A *scope* walks into a *var*

An overview of scope in JavaScript

Hi, there!
We’re **Ignacio Mena** *@nachomena*
and **Luciano Battagliero** *@battaglr*

Scope

Mechanism that determines **where** and **how** to **look-up** for **identifiers**

An identifier is a **variable** or a **function** name

In other words… scope refers to
who can **see** or **use** an **identifier**

For simplicity's sake let's assume all our code runs in `strict mode`

```
var x = 42;

console.log(x);
```

```
var x = 42;
console.log(x); // → 42
```

```
var x = 42;

console.log(x); // → 42
```

```
var x = 42;

console.log(x); // → 42
```

```
var x = 42;

console.log(x); // → 42
```

That’s just the beginning. Let’s introduce some new concepts!

Scope chain

A **stack** of **currently accessible** scopes, from the most immediate **local** scope up to the **global** scope

To find an **identifier**, a **scope look-up** must be made

A scope look-up **stops** once
it finds the **first match**

```
var y = 40;

function foo() {
    var x = 2;
    return x + y;
}

foo();
```

```
var y = 40;

function foo() {
    var x = 2;
    return x + y;
}

foo(); // → 42
```

```
var y = 40;

function foo() {
    var x = 2;
    return x + y;
}

foo(); // → 42
```

```
var y = 40;

function foo() {
    var x = 2;
    return x + y;
}

foo(); // → 42
```

```
var y = 40;

function foo() {
    var x = 2;
    return x + y;
}

foo(); // → 42
```

Shadowing

An identifier declared within a certain scope that has the same name as one declared in an outer scope

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo();
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo(); // → shadow
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo(); // → shadow
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo(); // → shadow
```

Reference error

A failed attempt to find an identifier anywhere in the scope chain results in a ReferenceError

```
function foo() {
    var x = 2;
    return x + y;
}

foo();
```

```
function foo() {
    var x = 2;
    return x + y;
}

foo(); // → ReferenceError: y is not defined
```

```
function foo() {
    var x = 2;
    return x + y;
}

foo(); // → ReferenceError: y is not defined
```

```
function foo() {
    var x = 2;
    return x + y;
}

foo(); // → ReferenceError: y is not defined
```

```
function foo() {
    var x = 2;
    return x + y;
}

foo(); // → ReferenceError: y is not defined
```

Global scope

The global scope is the **outermost scope** and it’s **automatically created** by the **JavaScript engine**

Identifiers declared globally are **accessible** to the **entire program**

Browser

```
window
```

```
window === this &&
window === self &&
window === frames
window === this &&
window === self &&
window === frames // → true
```

Node

```
global
```

```
global === this
global === this // → true
```

Cross-platform

```
function getGlobal() {
    if (typeof self !== 'undefined') { return self; }
    if (typeof window !== 'undefined') { return window; }
    if (typeof global !== 'undefined') { return global; }

    throw new Error('Unable to locate global object');
};
```

(╯°□°)╯︵ ┻━┻

Luckily there’s a proposal to standardize the global identifier

bit.ly/2xl59tn

┬─┬ノ(° _ °ノ)

It's also possible to **reference** a **global identifier** as a **property** of the **global object**

```
var x = 42;

console.log(x)
console.log(window.x)
```

```
var x = 42;

console.log(x) // → 42
console.log(window.x) // → 42
```

```
var x = 42;

window.console.log(this.x)
```

```
var x = 42;

window.console.log(this.x) // → 42
```

(⊙_⊙)

Remember **shadowing**?

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo();
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return x;
}

foo(); // → shadow
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return window.x;
}

foo();
```

```
var x = 'tree';

function foo() {
    var x = 'shadow';
    return window.x;
}

foo(); // → tree
```

Local scope

Functions and **blocks** have their own **scope** called **local scope**

Identifiers declared **locally** are **only accessible** within that **scope**

Function scope

Back in the dark old days of ***ES5***
only functions created **new scopes**

(ಥ﹏ಥ)

```
function foo() {
    var x = 42;
    console.log(x);
};

foo();

console.log(x);
```

```
function foo() {
    var x = 42;
    console.log(x); // → 42
};

foo();

console.log(x); // → ReferenceError
```

```
function foo() {
    var x = 42;
    console.log(x); // → 42
};

foo();

console.log(x); // → ReferenceError
```

```
function foo() {
    var x = 42;
    console.log(x); // → 42
};

foo();

console.log(x); // → ReferenceError
```

```
function foo() {
    var x = 42;
    console.log(x); // → 42
};

foo();

console.log(x); // → ReferenceError
```

```
function foo() {
    var x = 42;
    console.log(x); // → 42
};

foo();

console.log(x); // → ReferenceError
```

Since there was **no block scope**, we got a bit ~~crazy~~ creative...

