39");         // 39
var myCatsAge = Number("n/a");        // NaN
myAge - "my son's age";               // NaN

myCatsAge === myCatsAge;              // false  OOPS!

isNaN(myAge);                         // false
isNaN(myCatsAge);                     // true
isNaN("my son's age");                // true  OOPS!

Number.isNaN(myCatsAge);              // true
Number.isNaN("my son's age");         // false
```

Special Values: NaN

NaN: Invalid Number

don't: undefined

don't: null

don't: false

don't: -1

don't: 0

Negative Zero

```
var trendRate = -0;
trendRate === -0;                    // true

trendRate.toString();                // "0"  OOPS!
trendRate === 0;                     // true  OOPS!
trendRate < 0;                       // false
trendRate > 0;                       // false

Object.is(trendRate,-0);             // true
Object.is(trendRate,0);              // false
```

Special Values: -0

```js
Math.sign(-3);          // -1
Math.sign(3);           // 1
Math.sign(-0);          // -0   WTF?
Math.sign(0);           // 0   WTF?

// "fix" Math.sign(..)
function sign(v) {
  return v !== 0 ? Math.sign(v) : Object.is(v,-0) ? -1 : 1;
}

sign(-3);               // -1
sign(3);                // 1
sign(-0);               // -1
sign(0);                // 1
```

Special Values: -0

```
function formatTrend(trendRate) {
    var direction =
        (trendRate < 0 || Object.is(trendRate,-0)) ? "▼" :
        "▲";
    return `${direction} ${Math.abs(trendRate)}`;
}

formatTrend(-3);            // "▼ 3"
formatTrend(3);             // "▲ 3"
formatTrend(-0);            // "▼ 0"
formatTrend(0);             // "▲ 0"
```

Special Values: -0

Fundamental Objects

aka: Built-In Objects

aka: Native Functions

Use new:

- Object()
- Array()
- Function()
- Date()
- RegExp()
- Error()

Don't use new:

- String()
- Number()
- Boolean()

Fundamental Objects

```
var yesterday = new Date("March 6, 2019");
yesterday.toUTCString();
// "Wed, 06 Mar 2019 06:00:00 GMT"

var myGPA = String(transcript.gpa);
// "3.54"
```

Fundamental Objects

7 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification.

7.1 Type Conversion

The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; they can accept a value of any ECMAScript language type. But no other specification types are used with these operations.

(aka "coercion")

ToPrimitive(hint) (7.1.1)

hint: "number"	hint: "string"
valueOf()	toString()
toString()	valueOf()

ToString (7.1.12)

null	"null"
undefined	"undefined"
true	"true"
false	"false"
3.14159	"3.14159"
0	"0"
-0	"0"

Abstract Operations: ToString

ToString (object):
ToPrimitive (string)

aka: toString() / valueOf()

[]	""
[1,2,3]	"1,2,3"
[null,undefined]	","
[[[],[],[]],[]]	",,,"
[,,,,]	",,,"

Abstract Operations: ToString (Array)

{}	"[object Object]"
{a:2}	"[object Object]"
{ toString(){ return "X"; } }	"X"

Abstract Operations: ToString (Object)

ToNumber (7.1.3)

""	0
"0"	0
"-0"	-0
" 009 "	9
"3.14159"	3.14159
"0."	0
".0"	0
"."	NaN
"0xaf"	175

Abstract Operations: ToNumber

false	0
true	1
null	0
undefined	NaN

Abstract Operations: ToNumber

ToNumber (object):

ToPrimitive (number)

aka: valueOf() / toString()

(for [] and {} by default):

```
valueOf() { return this; }
--> toString()
```

[""]	0
["0"]	0
["-0"]	-0
[null]	0
[undefined]	0
[1,2,3]	NaN
[[[[]]]]	0

Coercion: ToNumber (Array)

{ .. }	NaN
{ valueOf() { return 3; } }	3

Coercion: ToNumber (Object)

ToBoolean (7.1.2)

Falsy	Truthy
“”	“foo”
0, -0	23
null	{ a:1 }
NaN	[1,3]
false	true
undefined	function(){..}
	...

Abstract Operations: ToBoolean

Coercion

You claim to avoid coercion because it's evil, but...

```
var numStudents = 16;

console.log(
    `There are ${numStudents} students.`
);
// "There are 16 students."
```

Coercion: we all do it...

```
var msg1 = "There are ";
var numStudents = 16;
var msg2 = " students.";
console.log(msg1 + numStudents + msg2);
// "There are 16 students."
```

Coercion: string concatenation (number to string)

```
var numStudents = 16;

console.log(
    `There are ${numStudents+""} students.`
);
// "There are 16 students."
```

Coercion: string concatenation (number to string)

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Coercion: string concatenation (number to string)

```
var numStudents = 16;

console.log(
    `There are ${[numStudents].join("")} students.`
);
// "There are 16 students."
```

Coercion: number to string

```javascript
var numStudents = 16;

console.log(
    `There are ${numStudents.toString()} students.`
);
// "There are 16 students."
```

Coercion: number to string

```
var numStudents = 16;

console.log(
	`There are ${String(numStudents)} students.`
);
// "There are 16 students."
```

Coercion: number to string

OK, OK... but, what about...?

```
function addAStudent(numStudents) {
    return numStudents + 1;
}

addAStudent(studentsInputElem.value);
// "161"  OOPS!
```

Coercion: string to number

```javascript
function addAStudent(numStudents) {
    return numStudents + 1;
}

addAStudent(
    +studentsInputElem.value
);
// 17
```

Coercion: string to number

```
function addAStudent(numStudents) {
    return numStudents + 1;
}

addAStudent(
    Number(studentsInputElem.value)
);
// 17
```

Coercion: string to number

```
function kickStudentOut(numStudents) {
    return numStudents - 1;
}

kickStudentOut(
    studentsInputElem.value
);
// 15
```

Coercion: string to number

Yeah, but…

Recall Falsy vs Truthy?

```
if (studentsInputElem.value) {
    numStudents =
        Number(studentsInputElem.value);
}
```

```
while (newStudents.length) {
    enrollStudent(newStudents.pop());
}
```

Coercion: ___ to boolean

```
if (!!studentsInputElem.value) {
    numStudents =
        Number(studentsInputElem.value);
}
```

```
while (newStudents.length > 0) {
    enrollStudent(newStudents.pop());
}
```

Coercion: ___ to boolean

```
if (studentNameElem.value) {
    student.name = studentNameElem.value;
}

// ******************************

Boolean("");        // false
Boolean("  \t\n");  // true  OOPS!
```

Coercion: ___ to boolean

```
var workshop = getRegistration(student);

if (workshop) {
	console.log(
		`Welcome ${student.name} to ${workshop.name}.`
	);
}

// ******************************

Boolean(undefined);      // false
Boolean(null);           // false
Boolean({});             // true
```

Coercion: ___ to boolean

Ummmm.....

Boxing

```javascript
if (studentNameElem.value.length > 50) {
    console.log("Student's name too long.");
}
```

Coercion: primitive to object

All programming languages have type conversions, because it's absolutely necessary.

You use coercion in JS whether you admit it or not, because you have to.

Every language has type conversion corner cases

```javascript
Number( "" );                      // 0           OOPS!
Number( "  \t\n" );                // 0           OOPS!
Number( null );                    // 0           OOPS!
Number( undefined );               // NaN
Number( [] );                      // 0           OOPS!
Number( [1,2,3] );                 // NaN
Number( [null] );                  // 0           OOPS!
Number( [undefined] );             // 0           OOPS!
Number( {} );                      // NaN

String( -0 );                      // "0"         OOPS!
String( null );                    // "null"
String( undefined );               // "undefined"
String( [null] );                  // ""          OOPS!
String( [undefined] );             // ""          OOPS!

Boolean( new Boolean(false) );     // true        OOPS!
```

Coercion: corner cases

The Root Of All (Coercion) Evil

```
studentsInput.value = "";

// ..

Number(studentsInput.value);          // 0
```

```
studentsInput.value = "   \t\n";

// ..

Number(studentsInput.value);          // 0
```

Coercion: corner cases

```
Number(true);            // 1
Number(false);           // 0

1 < 2;                   // true
2 < 3;                   // true
1 < 2 < 3;               // true  (but...)

(1 < 2) < 3;
(true) < 3;
1 < 3;                   // true  (hmm...)

// *********************

3 > 2;                   // true
2 > 1;                   // true
3 > 2 > 1;               // false  OOPS!

