
JS

HERE WE WILL FOCUS ON
THE LANGUAGE ITSELF AND
THE VM BEHAVIOR SPECIFIED BY ECMASCRIPT STANDARD

We are not talking about the host-specific
features (things provided from browser or node for example)

Basic Concepts About Variables

Definition

A variable is a named container for a ***value***

The ***name*** that refers to a variable
is sometime called *an* ***identifier***

```
var x;
var y = "Hello JS!";
var z;
```

COPY

These red boxes are ***variables***, and each of them has a ***name (identifier)***

Any ***JavaScript value*** can be contained within these boxes

```
var x;
var y = "Hello JS!";
var z;
z = false;
z = 101;
```

We can ***assign*** another ***value*** to a variable later after its creation

Reserved Words

Some keywords can not be used as variable names:

```
null true false break do instanceof typeof
case else new var catch finally return void
continue for switch while debugger function
this with default if throw delete in try
class enum extends super const export import

implements let private public yield
interface package protected static
```

We don't need to remember them all. Just be aware of the possible cause for some SyntaxError exceptions in our program.

Basic Concepts About Values & Types

Definition

A value represents the most basic data we can deal with

value

A type is a ***set*** of data values, and there are exactly 6 types

(7 from ES6)

Type :: { v1 , v2 , v3 , ... }

There are 5 primitive (non-Object) types

6 from ES6 (Symbol is new type)

Undefined `undefined`

Null `null`

Boolean `true` `false`

Number `.33` `-3.14` `011` `7e-2` `-Infinity` `0x101` `NaN`

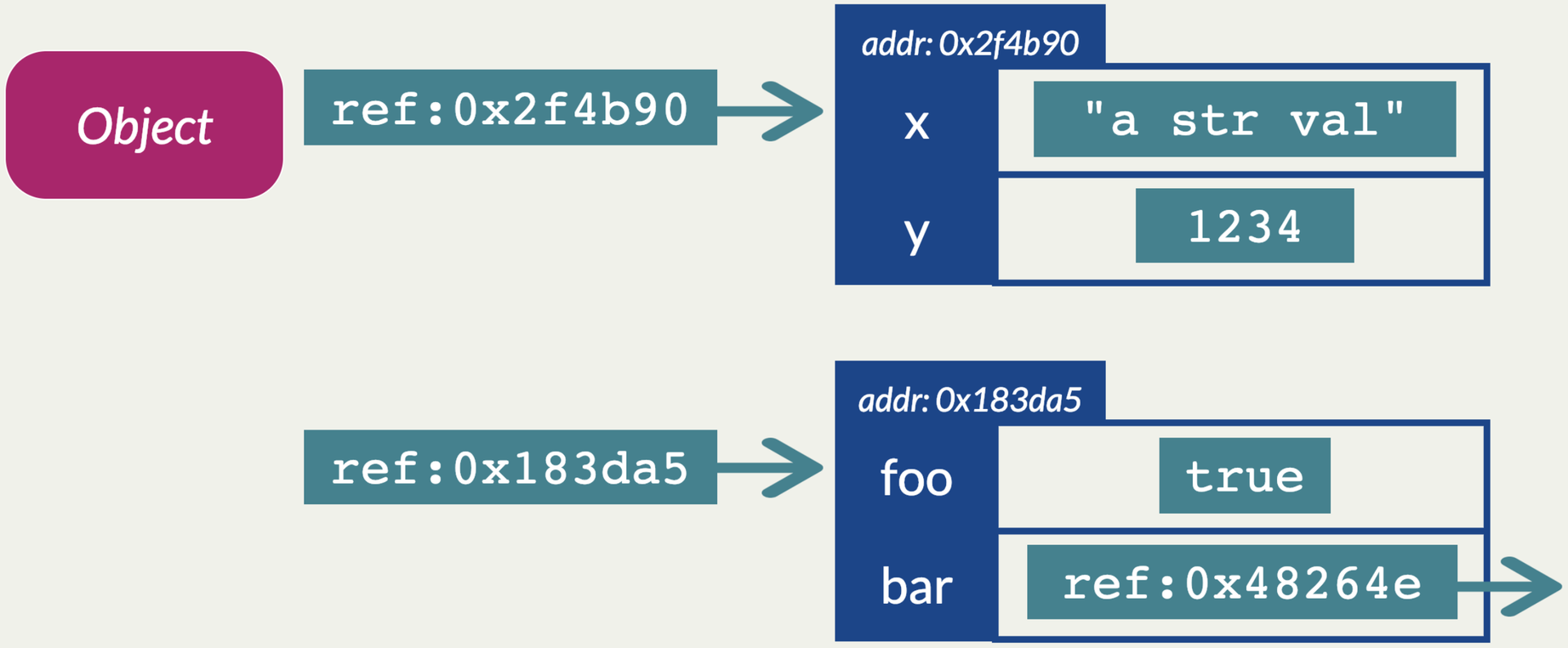
(IEEE754 64-bit doubles)

String `""` `"Hello world!"` `"哈囉。"` `"\n"` `"\""` `'w Single Quotes'`

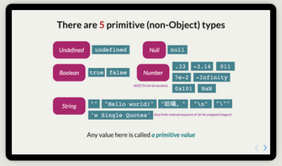
(Any finite ordered sequence of 16-bit unsigned integers)

Any value here is called ***a primitive value***

And then there is the "Object" type



Any value of this type is ***a reference to some "object"***;
sometimes we would simply call such value ***an object***



Definition

An object is a
collection of ***properties***

A property is a
named container for a ***value***

w/ some additional attributes

Definition

The ***name of a property*** is called ***a key***;
thus, ***an object*** can be considered as
a collection of key-value pairs.

There are similar concepts in other programming languages,
e.g., *Map*, *Dictionary*, *Associative Array*, *Symbol Table*, *Hash Table*, ...

Object Initialiser (Object Literal)

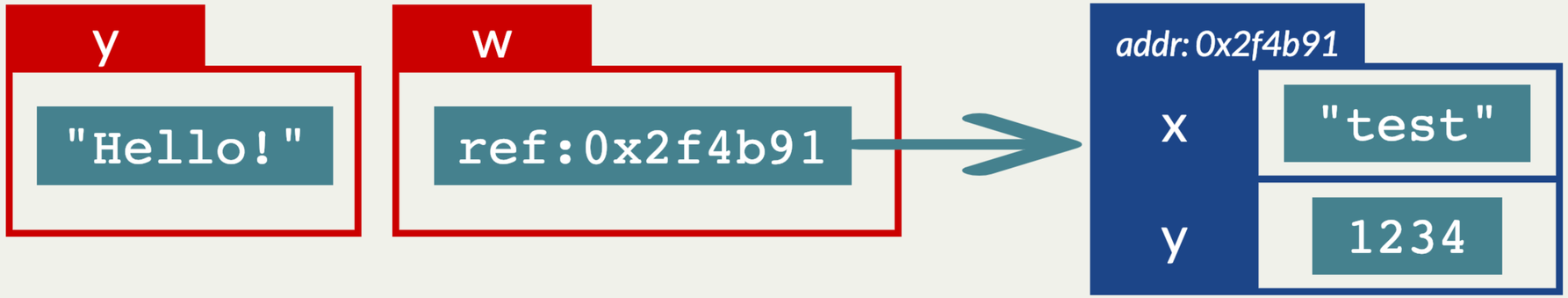
The notation using a pair of curly braces
to ***initialize*** a new JavaScript object.

```
var w = {
    x: "test",
    y: 1234,
    z: {},
    w: {},
    "": "hi"
};
```

```
var w = new Object();
w.x = "test";
w.y = 1234;
w.z = new Object();
w.w = new Object();
w[""] = "hi";
```

The code on the left-hand side has exactly the
same result as the one on the right-hand side