IIFE

Immediately Invoked Function Expressions

```
(function () {
    var x = 42;
    console.log(x);
})();

console.log(x);
```

```
(function () {
    var x = 42;
    console.log(x); // → 42
})();

console.log(x); // → ReferenceError
```

```
(function () {
    var x = 42;
    console.log(x); // → 42
})();

console.log(x); // → ReferenceError
```

```
(function () {
    var x = 42;
    console.log(x); // → 42
})();

console.log(x); // → ReferenceError
```

```
(function () {
    var x = 42;
    console.log(x); // → 42
})();

console.log(x); // → ReferenceError
```

```
(function () {
   var x = 42;
   console.log(x); // → 42
})();

console.log(x); // → ReferenceError
```

Block scope

While the bright future of ***ES6*** was **still far away** we had **no hope** and **no block scope**

```
if (true) {
    var x = 42;
    console.log(x);
}

console.log(x);
```

```
if (true) {
    var x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
if (true) {
    var x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
if (true) {
    var x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
if (true) {
    var x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
if (true) {
    var x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
// “Under the hood”

var x;

if (true) {
   x = 42;
   console.log(x);
}

console.log(x);
```

```
// “Under the hood”

var x;

if (true) {
    x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

```
// “Under the hood”

var x;

if (true) {
    x = 42;
    console.log(x);
}

console.log(x);
```

```
// “Under the hood”

var x;

if (true) {
    x = 42;
    console.log(x); // → 42
}

console.log(x); // → 42
```

At some point even **IIFEs were not enough** and some people went totally insane

They discovered another way to create block scope: **try…catch**

```
try {
    throw 42;
} catch(x) {
    console.log(x);
}

console.log(x);
```

```
try {
    throw 42;
} catch(x) {
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

```
try {
    throw 42;
} catch(x) {
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

```
try {
    throw 42;
} catch(x) {
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

```
try {
    throw 42;
} catch(x) {
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

```
try {
   throw 42;
} catch(x) {
   console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

☞(⌐■_■)☞

But no one is crazy enough to write ugly code like that!

It’s mainly used by transpilers…

Luckily those times are past behind us, and ***ES6*** brought **let** and **const**

Which allows us to create
block scope

Without having to go insane! (that’s a good thing)

```
if (true) {
    let x = 42;
    console.log(x);
}

console.log(x);
```

```
if (true) {
    let x = 42;
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

(ノ°∀°)ノ⌒･*:.｡. .｡.:*･゜ﾟ･*☆

Creating a new **block scope** it’s as simple as adding a **pair** of **curly brackets**

```
{
   let x = 42;
   console.log(x);
}

console.log(x);
```

```
{
    let x = 42;
    console.log(x); // → 42
}

console.log(x); // → ReferenceError
```

ヽ(°〇°)ﾉ

Hoisting

JavaScript code is **executed line-by-line** from **top to bottom**

At least it seems that way most of the time…

```
console.log(x);
console.log(x); // → ReferenceError
```

┐(˘、˘)┌

```
console.log(x);

var x;
```

```
console.log(x); // → undefined

var x;
```

(・・)?

```
console.log(x);

var x = 42;
```

```
console.log(x); // → undefined

var x = 42;
```

ლ(ಠ_ಠ ლ)

Yeap… it’s not *that* simple!

Declaration and **assignment statements**

```
var x = 42;
var x;

x = 42;
```

```
var x; // Declaration

x = 42; // Assignment
```

Let’s try to **roughly** understand **how** a JavaScript **engine works**

It has two phases: ***compilation* phase** and **execution phase**

Yeah, it’s an over-simplification! But it will work for the purposes of this presentation!

Declarations are processed during the ***compilation*** **phase**

```
console.log(x);

var x;
```

```
console.log(x); // → undefined

var x;
```

```
// “Under the hood”

var x;

console.log(x);
```

```
// “Under the hood”
var x;
console.log(x); // → undefined
```

Assignments are processed during the **execution phase**

```
console.log(x);

var x = 42;
```

```
console.log(x); // → undefined

var x = 42;
```

```
// “Under the hood”

var x;

console.log(x);

x = 42;
```

```
// “Under the hood”

var x;

console.log(x); // → undefined

x = 42;
```

What about **functions**?

Unlike variables, **function declarations** are hoisted alongside its **definition**