(3 > 2) > 1;
(true) > 1;
1 > 1;                   // false
```

Coercion: corner cases

You don't deal with these type conversion corner cases by avoiding coercions.

Instead, you have to adopt a coding style that makes value types plain and obvious.

A quality JS program embraces coercions, making sure the types involved in every operation are clear. Thus, corner cases are safely managed.

~~Type Rigidity~~

~~Static Types~~

~~Type Soundness~~

JavaScript's dynamic typing is not a weakness, it's one of its strong qualities

But… but…
what about the junior devs?

Implicit != Magic

Implicit != Bad

Implicit: Abstracted

Hiding unnecessary details, re-focusing the reader and increasing clarity

```
var numStudents = 16;

console.log(
    `There are ${String(numStudents)} students.`
);
// "There are 16 students."
```

```
var numStudents = 16;

console.log(
    `There are ${numStudents} students.`
);
// "There are 16 students."
```

Coercion: implicit can be good (sometimes)

```
var workshopEnrollment1 = 16;
var workshopEnrollment2 = workshop2Elem.value;

if (Number(workshopEnrollment1) < Number(workshopEnrollment2)) {
    // ..
}

if (workshopEnrollment1 < workshopEnrollment2) {
    // ..
}
```

Coercion: implicit can be good (sometimes)

Is showing the reader the extra type details helpful or distracting?

~~"If a feature is sometimes useful and sometimes dangerous and if there is a better option then always use the better option."~~

-- "The Good Parts", Crockford

Useful: when the reader is focused on what's important

Dangerous: when the reader can't tell what will happen

Better: when the reader understands the code

It is irresponsible to knowingly avoid usage of a feature that can improve code readability

Equality

== vs. ===

== checks value (loose)

=== checks value and type (strict)

?

Loose Equality vs. Strict Equality

If you're trying to understand your code, it's critical you learn to think like JS

7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

Loose Equality: still types, and ===

```
var studentName1 = "Frank";
var studentName2 = `${studentName1}`;

var workshopEnrollment1 = 16;
var workshopEnrollment2 = workshopEnrollment1 + 0;

studentName1 == studentName2;                 // true
studentName1 === studentName2;                // true

workshopEnrollment1 == workshopEnrollment2;   // true
workshopEnrollment1 === workshopEnrollment2;  // true
```

Coercive Equality: == and ===

7.2.15 Strict Equality Comparison

The comparison *x* === *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Number, then
   a. If *x* is **NaN**, return **false**.
   b. If *y* is **NaN**, return **false**.
   c. If *x* is the same Number value as *y*, return **true**.
   d. If *x* is **+0** and *y* is **-0**, return **true**.
   e. If *x* is **-0** and *y* is **+0**, return **true**.
   f. Return **false**.
3. Return SameValueNonNumber(*x*, *y*).

Strict Equality: types and lies

```
var workshop1 = {
    name: "Deep JS Foundations"
};

var workshop2 = {
    name: "Deep JS Foundations"
};

if (workshop1 == workshop2) {
    // Nope
}

if (workshop1 === workshop2) {
    // Nope
}
```

Equality: identity, not structure

== checks value (loose)

=== checks value and type (strict)

== allows coercion (types different)

=== disallows coercion (types same)

Coercive Equality vs. Non-Coercive Equality

Like every other operation, is coercion helpful in an equality comparison or not?

Like every other operation, do we know the types or not?

7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

Coercive Equality: null == undefined

```javascript
var workshop1 = { topic: null };
var workshop2 = {};

if (
    (workshop1.topic === null || workshop1.topic === undefined) &&
    (workshop2.topic === null || workshop2.topic === undefined) &&
) {
    // ..
}

if (
    workshop1.topic == null &&
    workshop2.topic == null
) {
    // ..
}
```

Coercive Equality: null == undefined

7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

Coercive Equality: prefers numeric comparison

```
var workshopEnrollment1 = 16;
var workshopEnrollment2 = workshop2Elem.value;

if (Number(workshopEnrollment1) === Number(workshopEnrollment2)) {
    // ..
}

// Ask: what do we know about the types here?
if (workshopEnrollment1 == workshopEnrollment2) {
    // ..
}
```

Coercive Equality: prefers numeric comparison

7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

Coercive Equality: only primitives

```js
var workshop1Count = 42;
var workshop2Count = [42];

if (workshop1Count == workshop2Count) {
    // Yep (hmm...)
}
```

Coercive Equality: only primitives

```
var workshop1Count = 42;
var workshop2Count = [42];

// if (workshop1Count == workshop2Count) {
// if (42 == "42") {
// if (42 === 42) {
if (true) {
    // Yep (hmm...)
}
```

Coercive Equality: only primitives

== Summary:

If the types are the same: ===

If null or undefined: equal

If non-primitives: ToPrimitive

Prefer: ToNumber

== Corner Cases

```js
[] == ![];          // true  WAT!?
```

```js
var workshop1Students = [];
var workshop2Students = [];

if (workshop1Students == !workshop2Students) {
    // Yep, WAT!?
}

if (workshop1Students != workshop2Students) {
    // Yep, WAT!?
}
```

== Corner Cases: WAT!?

```
var workshop1Students = [];
var workshop2Students = [];

// if (workshop1Students == !workshop2Students) {
// if ([] == false) {
// if ("" == false) {
// if (0 == false) {
// if (0 === 0) {
if (true) {
    // Yep, WAT!?
}

// if (workshop1Students != workshop2Students) {
// if (!(workshop1Students == workshop2Students)) {
// if (!(false)) {
if (true) {
    // Yep, WAT!?
}
```

== Corner Cases: WAT!?

```
var workshopStudents = [];

if (workshopStudents) {
    // Yep
}

if (workshopStudents == true) {
    // Nope :(
}

if (workshopStudents == false) {
    // Yep :(
}
```

== Corner Cases: booleans

```
var workshopStudents = [];

// if (workshopStudents) {
// if (Boolean(workshopStudents)) {
if (true) {
    // Yep
}

// if (workshopStudents == true) {
// if ("" == true) {
// if (0 === 1) {
if (false) {
    // Nope :(
}