A “*variable*” vs a “*property*” in an object




```
// Value containers
var y = "Hello!";
var w = {
    x: "test",
    y: 1234
};
```

```
// To get the values
y;       // "Hello!"
w;       // (the object ref)
w.x;     // "test"
w['x']; // "test"
w.y;     // 1234
w["y"]; // 1234
```

Add/Get/Set/Remove A Property

We can dynamically modify an object after its creation

```
var obj = {
    1   : "Hello",
    "3": "Good",
    x   : "JavaScript",
    foo: 101,
    bar: true,
    ""  : null
};

obj["2"] = "World";      // *1 Add & Set
obj["1"];                // *2 Get        -> "Hello"
obj[2];                  // *3 Get        -> "World"
obj[3];                  // *4 Get        -> "Good"
obj.foo = 202;           // *5 Set
delete obj.bar;          // *6 Remove
delete obj[""];          // *7 Remove
```

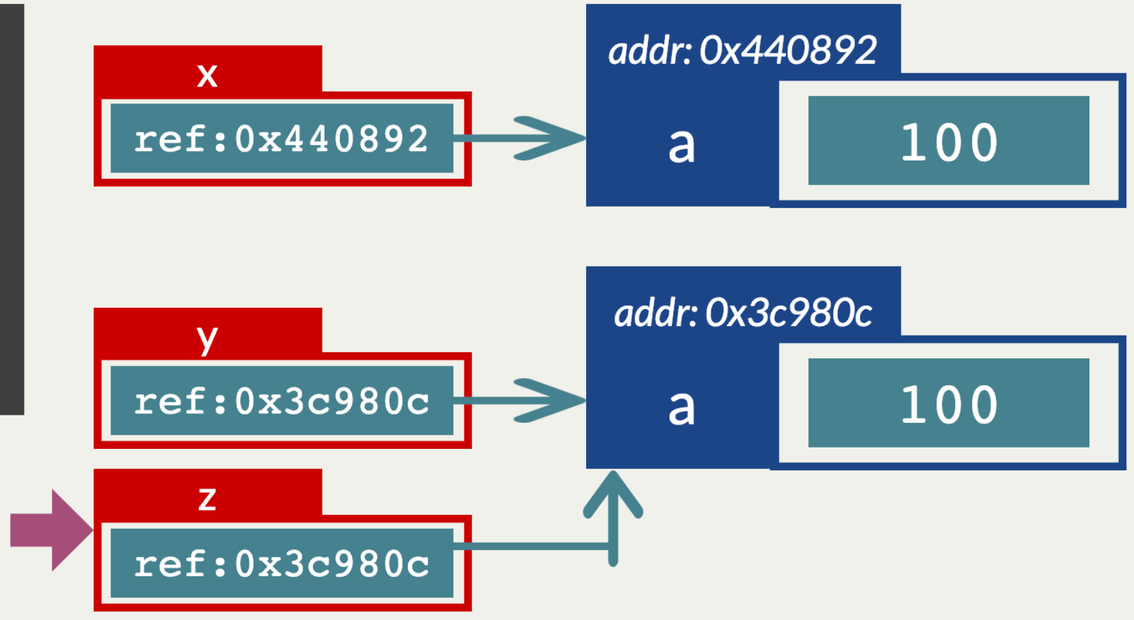
Don't Forget Any Value Of The Object Type Is Actually A “Reference”

```
var x = { a: 100 };
var y = { a: 100 };
```

Similar to the “pointer” / “address” concept
in programming languages like C or C++

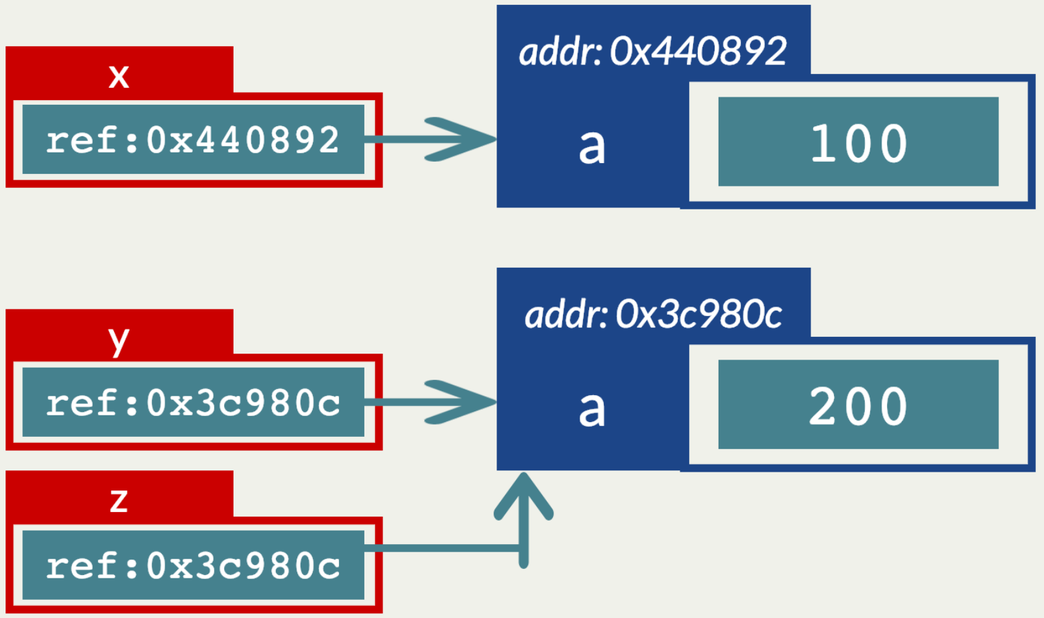
Don't Forget Any Value Of The Object Type Is Actually A “Reference”

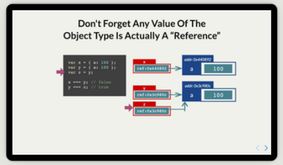
```
var x = { a: 100 };
var y = { a: 100 };
var z = y;

x === y; // false
y === z; // true
```



Don't Forget Any Value Of The Object Type Is Actually A “Reference”

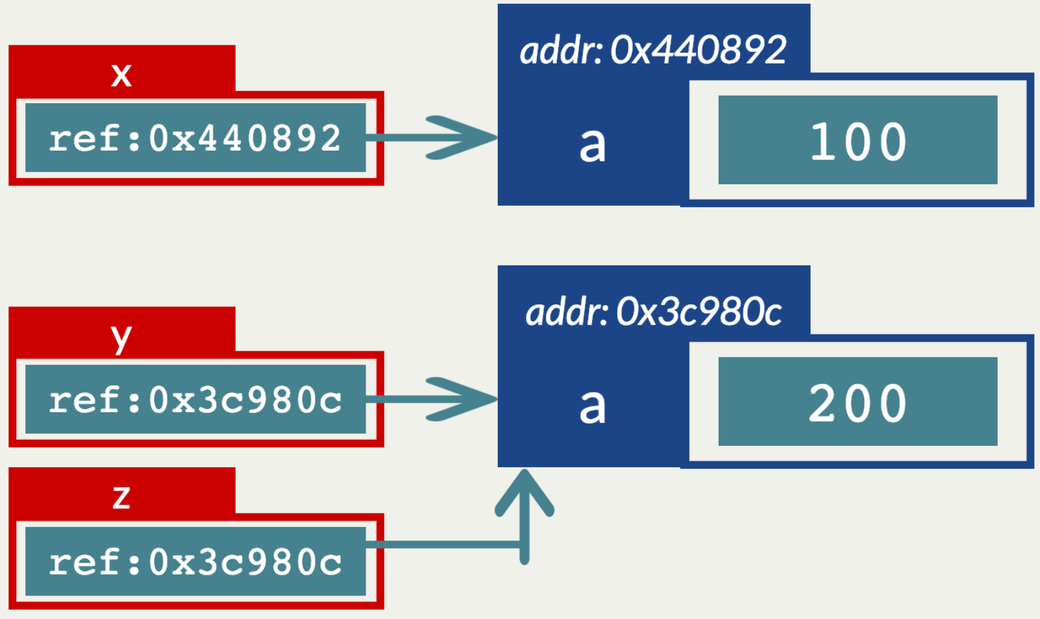
```
var x = { a: 100 };
var y = { a: 100 };
var z = y;

x === y; // false
y === z; // true


z.a = 200;
```