```
foo();

function foo() { return 42; }
```

```
foo(); // → 42

function foo() { return 42; }
```

w(°o°)w

```
// “Under the hood”

function foo() { return 42; }

foo();
```

```
// “Under the hood”

function foo() { return 42; }

foo(); // → 42
```

```
function foo() { return 2; }

foo();

function foo() { return 4; }
function foo() { return 2; }

foo(); // → 4

function foo() { return 4; }
```

```
// “Under the hood”

function foo() { return 2; }
function foo() { return 4; }

foo();
```

```
// “Under the hood”

function foo() { return 2; }
function foo() { return 4; }

foo(); // → 4
```

```
function foo() { return 2; }

foo();

var foo = function () { return 4; };
```

```
function foo() { return 2; }

foo(); // → 2

var foo = function () { return 4; };
```

(งಠ_ಠ)ง

```
// “Under the hood”

function foo() { return 2; }
~~var foo;~~

foo();

foo = function () { return 4; };
```

```
// “Under the hood”

function foo() { return 2; }
~~var foo;~~

foo(); // → 2

foo = function () { return 4; };
```

```
foo();

var foo = function () { return 42; };
```

```
foo(); // → TypeError: foo is not a function

var foo = function () { return 42; };
```

╰(°益°)╯

```
// “Under the hood”
var foo;
foo();
foo = function () { return 42; };
```

```
// “Under the hood”
var foo;
foo(); // → TypeError: foo is not a function
foo = function () { return 42; };
```

Those examples were just a simple analogy to illustrate how the engine phases work; code is never really “moved” anywhere

Okay… at least **let** and **const** behave like **var**, right?

```
console.log(x);

let x = 42;
```

```
console.log(x); // → ReferenceError

let x = 42;
```

ಠ_ಠ

Hoisting applies, but it’s **inaccessible until** the point where the variable it’s **actually declared**

```
let x = 2;

{
    console.log(x);
    let x = 4;
}
```

```
let x = 2;

{
    console.log(x); // → ReferenceError
    let x = 4;
}
```

That will save you some bugs!

Closures

We arrive at this point with **hopefully** a very **healthy**, **solid** understanding of **how scope works**

(•_•)

Closures allow **functions** to **remember** and **access** their *original* **scope**…

Even when said **functions** are **executed** in a **different scope**

Let’s **declare** a **closure**…

```
function foo() {
    return 'I am a Closure!’;
}
```

(⊙_⊙)

Closures are a **fundamental** part of how **JavaScript works**

There's **no especial syntax** to declare them

They are **just functions**…

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar();

console.log(x);
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

```
function foo() {
    let x = 42;
    return function () { console.log(x); };
}

let bar = foo();

bar(); // → 42

console.log(x); // → ReferenceError
```

Any **function passed** as **value**, and later **invoked** in another **scope**, are all examples of **closures**

Remember **callbacks**?

Yes! They *are* **closures**!

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo);
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
let x = 'FOO_SCOPE';

function foo() { console.log(x); }

function bar(callback) {
    let x = 'BAR_SCOPE';
    callback();
}

bar(foo); // → FOO_SCOPE
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x);
  }, 1000);
}

foo(42);
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

```
// “Deep down in the JavaScript Engine”

function setTimeout(callback, delay) {
    // Works using magic!
    isItTimeAlready(delay) && callback();
}
```

```
// “Deep down in the JavaScript Engine”

function setTimeout(callback, delay) {
    // Works using magic!
    isItTimeAlready(delay) && callback();
}
```

```
// “Deep down in the JavaScript Engine”

function setTimeout(callback, delay) {
    // Works using magic!
    isItTimeAlready(delay) && callback();
}
```

(ノ°∀°)つ──☆*:・ﾟ

```
function foo(x) {
  setTimeout(function () {
    console.log(x); // → 42
  }, 1000);
}

foo(42);
```

this

All right! You were wondering for sure about ***this***, right!?

The ***this*** keyword is associated with the **execution context**

Scope and **execution context** are closely related, but they are **not the same**...

So, let’s not talk about
this right now, okay?

[awkward silence]

Questions?

Fruits
Vegetables
Soups
Juices
BOTTLED
FROZEN
18 Points
8 Points
CEILING PRICES
NOT RATIONED
YOUR POINT ALLOWANCE MUST LAST FOR THE FULL
Plan How Many Points You Will Use Each Time Before
BUY EARLY IN THE WEEK
A 8 B 8 C 8
A 5 B 5 C 5
B 2 2
B 1

Thanks!