// if (workshopStudents == false) {
// if ("" == false) {
// if (0 === 0) {
if (true) {
    // Yep :(
}
```

== Corner Cases: booleans

Avoid:

1. == with 0 or "" (or even " ")
2. == with non-primitives
3. == true or == false : allow ToBoolean or use ===

The case for preferring ==

Knowing types is always better than not knowing them

Static Types is <u>not</u> the only (or even necessarily best) way to know your types

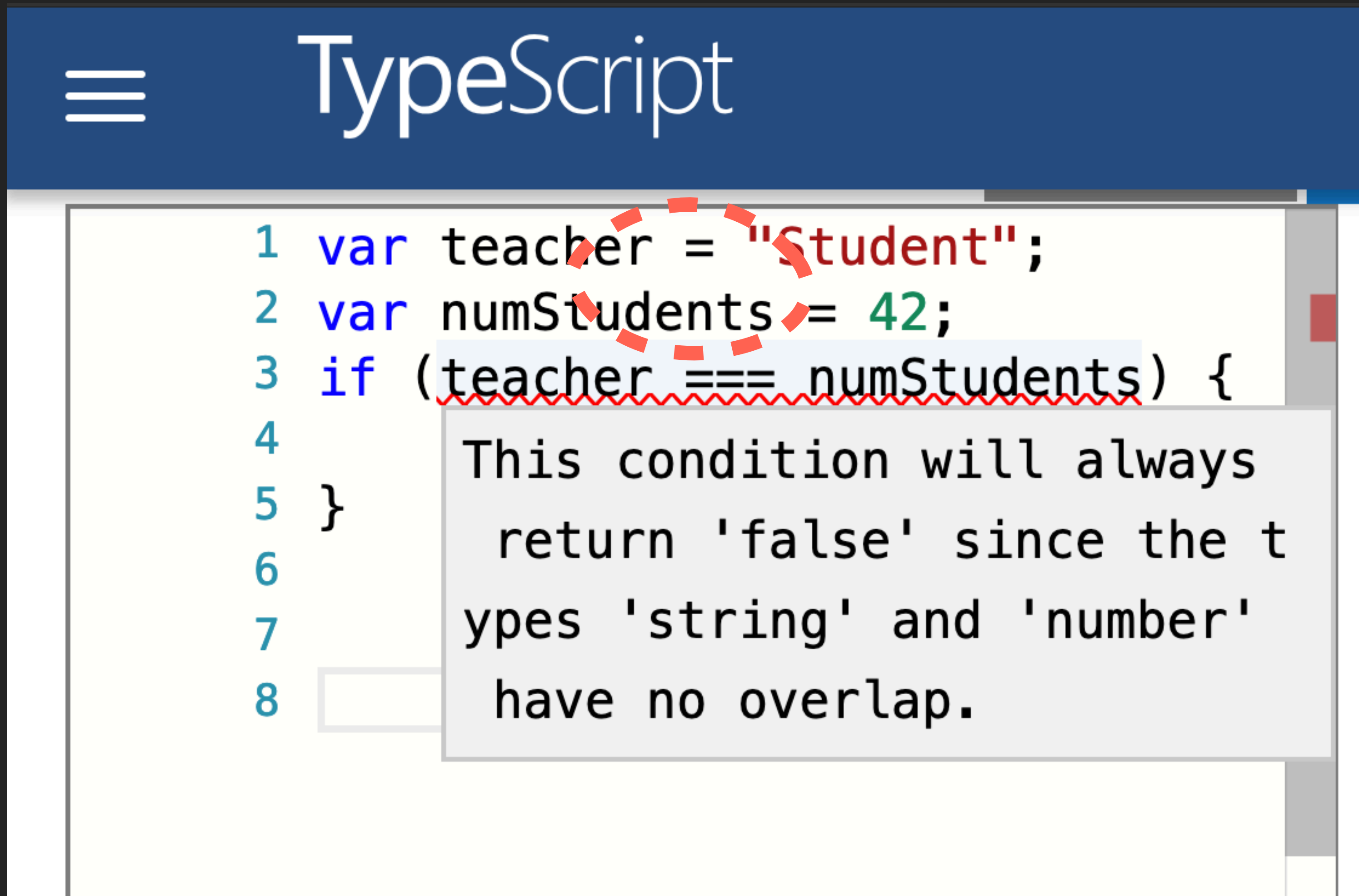
== is <u>not</u> about comparisons with unknown types

== is about comparisons with known type(s), <u>optionally</u> where conversions are helpful

If you know the type(s) in a comparison:

If both types are the same,
== is identical to ===

Using === would be unnecessary,
so prefer the shorter ==



Since === is pointless when the types don't match, it's similarly unnecessary when they do match.

If you know the type(s) in a comparison:

If the types are different, using one === would be broken

Prefer the more powerful == or don't compare at all

If you <u>know</u> the type(s) in a comparison:

If the types are different, the equivalent of one == would be two (or more!) === (ie, "slower")

Prefer the "<u>faster</u>" single ==

If you know the type(s) in a comparison:

If the types are different, two (or more!) === comparisons may distract the reader

Prefer the cleaner single ==

If you <u>know</u> the type(s) in a comparison:

Summary: whether the types match or not, `==` is the <u>more sensible</u> choice

If you don't know the type(s) in a comparison:

Not knowing the types means not fully understanding that code

So, best to refactor so you can know the types

If you <u>don't know</u> the type(s) in a comparison:

The uncertainty of not knowing types should be obvious to reader

The <u>most obvious</u> signal is ===

If you don't know the type(s) in a comparison:

Not knowing the types is equivalent to assuming type conversion

Because of corner cases, the only safe choice is ===

If you don't know the type(s) in a comparison:

Summary: if you can't or won't use known and obvious types, === is the only reasonable choice

Even if === would always be equivalent to == in your code, using it <u>everywhere</u> sends a wrong semantic signal: "Protecting myself since I don't know/trust the types"

Summary: making types known and obvious leads to better code. If types are known, == is best.

Otherwise, fall back to ===.

TypeScript, Flow, and type-aware linting

Benefits:

1. Catch type-related mistakes
2. Communicate type intent
3. Provide IDE feedback

Caveats:

1. Inferencing is best-guess, not a guarantee
2. Annotations are optional
3. Any part of the application that isn't typed introduces uncertainty

TypeScript & Flow

```
var teacher = "Kyle";

// ..

teacher = { name: "Kyle" };
// Error: can't assign object
// to string
```

Type-Aware Linting: inferencing

TypeScript & Flow

```
var teacher: string = "Kyle";

// ..

teacher = { name: "Kyle" };
// Error: can't assign object
// to string
```

Type-Aware Linting: annotating

TypeScript & Flow

```
type student = { name: string };

function getName(studentRec: student): string {
    return studentRec.name;
}

var firstStudent: student = { name: "Frank" };

var firstStudentName: string = getName(firstStudent);
```

Type-Aware Linting: custom types & signatures

TypeScript & Flow

```
var studentName: string = "Frank";

var studentCount: number = 16 - studentName;
// error: can't substract string
```

Type-Aware Linting: validating operand types https://github.com/niieani/typescript-vs-flowtype

TypeScript & Flow:
Pros and Cons

They make types more obvious in code

Familiarity: they look like other language's type systems

TypeScript/Flow: Pros

Extremely popular these days

They're very sophisticated and good at what they do

They use "non-JS-standard" syntax (or code comments)

They require* a build process, which raises the barrier to entry

Their sophistication can be intimidating to those without prior formal types experience

They focus more on "static types" (variables, parameters, returns, properties, etc) than <u>value types</u>

The only way to have confidence over the runtime behavior is to limit/eliminate dynamic typing

Alternative?

Typl

https://github.com/getify/Typl

Motivations:

1. Only standard JS syntax
2. Compiler and Runtime (both optional)
3. Completely configurable (ie, ESLint)
4. Main focus: inferring or annotating values; Optional: "static typing"
5. With the grain of JS, not against it

```
var teacher = "Kyle";

// ..

teacher = { name: "Kyle" };
// Error: can't assign object
// to string
```

TypI: inferencing + optional "static types"

```
var teacher = string`Kyle`;

// ..

teacher = { name: string`Kyle` };
// Error: can't assign object
// to string
```

```
var teacher = string`Kyle`;

// ..

teacher = object`${{ name: string`Kyle` }}`;
// Error: can't assign object
// to string
```

Typl: tagging literals

```
var student = { age: int`42` };

var studentAge = number`${student.age}` + number`1`;
```

Typl: type assertion (tagging expressions)

```
function getName(studentRec = { name = string }) {
    return studentRec.name;
}

var firstStudent = { name: string`Frank` };

var firstStudentName = getName(firstStudent);
```

Typl: type signatures (functions, objects, etc)

```
function fetchStudent(
    id = int,
    onRecord = func`({ name = string }) => undef`
) {
    // do something asynchronous

    onRecord(student);
}

function printName(student = { name = string }) {
    console.log(student.name);
}

var cb = printName;

fetchStudent(42,cb);
```

Typl: inline & persistent type signatures

```
var three = gimme(3);
var greeting = "hello " + three;
// error: 'string' + 'int'

function gimme(num) {
	return num;
}
```

Typl: powerful multi-pass inferencing

```
function showInfo(
    name = string``, topic = string``, count = int`0`
) {
    console.log(
        `${name}: ${topic} (${String(count)})`
    );
}

var teacher = string`Kyle`;
var workshop = string`Deep JS Foundations`;
var numStudents =
    int`${Number(studentsElem.value)}`;

showInfo(teacher,workshop,numStudents);
```

Typl: compiler vs runtime

```
function showInfo(name,topic = "",count = 0) {
    name = string`${name}`;
    topic = string`${topic}`;
    count = int`${count}`;
    console.log(
        `${name}: ${topic} (${String(count)})`
    );
}

var teacher = "Kyle";
var workshop = "Deep JS Foundations";
var numStudents =
    int`${Number(studentsElem.value)}`;

showInfo(teacher,workshop,numStudents);
```

Typl: compiled (some runtime removed)

Much more to come...

Wrapping Up

JavaScript has a (dynamic) type system, which uses various forms of coercion for value type conversion, including equality comparisons

However, the prevailing response seems to be: avoid as much of this system as possible, and use === to "protect" from needing to worry about types

Part of the problem with <u>avoidance</u> of whole swaths of JS, like pretending === saves you from needing to know types, is that it tends to systemically perpetuate bugs

You simply cannot write quality JS programs without knowing the types involved in your operations.

Alternately, many choose to adopt a different "static types" system layered on top

While certainly helpful in some respects, this is "avoidance" of a different sort

Apparently, JS's type system is inferior so it must be replaced, rather than learned and leveraged

Many claim that JS's type system is too difficult for newer devs to learn, and that static types are (somehow) more learnable

My claim: the better approach is to embrace and learn JS's type system, and to adopt a coding style which makes types as obvious as possible

By doing so, you will make your code more readable and more robust, for experienced and new developers alike

As an option to aid in that effort, I created Typl, which I believe embraces and unlocks the best parts of JS's types and coercion.

Scope

- Nested Scope
- Hoisting
- Closure
- Modules

Scope: where to look for things

```
x = 42;
console.log(y);
```

Scope: sorting marbles

JavaScript organizes scopes with <u>functions</u> and blocks