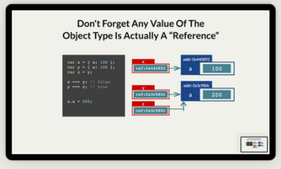




Don't Forget Any Value Of The Object Type Is Actually A “Reference”

```
var x = { a: 100 };
var y = { a: 100 };
var z = y;

x === y; // false
y === z; // true


z.a = 200;

x.a; // 100
y.a; // 200
z.a; // 200
```





Definition

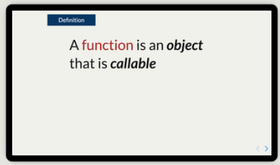
A function is an ***object*** that is ***callable***

Function Expression

The notation using the keyword "***function***"
followed by an ***argument list*** and a ***code block***
to ***create/initialize*** a JavaScript ***Function*** object

```
var a = 7;
var sayhi = function (name) {
    var a = "Hello " + name;
    console.log(a);
    return a;
};
```

Any function in JavaScript is ***first-class***, which can be assigned
to a variable or passed as an ordinary value to another function.



Function Expression Can Be *Named*

```
var fact = function (n) {
    if (n === 0) return 1;
    if (n > 0) return n * fact(n - 1);              // *1
};
fact(3);      // 6, which is the factorial of 3

var fact2 = fact;
fact = null; // But if the value changes...
fact2(3);    // TypeError: "null" is not function



var f = function fib(n) {
    if (n === 0) return 0;
    if (n === 1) return 1;
    if (n > 1) return fib(n - 1) + fib(n - 2);      // *2
};
f(10);        // 55

var g = f;
f = null;     // If the value changes...
g(10);        // 55  (still working)
```

COPY

The name "fib" is accessible only inside the function itself

Function Invocation

A pair of parentheses invokes the preceding function;
the values of zero or more arguments are passed in.

Each time a function is called/invoked, it has ***its own variable scope***
with ***local variables*** and ***arguments*** filled with corresponding values

```
var a = 7;
var sayhi = function (name) {
    var a = "Hello " + name;
    console.log(a);
    return a;
};

sayhi("J"); // "Hello J"
a; // 7
```

Function Not Returning A Value

A function does ***not*** necessarily
need to ***return a value explicitly***.

When that is the case, the "***undefined***" value is ***returned***.

```
var f = function () {
    return;
};

var g = function () {
};

var foo = f();
foo; // undefined
g(); // undefined
```

When # of Arguments Not Matched

In JavaScript, it is OK to invoke a function
with a ***mismatched number of arguments***.

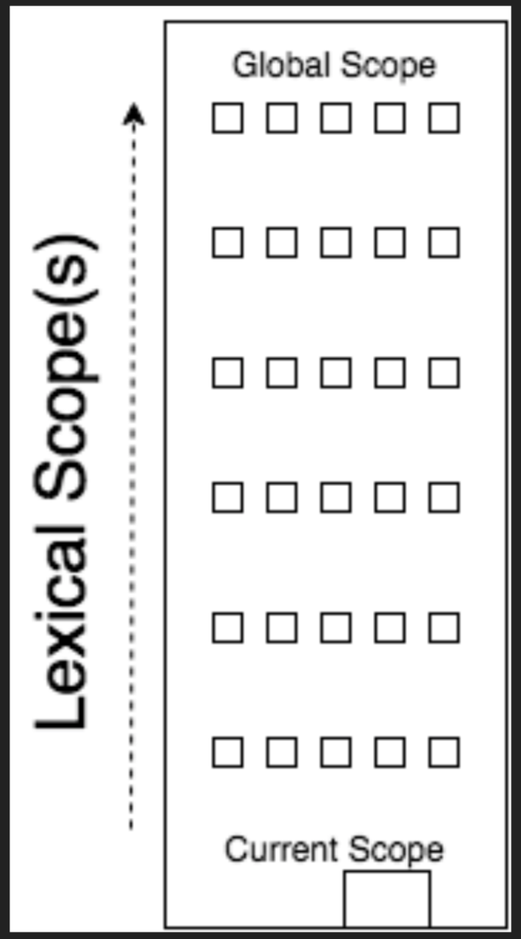
At run time, any ***argument without passing a value*** is filled with the "***undefined***" value, just like a local variable without any assignment.

```
var f = function (a, b) {
    console.log(typeof b);
    return a + b;
};

f(3);        // NaN ("undefined" printed)
f(3, 4);     // 7   ("number" printed)
f(3, 4, 5); // 7   ("number" printed)
```