```javascript
var teacher = "Kyle";

function otherClass() {
    var teacher = "Suzy";
    console.log("Welcome!");
}

function ask() {
    var question = "Why?";
    console.log(question);
}

otherClass();           // Welcome!
ask();                  // Why?
```

Scope

```
var teacher = "Kyle";

function otherClass() {
    teacher = "Suzy";
    topic = "React";
    console.log("Welcome!");
}

otherClass();            // Welcome!

teacher;                 // ??
topic;                   // ??
```

Suzy

React

Scope

```
"use strict";

var teacher = "Kyle";

function otherClass() {
	teacher = "Suzy";
	topic = "React"; ReferenceError
	console.log("Welcome!");
}

otherClass();
```

Scope

```
var teacher = "Kyle";

function otherClass() {
	var teacher = "Suzy";

	function ask(question) {
		console.log(teacher,question);
	}

	ask("Why?");
}

otherClass();          // Suzy Why?
ask("????");
```

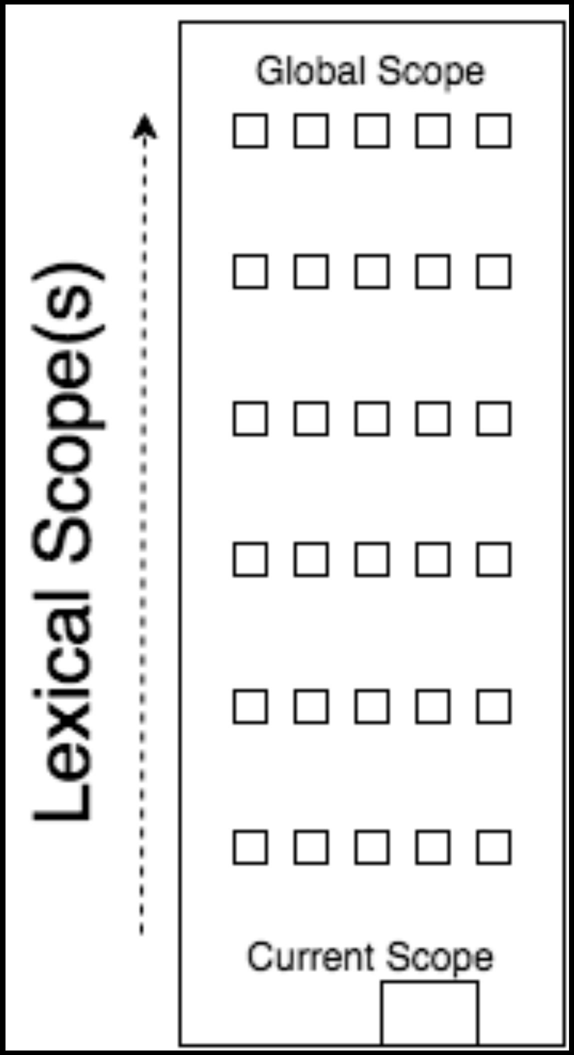
ReferenceError

Scope

undefined

vs.

undeclared



Scope

```
function teacher() { /* .. */ }

var myTeacher = function anotherTeacher(){
    console.log(anotherTeacher);
};

console.log(teacher);
console.log(myTeacher);
console.log(anotherTeacher);
```

ReferenceError

Scope: which scope?

Named Function Expressions

```
var clickHandler = function(){
    // ..
};

var keyHandler = function keyHandler(){
    // ..
};
```

Named Function Expressions

1. Reliable function self-reference (recursion, etc)

2. More debuggable stack traces

3. More self-documenting code

Named Function Expressions: Benefits

```javascript
var ids = people.map(person => person.id);

var ids = people.map(function getId(person){
	return person.id;
});

// ********************

getPerson()
.then(person => getData(person.id))
.then(renderData);

getPerson()
.then(function getDataFrom(person){
	return getData(person.id);
})
.then(renderData);
```

Named Function Expressions vs. Anonymous Arrow Functions

```
var getId = person => person.id;
var ids = people.map(getId);

// *********************

var getDataFrom = person => getData(person.id);
getPerson()
.then(getDataFrom)
.then(renderData);
```

Named (Arrow) Function Expressions? Still no...

(Named) Function Declaration

>

Named Function Expression

>

Anonymous Function Expression

lexical scope

dynamic scope

```
var teacher = "Kyle";

function otherClass() {
    var teacher = "Suzy";

    function ask(question) {
        console.log(teacher,question);
    }

    ask("Why?");
}
```

Scope: lexical

```
var x = function foo() {
    var y = foo();
}
```

```
var teacher = "Ky

function otherClass() {
    var teacher = "Suzy";

    function ask(question) {
        console.log(teacher,question);
    }

    ask("Why?");
}
```

Sublime-Levels

Scope: lexical

```
var teacher = "Kyle";

function ask(question) {
    console.log(teacher,question);
}

function otherClass() {
    var teacher = "Suzy";

    ask("Why?");
}

otherClass();
```

THEORETICAL

Scope: dynamic

Function Scoping

```
var teacher = "Kyle";

// ..

var teacher = "Suzy";
console.log(teacher);    // Suzy

// ..

console.log(teacher);    // Suzy -- oops!
```

Function Scoping

```
var teacher = "Kyle";

function anotherTeacher() {
    var teacher = "Suzy";
    console.log(teacher);    // Suzy
}

anotherTeacher();

console.log(teacher);    // Kyle
```

Function Scoping

```
var teacher = "Kyle";

function anotherTeacher() {
    var teacher = "Suzy";
    console.log(teacher);   // Suzy
}

( anotherTeacher )();

console.log(teacher);   // Kyle
```

Function Scoping

```js
var teacher = "Kyle";

( function anotherTeacher() {
    var teacher = "Suzy";
    console.log(teacher);    // Suzy
} )();

console.log(teacher);    // Kyle
```

http://benalman.com/news/2010/11/immediately-invoked-function-expression/

Function Scoping: IIFE

```js
var teacher = "Kyle";

// this IIFE is anonymous :(
(function(teacher){
    console.log(teacher);    // Suzy
})("Suzy");

console.log(teacher);   // Kyle
```

Function Scoping: IIFE

```
var teacher;
try {
    teacher = fetchTeacher(1);
}
catch (err) {
    teacher = "Kyle";
}
```

Function Scoping: IIFE

```javascript
var teacher = (function getTeacher(){
    try {
        return fetchTeacher(1);
    }
    catch (err) {
        return "Kyle";
    }
})();
```

Function Scoping: IIFE

Block Scoping

Instead of an IIFE?

```
var teacher = "Kyle";

( function anotherTeacher() {
    var teacher = "Suzy";
    console.log(teacher);    // Suzy
} )();

console.log(teacher);    // Kyle
```

Block Scoping: encapsulation

```js
var teacher = "Kyle";

{
    let teacher = "Suzy";
    console.log(teacher);    // Suzy
}

console.log(teacher);    // Kyle
```

Block Scoping: encapsulation

```javascript
function diff(x,y) {
    if (x > y) {
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Block Scoping: intent

```
function diff(x,y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Block Scoping: let

```
function repeat(fn,n) {
    var result;

    for (var i = 0; i < n; i++) {
        result = fn( result, i );
    }

    return result;
}
```

Block Scoping: "well, actually, not all vars..."

```
function repeat(fn,n) {
    var result;

    for (let i = 0; i < n; i++) {
        result = fn( result, i );
    }

    return result;
}
```

Block Scoping: let + var

```
function lookupRecord(searchStr) {
    try {
        var id = getRecord( searchStr );
    }
    catch (err) {
        var id = -1;
    }

    return id;
}
```

Block Scoping: sometimes var > let

```
function formatStr(str) {
    { let prefix, rest;
        prefix = str.slice( 0, 3 );
        rest = str.slice( 3 );
        str = prefix.toUpperCase() + rest;
    }

    if (/^FOO:/.test( str )) {
        return str;
    }

    return str.slice( 4 );
}
```

Block Scoping: explicit let block

```javascript
var teacher = "Suzy";
teacher = "Kyle";     // OK

const myTeacher = teacher;
myTeacher = "Suzy"; // TypeError

const teachers = ["Kyle","Suzy"];
teachers[1] = "Brian";  // Allowed!
```

Block Scoping: const(antly confusing)

Hoisting

```
student;                                   // ??
teacher;                                   // ??
var student = "you";
var teacher = "Kyle";
```

Scope: hoisting

```
var student;
var teacher;

student;                    // undefined
teacher;                    // undefined
student = "you";
teacher = "Kyle";
```

Scope: hoisting

```
teacher();          // Kyle
otherTeacher();     // ??

function teacher() {
    return "Kyle";
}

var otherTeacher = function(){
    return "Suzy";
};
```

Scope: hoisting

```javascript
function teacher() {
	return "Kyle";
}
var otherTeacher;

teacher();		// Kyle
otherTeacher();	// TypeError

otherTeacher = function(){
	return "Suzy";
};
```

Scope: hoisting

```javascript
var teacher = "Kyle";
otherTeacher();                  // ??

function otherTeacher() {
    console.log(teacher);
    var teacher = "Suzy";
}
```

undefined

Scope: hoisting

```
// var hoisting?
// usually bad :/
teacher = "Kyle";
var teacher;

// function hoisting?
// IMO actually pretty useful
getTeacher();             // Kyle

function getTeacher() {
    return teacher;
}
```

Scope: hoisting

"let doesn't hoist"? false

```
{
    teacher = "Kyle"; // TDZ error!
    let teacher;
}
```

```
var teacher = "Kyle";

{
    console.log(teacher); // TDZ error!
    let teacher = "Suzy";
}
```

"let doesn't hoist"? false

13.3.1 Let and Const Declarations

NOTE **`let`** and **`const`** declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initializer* is assigned the value of its *Initializer*'s *AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a **`let`** declaration does not have an *Initializer* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

Closure

Closure is when a function “remembers” its lexical scope even when the function is executed outside that lexical scope.

```
function ask(question) {
    setTimeout(function waitASec(){
        console.log(question);
    },100);
}

ask("What is closure?");
// What is closure?
```

Closure

```
function ask(question) {
    return function holdYourQuestion(){
        console.log(question);
    };
}

var myQuestion = ask("What is closure?");

// ..

myQuestion(); // What is closure?
```

Closure

```
var teacher = "Kyle";

var myTeacher = function(){
    console.log(teacher);
};

teacher = "Suzy";

myTeacher();        // ??  Suzy
```

Closure: NOT capturing a value

```
for (var i = 1; i <= 3; i++) {
        setTimeout(function(){
                console.log(`i: ${i}`);
        },i * 1000);
}
// i: 4
// i: 4
// i: 4
```

Closure: loops

```
for (var i = 1; i <= 3; i++) {
    let j = i;
    setTimeout(function(){
        console.log(`j: ${j}`);
    },j * 1000);
}
// j: 1
// j: 2
// j: 3
```

Closure: loops

```
for (let i = 1; i <= 3; i++) {
    setTimeout(function(){
        console.log(`i: ${i}`);
    },i * 1000);
}
// i: 1
// i: 2
// i: 3
```

Closure: loops

Modules

```javascript
var workshop = {
	teacher: "Kyle",
	ask(question) {
		console.log(this.teacher,question);
	},
};

workshop.ask("Is this a module?");
// Kyle Is this a module?
```

Namespace, NOT a module

Modules encapsulate data and behavior (methods) together. The state (data) of a module is held by its methods via closure.

```
var workshop = (function Module(teacher){
    var publicAPI = { ask, };
    return publicAPI;

    // ***********

    function ask(question) {
        console.log(teacher,question);
    }
})("Kyle");

workshop.ask("It's a module, right?");
// Kyle It's a module, right?
```

Classic/Revealing module pattern

```
function WorkshopModule(teacher){
    var publicAPI = { ask, };
    return publicAPI;

    // ***********

    function ask(question) {
        console.log(teacher,question);
    }
};

var workshop = WorkshopModule("Kyle");

workshop.ask("It's a module, right?");
// Kyle It's a module, right?
```

Module Factory

workshop.mjs:

```
var teacher = "Kyle";

export default function ask(question) {
	console.log(teacher,question);
};
```

```
import ask from "workshop.mjs";

ask("It's a default import, right?");
// Kyle It's a default import, right?


import * as workshop from "workshop.mjs";

workshop.ask("It's a namespace import, right?");
// Kyle It's a namespace import, right?
```

ES6 module pattern

Objects (Oriented)

- this
- class { }
- Prototypes
- "Inheritance" vs. "Behavior Delegation" (OO vs. OLOO)

this

A function's this references the execution context for that call, determined entirely by how the function was called.

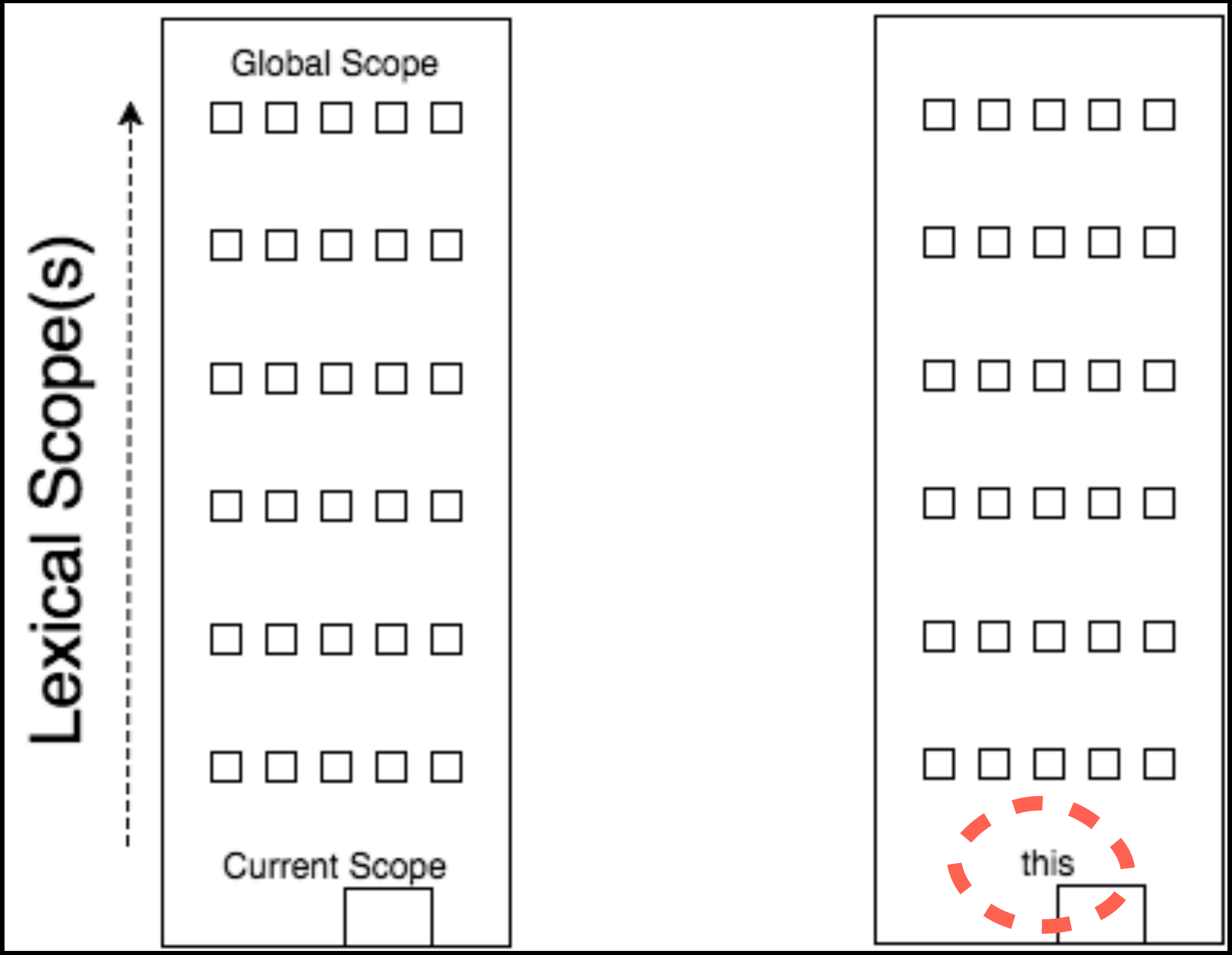
A this-aware function can thus have a different context each time it's called, which makes it more flexible & reusable.