Scope, Closures

- Nested Scope
- Hoisting
- Closure
- Modules

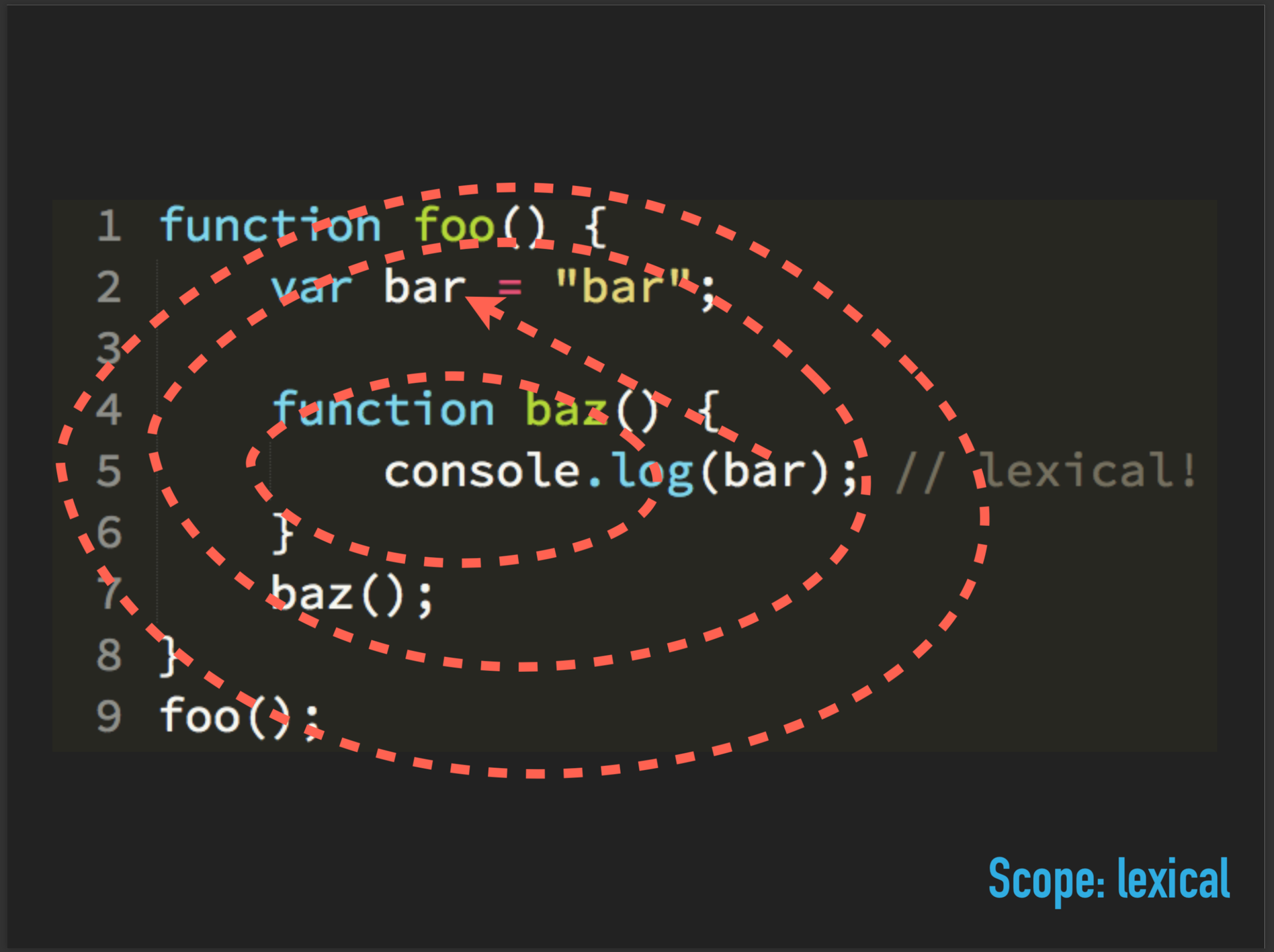
Scope: where to look for things



Scope

```
1 var foo = "foo";
2
3 function bob(){
4     var foo = "foo2";
5     console.log(foo);    // "foo2"
6 }
7 bob();
8
9 console.log(foo);    // "foo" -- phew!
```

Function Scoping


1 function foo() {
2 var bar = "bar";
3
4 function baz() {
5 console.log(bar); // lexical!
6 }
7 baz();
8 }
9 foo();
Scope: lexical

```
1 function diff(x,y) {
2         if (x > y) {
3                 var tmp = x;
4                 x = y;
5                 y = tmp;
6         }
7
8         return y - x;
9 }
```

Block Scoping: intent

```
1 function diff(x,y) {
2     if (x > y) {
3         let tmp = x;
4         x = y;
5         y = tmp;
6     }
7
8     return y - x;
9 }
```

Block Scoping: let

```
1 for (var i = 0; i < 5; i++) {
2     (function IIFE(){
3         var j = i;
4         console.log(j);
5     })();
6 }
```

Function Scoping: IIFE

```
1  var foo;
2
3  try {
4      foo.length;
5  }
6  catch (err) {
7      console.log(err); // TypeError
8  }
9
10 console.log(err); // ReferenceError
```

Scope: which scope?

Quiz

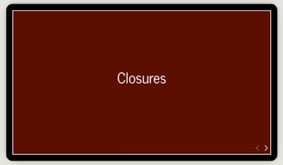
1. What type of scoping rule(s) does JavaScript have?
2. What are 3 different ways you can create a new scoped variable?
3. What's the difference between undeclared and undefined?

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

Closure

Definition

A closure is a ***pair*** of
a ***function*** and an ***environment***
for resolving *free variables*

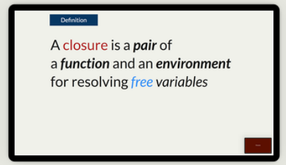


The Lexical Environment (1/3)

When we use ***non-local variables***...

COPY

```
var a = 1;
var closureFunc = function () {
    // The "environment" (the outer scope)
    // determines to which value does the
    // symbol `a` resolve here
    console.log(a);
    return a;
};
```



The Lexical Environment (2/3)

When we use ***non-local variables***...

COPY

```
var a = 1;
var closureFunc = function () {
    // The "environment" (the outer scope)
    // determines to which value does the
    // symbol `a` resolve here
    console.log(a);
    return a;
};


closureFunc(); // 1

a = 100;
closureFunc(); // 100

a = 'hello';
closureFunc(); // 'hello'
```

The Lexical Environment (3/3)

The "***environment***" is determined at ***creation-time***, not at run-time

Lexical scope

Which ***variable*** is the identifier "***a***" here ***bound to*** when the function is invoked?

Dynamic scope (not for JavaScript)

COPY

```
var a = 1;
var closureFunc = function () {
    // The "environment" (the outer scope)
    // determines to which value does the
    // symbol `a` resolve here
    console.log(a);
    return a;
};


var anotherFunc = function () {
    var a = 2;
    return closureFunc();
};

anotherFunc(); // 1
```

JavaScript Uses Static/Lexical Scope Model

How a variable/identifier/symbol inside a function ***will be resolved*** for all function calls in the future is completely determined at the ***creation-time*** of the function already

except for "***this***".

Lifetime of A Lexical Scope/Environment

The ***lifetime*** of the ***local variables*** (the environment/scope) does ***not*** necessarily end as the ***function*** returns. If there is any local variable referenced by a ***returned function***, a ***closure*** is created. Only the ***returned functions*** can access those ***hidden private states***.

```
var x = 999;

var f = function () {
    var x = 222;                // *1
    return function () {
        x += 100;
        return x;               // *2
    };
};

var g = f();                    // *3
g();        // 322              // *4
g();        // 422              // *5
```

COPY

Common Way To Create A Closure (1/3)

A function that ***returns*** a ***closure function***...

```
var getClosureFunc = function () {

    var a = 1;
    var closureFunc = function () {
        // The "environment" (the outer scope)
        // determines to which value does the
        // symbol `a` resolve here
        console.log(a);
        return a;
    };

    return closureFunc;
};


var a = 3;
var clsr = getClosureFunc();

clsr(); // 1
```

COPY

A ***closure*** is created each time **getClosureFunc** returns.

Common Way To Create A Closure (2/3)

A function that ***returns*** a ***closure function***...

```
var getClosureFunc = function () {

    var a = 1;
    return function () {
        // The "environment" (the outer scope)
        // determines to which value does the
        // symbol `a` resolve here
        console.log(a);
        return a;
    };


};


var a = 3;
var clsr = getClosureFunc();

clsr(); // 1
```

COPY

A ***closure*** is created each time **getClosureFunc** returns.

Common Way To Create A Closure (3/3)

A function that ***returns*** a ***closure function***... gets ***invoked*** immediately

```
var clsr = (function () {

    var a = 1;
    return function () {
        // The "environment" (the outer scope)
        // determines to which value does the
        // symbol `a` resolve here
        console.log(a);
        return a;
    };


}());


var a = 3;


clsr(); // 1
```

COPY

A ***closure*** is created when the anonymous function returns.

```
1  function foo() {
2      var bar = 0;
3
4      setTimeout(function(){
5          console.log(bar++);
6      },100);
7      setTimeout(function(){
8          console.log(bar++);
9      },200);
10 }
11
12 foo();  // 0 1
```

Closure: shared scope

```
1 for (var i=1; i<=5; i++) {
2     setTimeout(function(){
3         console.log("i: " + i);
4     },i*1000);
5 }
```

Closure: loops

```
1 for (var i=1; i<=5; i++) {
2     (function(i){
3         setTimeout(function(){
4             console.log("i: " + i);
5         },i*1000);
6     })(i);
7 }
```

Closure: loops

Code Me

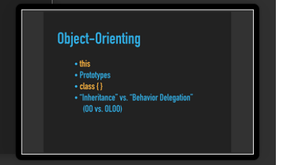
```
1 for (var i=1; i<=5; i++) {
2     let j = i;
3     setTimeout(function(){
4         console.log("j: " + j);
5     },j*1000);
6 }
```

Closure: loops + block scope

```
1 for (let i=1; i<5; i++) {
2       setTimeout(function(){
3           console.log("i: " + i);
4       },i*1000);
5 }
```

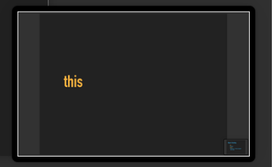
Closure: loops + block scope

this


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

Every* function, while executing, has a reference to its current execution context, called this.