```
var teacher = "Kyle";

function ask(question) {
    console.log(teacher,question);
}

function otherClass() {
    var teacher = "Suzy";

    ask("Why?");
}

otherClass();
```

Recall: dynamic scope

```
function ask(question) {
    console.log(this.teacher,question);
}

function otherClass() {
    var myContext = {
        teacher: "Suzy"
    };
    ask.call(myContext,"Why?"); // Suzy Why?
}

otherClass();
```

Dynamic Context ~= JS's Dynamic Scope



this vs. Scope

```
var workshop = {
    teacher: "Kyle",
    ask(question) {
        console.log(this.teacher,question);
    },
};

workshop.ask("What is implicit binding?");
// Kyle What is implicit binding?
```

this: implicit binding

```
function ask(question) {
    console.log(this.teacher,question);
}

var workshop1 = {
    teacher: "Kyle",
    ask: ask,
};

var workshop2 = {
    teacher: "Suzy",
    ask: ask,
}

workshop1.ask("How do I share a method?");
// Kyle How do I share a method?

workshop2.ask("How do I share a method?");
// Suzy How do I share a method?
```

this: dynamic binding -> sharing

```javascript
function ask(question) {
    console.log(this.teacher,question);
}

var workshop1 = {
    teacher: "Kyle",
};

var workshop2 = {
    teacher: "Suzy",
}

ask.call(workshop1,"Can I explicitly set context?");
// Kyle Can I explicitly set context?

ask.call(workshop2,"Can I explicitly set context?");
// Suzy Can I explicitly set context?
```

this: explicit binding

```javascript
var workshop = {
    teacher: "Kyle",
    ask(question) {
        console.log(this.teacher,question);
    },
};

setTimeout(workshop.ask,10,"Lost this?");
// undefined Lost this?

setTimeout(workshop.ask.bind(workshop),10,"Hard bound this?");
// Kyle Hard bound this?
```

this: hard binding

"constructor calls"

```
function ask(question) {
    console.log(this.teacher,question);
}

var newEmptyObject = new ask("What is 'new' doing here?");
// undefined What is 'new' doing here?
```

this: new binding

1. Create a brand new empty object

2.* Link that object to another object

3. Call function with this set to the new object

4. If function does not return an object,

assume return of this

new: steps

```javascript
var teacher = "Kyle";

function ask(question) {
    console.log(this.teacher,question);
}

function askAgain(question) {
    "use strict";
    console.log(this.teacher,question);
}

ask("What's the non-strict-mode default?");
// Kyle What's the non-strict-mode default?

askAgain("What's the strict-mode default?");
// TypeError
```

this: default binding

```js
var workshop = {
    teacher: "Kyle",
    ask: function ask(question) {
        console.log(this.teacher,question);
    },
};


new (workshop.ask.bind(workshop))("What does this do?");
// undefined What does this do?
```

this: binding rule precedence?

1. Is the function called by new?

2. Is the function called by call() or apply()?

   Note: bind() effectively uses apply()

3. Is the function called on a context object?

4. DEFAULT: global object (except strict mode)

this: determination

```
var workshop = {
	teacher: "Kyle",
	ask(question) {
		setTimeout(() => {
			console.log(this.teacher,question);
		},100);
	},
};

workshop.ask("Is this lexical 'this'?");
// Kyle Is this lexical 'this'?
```

this: arrow functions

An arrow function is this-bound (aka .bind()) to its parent function.

14.2.16 Runtime Semantics: Evaluation

ArrowFunction **:** *ArrowParameters* **=>** *ConciseBody*

1. If the function code for this *ArrowFunction* is strict mode code, let *strict* be **true**. Otherwise let *strict* be **false**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *parameters* be CoveredFormalsList of *ArrowParameters*.
4. Let *closure* be FunctionCreate(Arrow, *parameters*, *ConciseBody*, *scope*, *strict*).
5. Return *closure*.

NOTE An *ArrowFunction* does not define local bindings for **`arguments`**, **`super`**, **`this`**, or **`new.target`**. Any reference to **`arguments`**, **`super`**, **`this`**, or **`new.target`** within an *ArrowFunction* must resolve to a binding in a lexically enclosing environment. Typically this will be the Function Environment of an immediately enclosing

this: arrow functions

~~An arrow function is **this**-bound (aka **.bind()**) to its parent function.~~

An arrow function doesn't define a **this**, so it's like any normal variable, and resolves lexically (aka "lexical **this**").

```js
var workshop = {
	teacher: "Kyle",
	ask: (question) => {
		console.log(this.teacher,question);
	},
};

workshop.ask("What happened to 'this'?");
// undefined What happened to 'this'?

workshop.ask.call(workshop,"Still no 'this'?");
// undefined Still no 'this'?
```

this: arrow functions

Only use => arrow functions when you need lexical this.

https://github.com/getify/eslint-plugin-arrow-require-this class { }

ES6

```js
class Workshop {
	constructor(teacher) {
		this.teacher = teacher;
	}
	ask(question) {
		console.log(this.teacher,question);
	}
}

var deepJS = new Workshop("Kyle");
var reactJS = new Workshop("Suzy");

deepJS.ask("Is 'class' a class?");
// Kyle Is 'class' a class?

reactJS.ask("Is this class OK?");
// Suzy Is this class OK?
```

ES6 class

```
class Workshop {
    constructor(teacher) {
        this.teacher = teacher;
    }
    ask(question) {
        console.log(this.teacher,question);
    }
}

class AnotherWorkshop extends Workshop {
    speakUp(msg) {
        this.ask(msg);
    }
}

var JSRecentParts = new AnotherWorkshop("Kyle");

JSRecentParts.speakUp("Are classes getting better?");
// Kyle Are classes getting better?
```

ES6 class: extends (inheritance)

```
class Workshop {
	constructor(teacher) {
		this.teacher = teacher;
	}
	ask(question) {
		console.log(this.teacher,question);
	}
}

class AnotherWorkshop extends Workshop {
	ask(msg) {
		super.ask(msg.toUpperCase());
	}
}

var JSRecentParts = new AnotherWorkshop("Kyle");

JSRecentParts.ask("Are classes super?");
// Kyle ARE CLASSES SUPER?
```

ES6 class: super (relative polymorphism)

```
class Workshop {
    constructor(teacher) {
        this.teacher = teacher;
    }
    ask(question) {
        console.log(this.teacher,question);
    }
}

var deepJS = new Workshop("Kyle");

setTimeout(deepJS.ask,100,"Still losing 'this'?");
// undefined Still losing 'this'?
```

ES6 class: still dynamic this

```
class Workshop {
    constructor(teacher) {
        this.teacher = teacher;
        this.ask = question => {
            console.log(this.teacher,question);
        };
    }
}

var deepJS = new Workshop("Kyle");

setTimeout(deepJS.ask,100,"Is 'this' fixed?");
// Kyle Is 'this' fixed?
```

ES6 class: "fixing" this?

https://gist.github.com/getify/86bed0bb78ccb517c84a6e61ec16adca

```javascript
var method = (function defineMethod(){
    var instances = new WeakMap();

    return function method(obj,methodName,fn) {
        Object.defineProperty(obj,methodName,{
            get() {
                if (!instances.has(this)) {
                    instances.set(this,{});
                }
                var methods = instances.get(this);
                if (!(methodName in methods)) {
                    methods[methodName] = fn.bind(this);
                }
                return methods[methodName];
            }
        });
    }
})();

function bindMethods(obj) {
    for (let ownProp of Object.getOwnPropertyNames(obj)) {
        if (typeof obj[ownProp] == "function") {
            method(obj,ownProp,obj[ownProp]);
        }
    }
}
```

ES6 class: hacktastrophy

```
class Workshop {
	constructor(teacher) {
		this.teacher = teacher;
	}
	ask(question) {
		console.log(this.teacher,question);
	}
}

class AnotherWorkshop extends Workshop {
	speakUp(msg) {
		this.ask(msg);
	}
}

var JSRecentParts = new AnotherWorkshop("Kyle");

bindMethods(Workshop.prototype);
bindMethods(AnotherWorkshop.prototype);

JSRecentParts.speakUp("What's different here?");
// Kyle What's different here?