this



```
1  function foo() {
2      console.log(this.bar);
3  }
4
5  var bar = "bar1";
6  var o2 = { bar: "bar2", foo: foo };
7  var o3 = { bar: "bar3", foo: foo };
8
9  foo();                  // "bar1"
10 o2.foo();               // "bar2"
11 o3.foo();               // "bar3"
```

Code Me

this: implicit & default binding

```
1 function foo() {
2     console.log(this.bar);
3 }
4
5 var bar = "bar1";
6 var obj = { bar: "bar2" };
7
8 foo();                  // "bar1"
9 foo.call(obj);          // "bar2"
```

Code Me

this: explicit binding

```
function something() {
    this.hello = "hello";
    console.log(this.hello,this.who);
}

var who = "global", foobar, bazbam,
    obj1 = { who: "obj1", something: something },
    obj2 = { who: "obj2" };

something(); // "hello"  "global"
console.log(hello); // "hello"          <-- OOPS!!!

obj1.something(); // "hello"  "obj1"
console.log(obj1.hello); // "hello"

obj1.something.call(obj2); // "hello"  "obj2"
console.log(obj2.hello); // "hello"

foobar = something.bind(obj2);
foobar(); // "hello"  "obj2"
foobar.call(obj1); // "hello"  "obj2"  <-- STILL "obj2"

bazbam = new something(); // "hello"  undefined
console.log(bazbam.hello); // "hello"

bazbam = new obj1.something(); // "hello"  undefined
bazbam = new foobar(); // "hello"  undefined  <-- LOOK, not "obj2"
```

this: order of 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

```
1  function foo() {
2      return () => this.bar;
3  }
4
5  var bar = "bar1";
6  var o1 = { bar: "bar2", foo: foo };
7  var o2 = { bar: "bar3" };
8
9  var f1 = foo();
10 var f2 = o1.foo();
11 var f3 = foo.call(o2);
12
13 f1();              // "bar1"
14 f2();              // "bar2"
15 f3();              // "bar3"
16
17 f1.call( o2 );  // "bar1"    <--- hmmmm
```

this: what about => functions?

Quiz

1. How do you “borrow” a function and implicitly set this?
2. How do you explicitly set this for the function call?
3. How can you lock a specific this to a function?

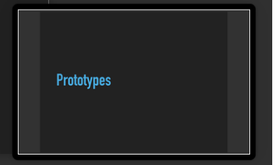
Why do that? Why not?

4. How do you create a new this for the function call?

this

Prototypes

Objects are built by constructor calls



A constructor makes an object linked to its own prototype

Definition

A constructor is a ***function*** that has a ***property*** named "***prototype***" which can be used to implement *inheritance* and *shared properties*

The value for the "***prototype***" property is either ***null*** or ***an object***

Native Functions

- String()
- Number()
- Boolean()
- Function()
- Object()
- Array()
- RegExp()
- Date()
- Error()

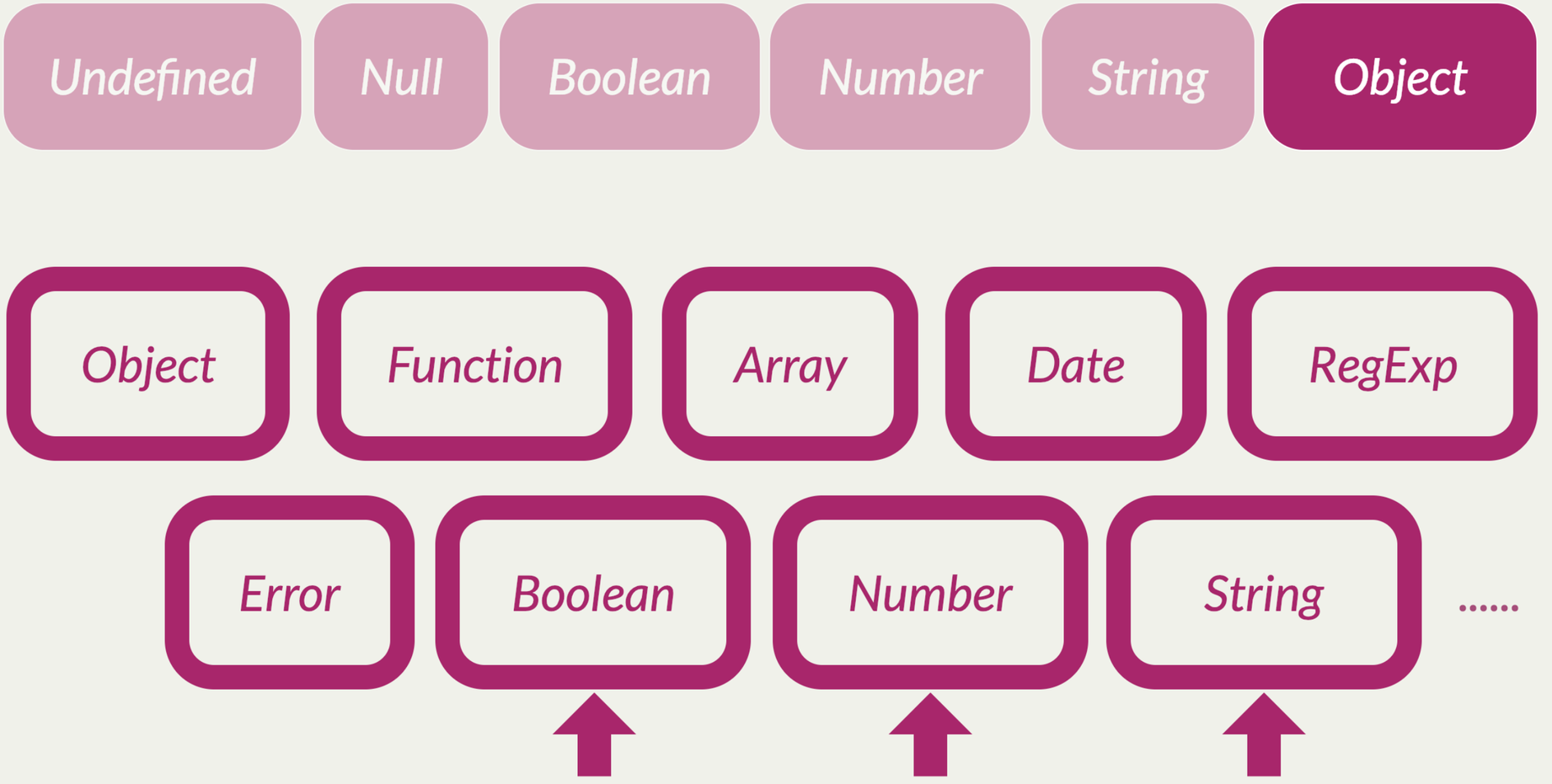
Natives

“Object” Type Can Be Further *Categorized*

Undefined | *Null* | *Boolean* | *Number* | *String* | *Object*

Object | *Function* | *Array*

These 3 Are Used By *ToObject* Conversion



Since there are ***wrapper objects*** for the 3 types of primitive values,
a Boolean/Number/String primitive value can ***behave like an object***

Let's play with some ***primitive*** values...