setTimeout(JSRecentParts.speakUp,100,"Oh! But does this feel gross?");
// Kyle Oh! But does this feel gross?
```

ES6 class: inheritable hard this-bound methods

Prototypes

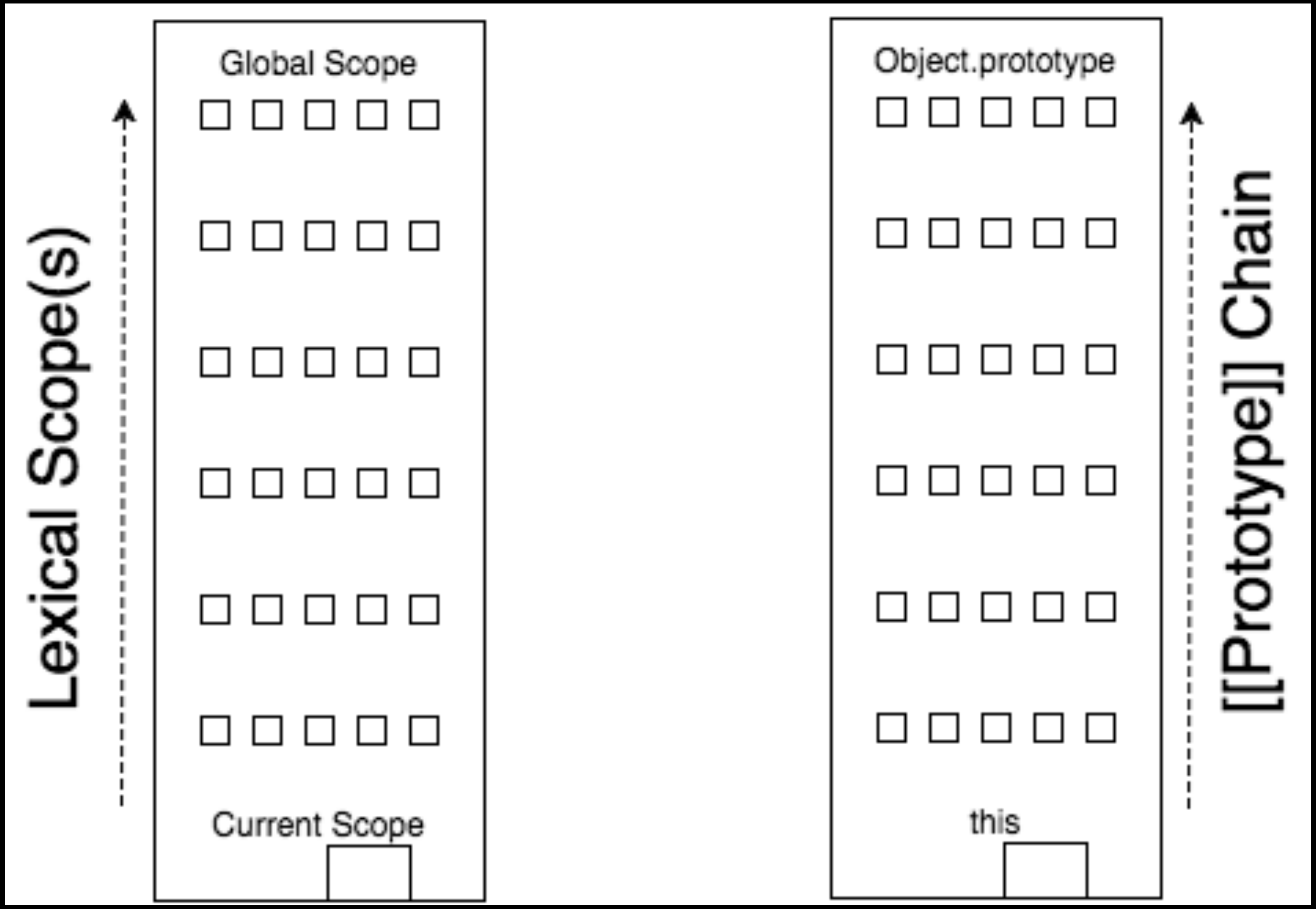
Objects are built by
"constructor calls" (via new)

A "constructor call" makes an object "~~based on~~" its own prototype

A "constructor call" makes an object linked to its own prototype

```javascript
function Workshop(teacher) {
	this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
	console.log(this.teacher,question);
};

var deepJS = new Workshop("Kyle");
var reactJS = new Workshop("Suzy");

deepJS.ask("Is 'prototype' a class?");
// Kyle Is 'prototype' a class?

reactJS.ask("Isn't 'prototype' ugly?");
// Suzy Isn't 'prototype' ugly?
```

Prototypes: as "classes"


Lexical Scope(s)
Global Scope
Current Scope
Object.prototype
this
[[Prototype]] Chain

Prototypes

```
function Workshop(teacher) {
    this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
    console.log(this.teacher,question);
};

var deepJS = new Workshop("Kyle");

deepJS.constructor === Workshop;

deepJS.__proto__ === Workshop.prototype; // true
Object.getPrototypeOf(deepJS) === Workshop.prototype; // true
```

Prototypes

```
function Workshop(teacher) {
    this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
    console.log(this.teacher,question);
};

var deepJS = new Workshop("Kyle");

deepJS.ask = function(question){
    this.ask(question.toUpperCase());
};

deepJS.ask("Oops, is this infinite recursion?");
```

Prototypes: shadowing

```javascript
function Workshop(teacher) {
    this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
    console.log(this.teacher,question);
};

var deepJS = new Workshop("Kyle");

deepJS.ask = function(question){
    this.__proto__.ask.call(this,question.toUpperCase());
};

deepJS.ask("Is this fake polymorphism?");
// Kyle IS THIS FAKE POLYMORPHISM?
```

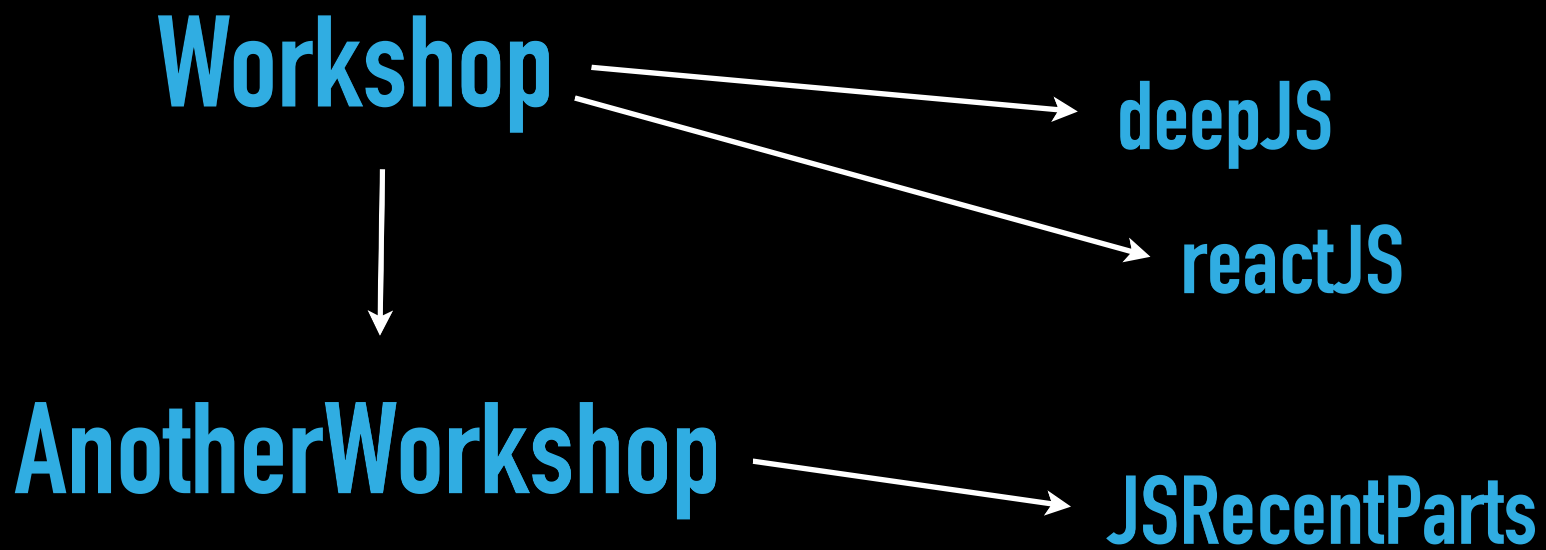
Prototypes: shadowing

“Prototypal Inheritance”

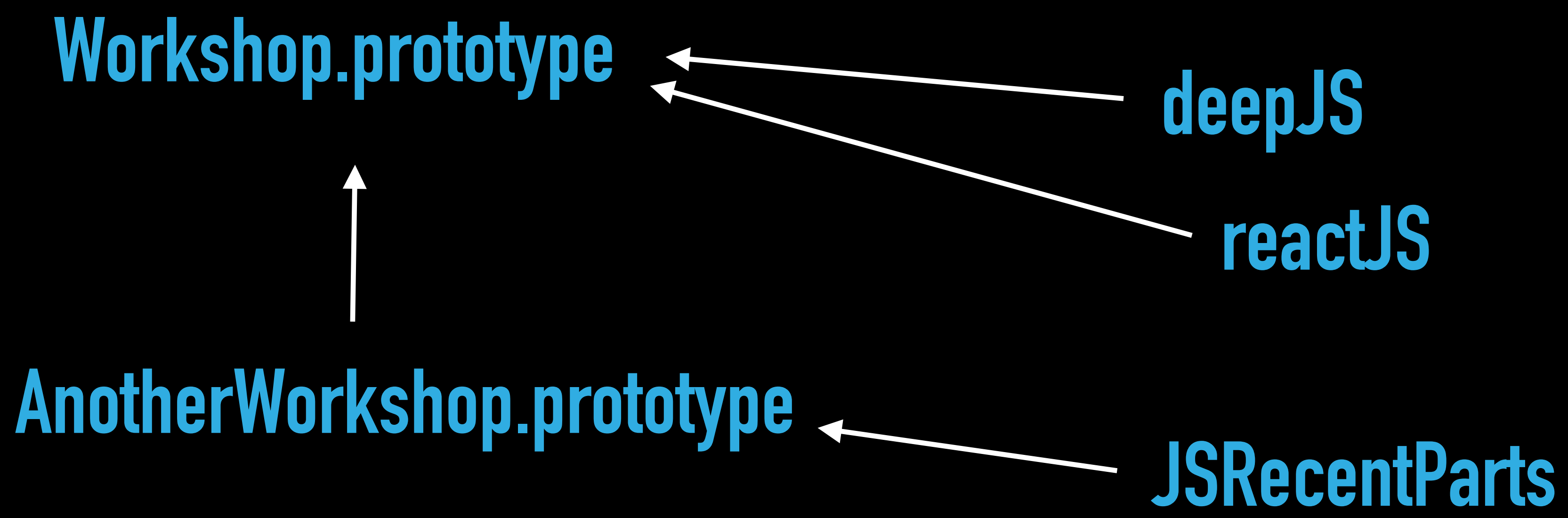
```
function Workshop(teacher) {
    this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
    console.log(this.teacher,question);
};