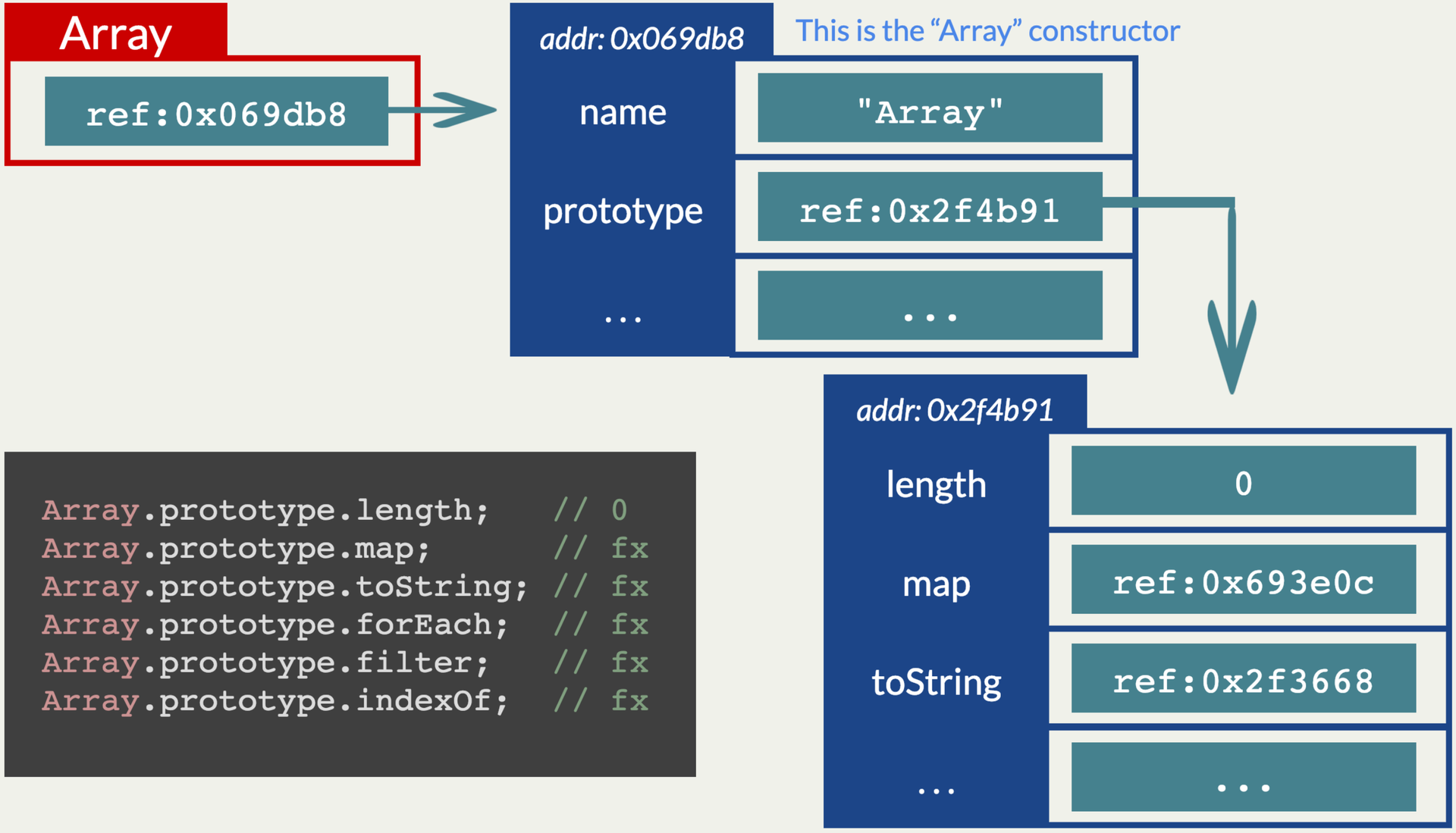
```
( true  ).toString(); // "true"
( false ).toString(); // "false"

( 3.14 ).toString();  // "3.14"
( 3.14 ).toFixed();   // "3"

( "Hello" ).length;              // 5
( "Hello" ).toUpperCase();       // "HELLO"
( "HELLO" ).slice(1, -1);        // "ELL"
( "Hello world" ).split("o"); // [ "Hell", " w", "rld" ]
```

Where do those ***properties*** and ***methods*** come?

Note: Check the **code snippets** for more about "***String***" values

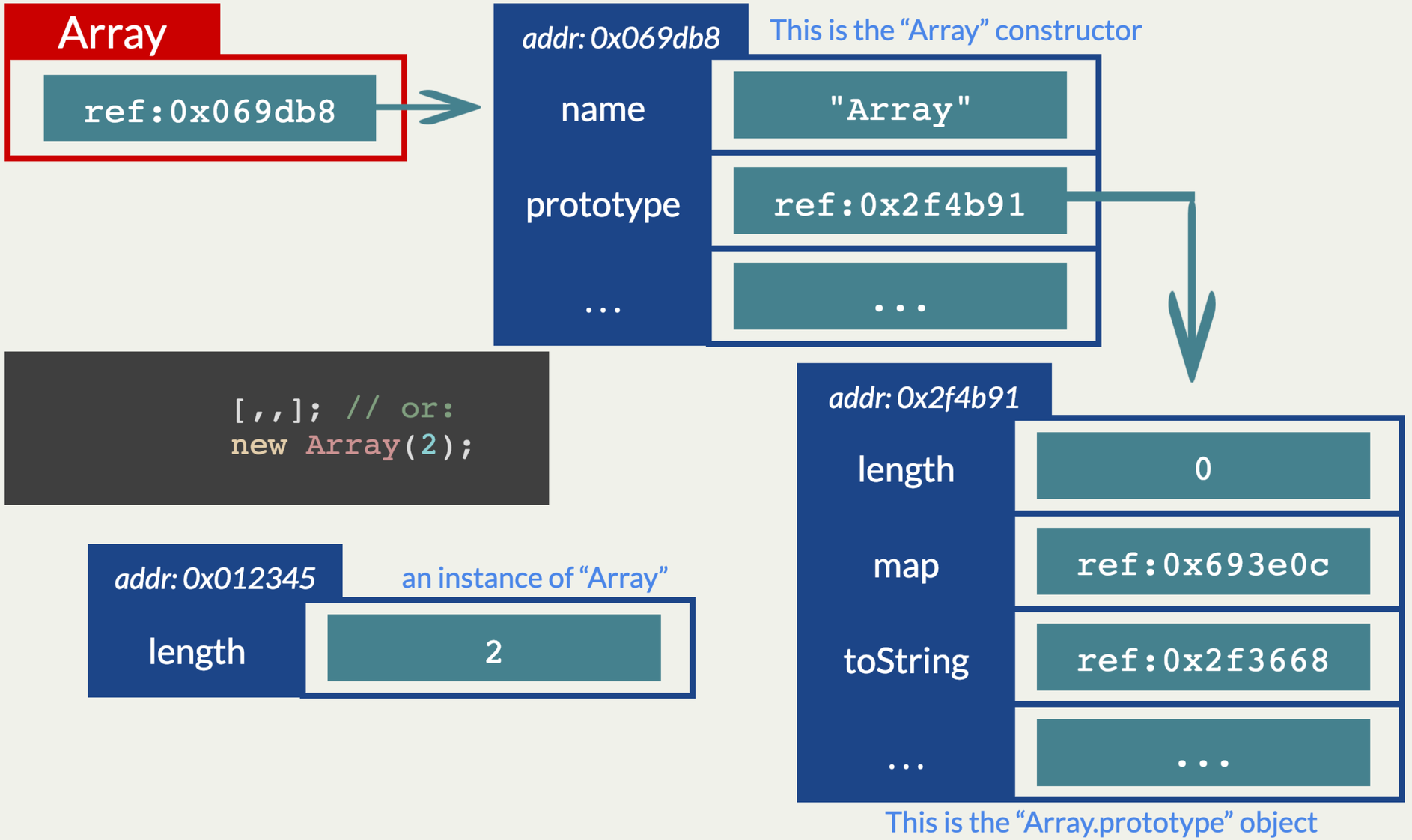
Let's take a look at ***how*** an ***object*** is ***created***

As an example, assume that we want to have
a new object instance of the "***Array***" category...