function AnotherWorkshop(teacher) {
    Workshop.call(this,teacher);
}
AnotherWorkshop.prototype = Object.create(Workshop.prototype);
AnotherWorkshop.prototype.speakUp = function(msg){
    this.ask(msg.toUpperCase());
};

var JSRecentParts = new AnotherWorkshop("Kyle");

JSRecentParts.speakUp("Is this actually inheritance?");
// Kyle IS THIS ACTUALLY INHERITANCE?
```

Prototypes: objects linked

Clarifying Inheritance



OO: classical inheritance



(another design pattern)

OO: “~~prototypal~~ inheritance”

JavaScript ~~“Inheritance”~~

“Behavior Delegation”

Let's Simplify!

OLOO:

Objects Linked to Other Objects

```javascript
class Workshop {
    constructor(teacher) {
        this.teacher = teacher;
    }
    ask(question) {
        console.log(this.teacher,question);
    }
}

class AnotherWorkshop extends Workshop {
    speakUp(msg) {
        this.ask(msg);
    }
}

var JSRecentParts = new AnotherWorkshop("Kyle");

JSRecentParts.speakUp("Are classes getting better?");
// Kyle Are classes getting better?
```

OLOO: recall class?

```javascript
function Workshop(teacher) {
    this.teacher = teacher;
}
Workshop.prototype.ask = function(question){
    console.log(this.teacher,question);
};
function AnotherWorkshop(teacher) {
    Workshop.call(this,teacher);
}
AnotherWorkshop.prototype = Object.create(Workshop.prototype);
AnotherWorkshop.prototype.speakUp = function(msg){
    this.ask(msg.toUpperCase());
};

var JSRecentParts = new AnotherWorkshop("Kyle");
JSRecentParts.speakUp("Isn't this ugly?");
// Kyle ISN'T THIS UGLY?
```

OLOO: prototypal objects

```js
var Workshop = {
	setTeacher(teacher) {
		this.teacher = teacher;
	},
	ask(question) {
		console.log(this.teacher,question);
	}
};
var AnotherWorkshop = Object.assign(
	Object.create(Workshop),
	{
		speakUp(msg) {
			this.ask(msg.toUpperCase());
		}
	}
);

var JSRecentParts = Object.create(AnotherWorkshop);
JSRecentParts.setTeacher("Kyle");
JSRecentParts.speakUp("But isn't this cleaner?");
// Kyle BUT ISN'T THIS CLEANER?
```

OLOO: delegated objects

```
if (!Object.create) {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

OLOO: Object.create()

Delegation: Design Pattern

AuthControllerClass

↓

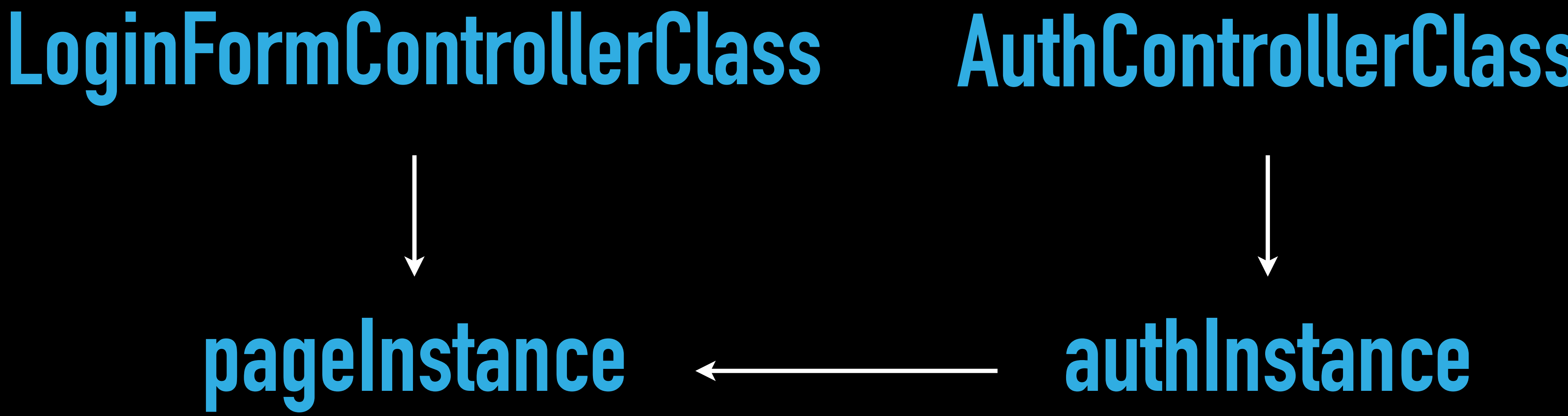
LoginFormControllerClass

↓

pageInstance

Composition Thru Inheritance

Composition Over Inheritance



Mixin Composition

LoginFormController → AuthController

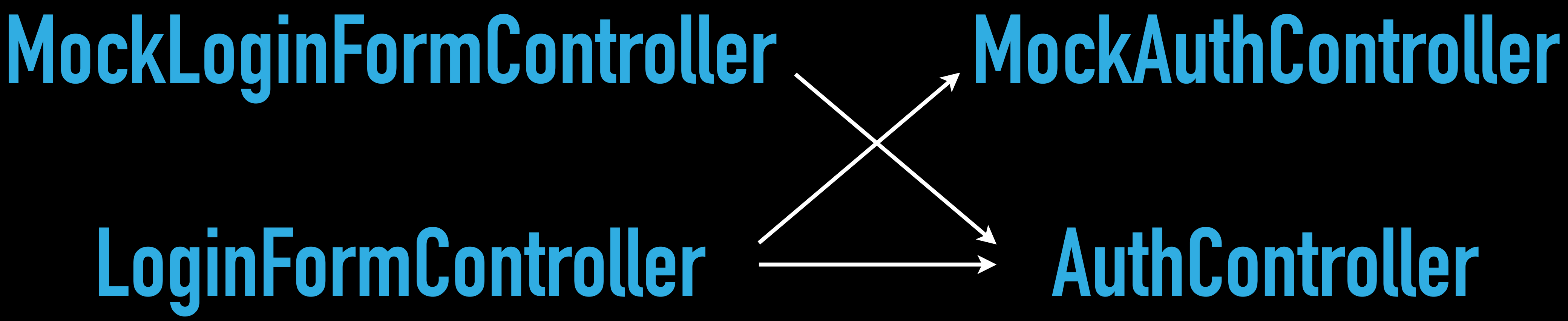
Delegation (Dynamic Composition)

~~Parent–Child~~ Peer–Peer

```
var AuthController = {
    authenticate() {
        server.authenticate(
            [ this.username, this.password ],
            this.handleResponse.bind(this)
        );
    },
    handleResponse(resp) {
        if (!resp.ok) this.displayError(resp.msg);
    }
};

var LoginFormController =
    Object.assign(Object.create(AuthController),{
        onSubmit() {
            this.username = this.$username.val();
            this.password = this.$password.val();
            this.authenticate();
        },
        displayError(msg) {
            alert(msg);
        }
    });
```

Delegation-Oriented Design

More Testable



Know Your JavaScript

THANKS!!!!

KYLE SIMPSON GETIFY@GMAIL.COM

DEEP JS FOUNDATIONS