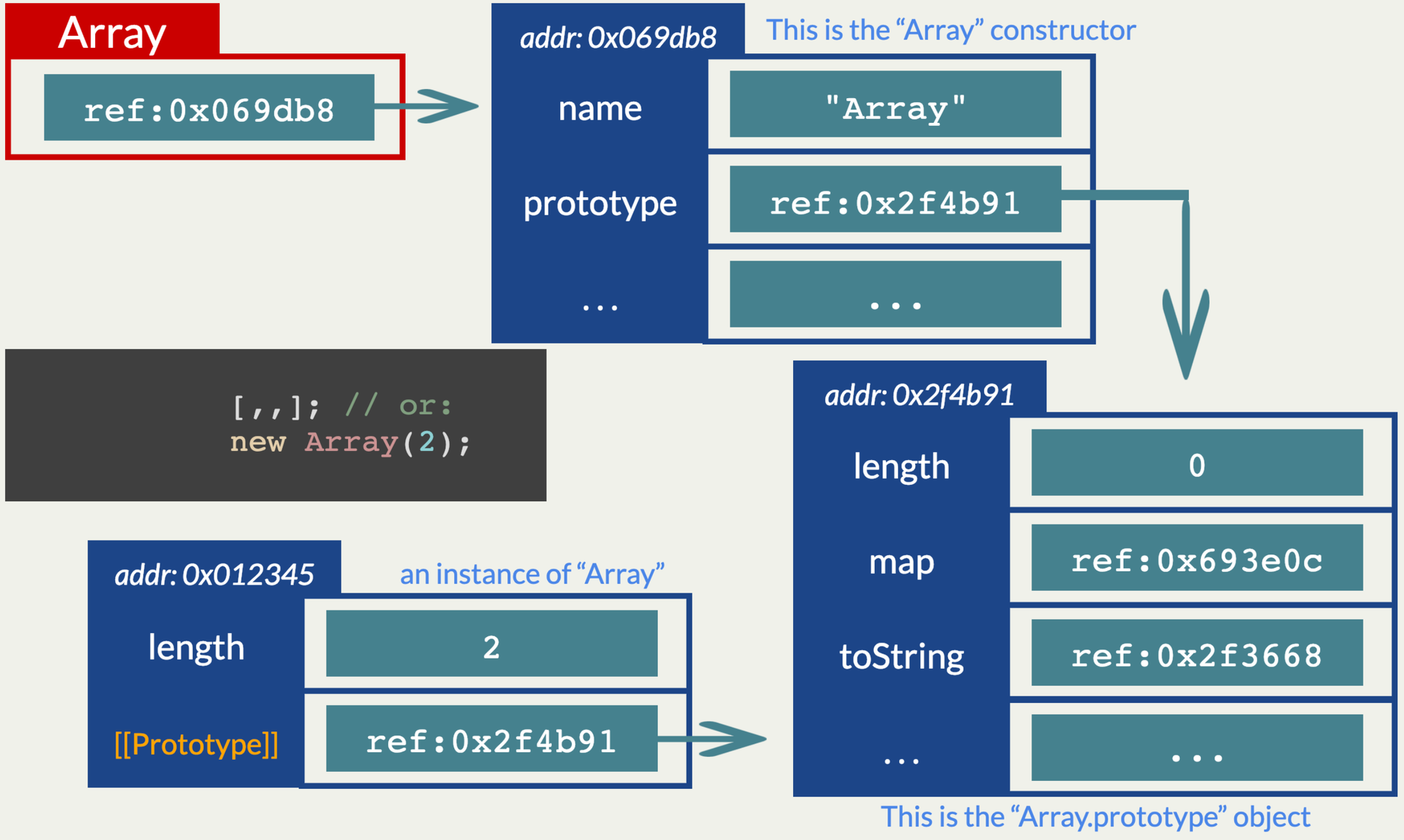



```
Array.prototype.length;    // 0
Array.prototype.map;       // fx
Array.prototype.toString;  // fx
Array.prototype.forEach;   // fx
Array.prototype.filter;    // fx
Array.prototype.indexOf;   // fx
```

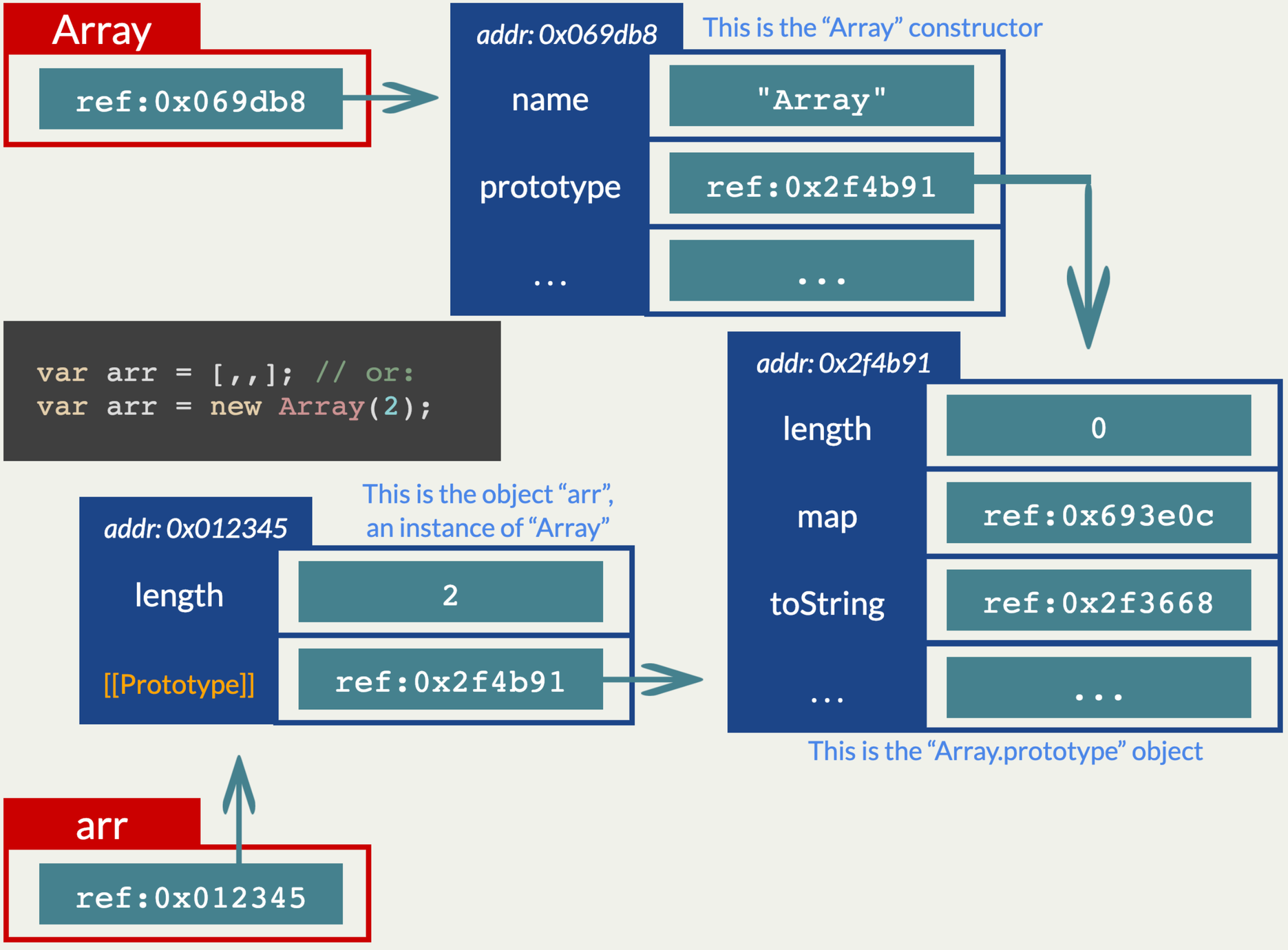
The built-in "***Array***" constructor/function/object is used to construct a new array, and the object "***Array.prototype***" already contains many useful properties by default.



When an “***initialiser***” or “***new***” occurs, an object is created with some properties depending on the constructor.
For example, the “***length***” property is set by “***Array***”.



Then, something called “***the prototype***” of the object is set to whatever “***Array.prototype***” is. Here we denote it as an internal property named “**[[Prototype]]**”.


Array
ref:0x069db8
addr: 0x069db8
This is the "Array" constructor
name
"Array"
prototype
ref:0x2f4b91
...
...
var arr = [,,]; // or:
var arr = new Array(2);
addr: 0x2f4b91
length
0
map
ref:0x693e0c
toString
ref:0x2f3668
...
...
This is the "Array.prototype" object
addr: 0x012345
This is the object "arr",
an instance of "Array"
length
2
[[Prototype]]
ref:0x2f4b91
arr
ref:0x012345

Array Constructor As An Example

```
var arr = [ 11, 22 ];

arr.length;                          // 2
arr.toString();                      // "11,22"
arr.map(function(n){return n+1});    // [ 12, 23 ]


arr.hasOwnProperty('length');        // true
arr.hasOwnProperty('toString');      // false
arr.hasOwnProperty('map');           // false
Array.prototype.hasOwnProperty('toString'); // true
Array.prototype.hasOwnProperty('map');      // true
```

COPY

For the object ***arr***, "***length***" is an own property, but "***toString***" and "***map***" are not.

Remark

We will use “***X's prototype***” to refer to ***the object*** or a null value referenced by the **internal [[Prototype]] property** of “**X**”

Definition

An “***own property***” is a property that is ***directly contained*** by its object

An “***inherited property***” is a property of an object that is ***not*** an ***own property*** but is a property (own or inherited) of the ***object's prototype***

Custom Constructor Function

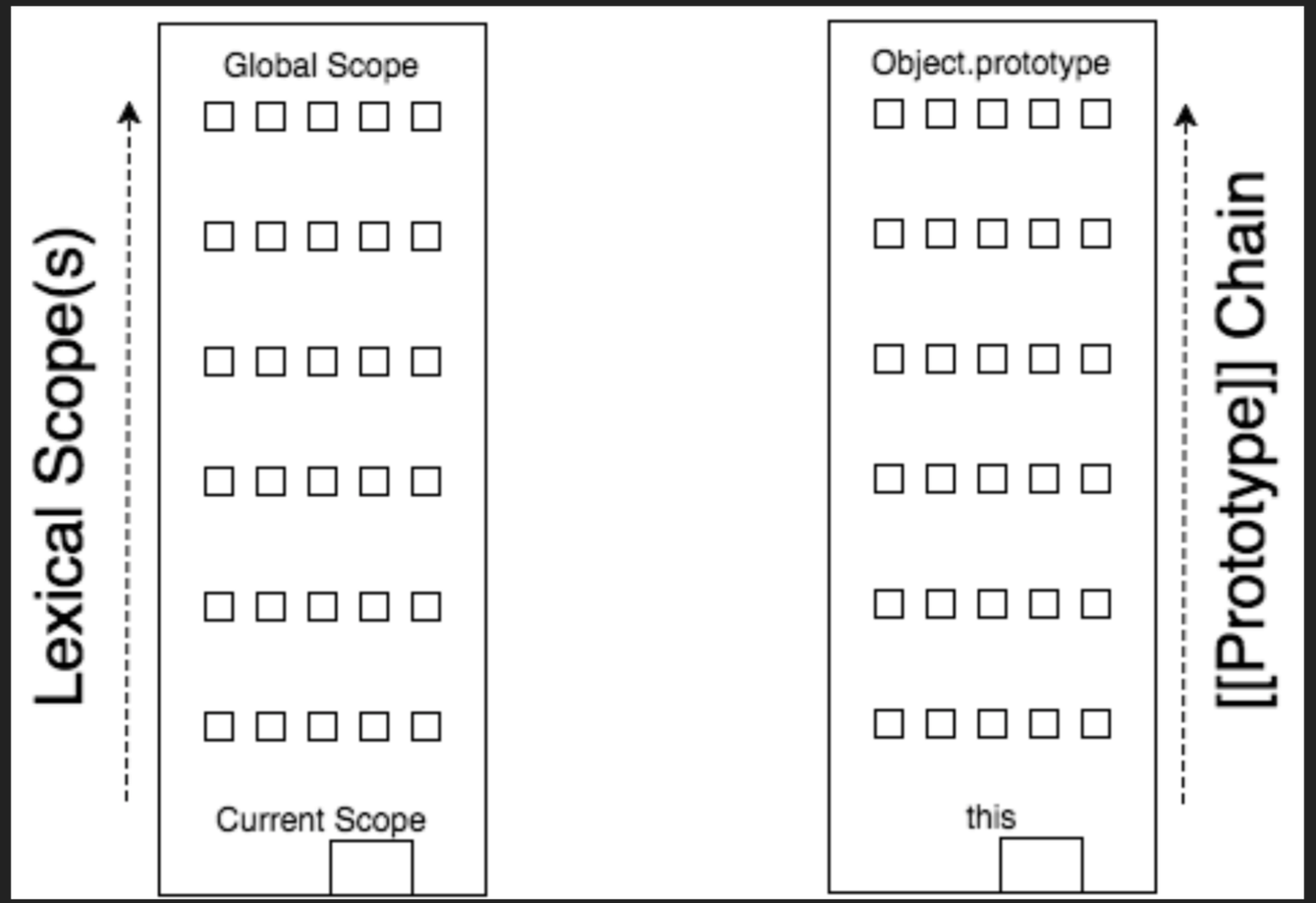
Note the identifier "***this***" is bound to ***the newly created object*** when a function is called as a ***constructor*** using the ***new*** operator.

```
var Cat = function (name) {
    if (name) this.name = name;
};
Cat.prototype.name = 'Nyan Cat';
Cat.prototype.meow = function () {
    return 'Meow~  I am ' + this.name;
};

var pusheen = new Cat('Pusheen the cat');


var nyancat = new Cat;  // () can be omitted when no args

pusheen.meow(); // "Meow~  I am Pusheen the cat"
nyancat.meow(); // "Meow~  I am Nyan Cat"
```

An Easier Way To Create New Objects

We can create new objects "***with specified prototypes***" directly

```
var catPrototype = {
    name: 'Nyan Cat',
    meow: function () {
        return 'Meow~  I am ' + this.name;
    }
};


var pusheen = Object.create(catPrototype);
pusheen.name = 'Pusheen the cat';

var nyancat = Object.create(catPrototype);

pusheen.meow(); // "Meow~  I am Pusheen the cat"
nyancat.meow(); // "Meow~  I am Nyan Cat"
```


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

Prototypes

The Global Object In JavaScript

What Is “The Global Object” For?

“***The global object***” stores all ECMAScript ***standard built-in objects/values*** as well as any ***object provided by the host***.

Each ***global variable*** is stored in “***the global object***” as a property with the corresponding name and value.

For example, the constructors we mentioned previously, e.g., *Object*, *Function*, *Array*, *String*, *Date*, *RegExp*, are all properties of the global object. Most of the host provided APIs like “*setTimeout*” and “*XMLHttpRequest*” are also properties of the global object.

Know Your JavaScript

Hackers **Learn**

By ~~Reading~~

Making Stuff

Find An Idea
To **Work** On
To **Write Code** For

Hackers **Learn**
By ~~Reading~~
Making Stuff