JavaScript

JavaScript

has nothing to do with Java

In this course:

ES6

(aka ES2015)

In this course:

ECMAScript

ES6

(aka ES2015)

In this course:

ES6

(aka ES2015)

ECMAScript

Official (standardized)
name of JavaScript

Running Javascript

Typically run on the browser

Open the Chrome console

- Ctrl + Shift + I (Windows) or Cmd + Alt + I (Mac)
- F12
- Right Click > Inspect > Switch to console tab

Basic (*Primitive*) Data Types

Boolean `true, false`

Numbers only 64-bit floats (no ints).
`0, 1, 3.14, 6.28`

Strings variable length, no separate character type.

`"6170's the best"`

No Value `null, undefined`

Advanced Data Types: *Arrays*

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

```
a = [];
a[0] = "hello";
a[1] = "there";
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
a.pop();
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
a.pop();
a.indexOf("hello");
...
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

Use `length` property for the size of the array.

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
a.pop();
a.indexOf("hello");
...

a.length
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

Use `length` property for the size of the array.

Arrays are heterogenous: they can have elements of different types

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
a.pop();
a.indexOf("hello");
...

a.length
```

Advanced Data Types: *Arrays*

Grow and shrink *dynamically*.

Special `Array` methods for manipulation

Use `length` property for the size of the array.

Arrays are heterogenous: they can have elements of different types

```
a = ["hello", 2, null, [1, 2], "there"];
```

```
a = [];
a[0] = "hello";
a[1] = "there";

a.push("everyone");
a.pop();
a.indexOf("hello");
...

a.length
```

Advanced Data Types: *Objects*

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

```
b = {qty: 3, country: ["USA", "Japan"],
     item: {name: "crayon", price: 5}};
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

```
b = {qty: 3, country: ["USA", "Japan"],
     item: {name: "crayon", price: 5}};
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

```
b = {qty: 3, country: ["USA", "Japan"],
     item: {name: "crayon", price: 5}};
```

```
b.item.price
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

```
b = {qty: 3, country: ["USA", "Japan"],
     item: {name: "crayon", price: 5}};
```

Advanced Data Types: *Objects*

Mappings between *properties* and *values*.

```
a = {hello: "there"};
```

Properties referenced with dot notation.

```
a.hello
```

Properties can be added/modified on the fly.

```
a.hello = "world";
a.goodbye = "for now";
```

Values can be any of the types we've seen so far, including nested objects.

```
b = {qty: 3, country: ["USA", "Japan"],
     item: {name: "crayon", price: 5}};
```

```
b.country[1]
```

JavaScript is a *dynamically-, weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

JavaScript is a *dynamically-*, *weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

```
a = 123;
a = a + 5;

a = "hello world";
a = a + "everyone";
```

JavaScript is a *dynamically-, weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

```
a = 123;
a = a + 5;

a = "hello world";
a = a + "everyone";
```

JavaScript is a *dynamically-, weakly-typed* language

Performs *implicit type coercion* at runtime.

JavaScript is a *dynamically-, weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

```
a = 123;
a = a + 5;

a = "hello world";
a = a + "everyone";
```

Performs *implicit type coercion* at runtime.

```
a = "hello world";
a = a + 5;
```

JavaScript is a *dynamically-, weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

```
a = 123;
a = a + 5;

a = "hello world";
a = a + "everyone";
```

Performs *implicit type coercion* at runtime.

```
a = "hello world";
a = a + 5;
a is "hello world5"
```

JavaScript is a *dynamically-, weakly-typed* language

No fixed types associated with variables. Interpreter determines types of *values* at *runtime*.

```
a = 123;
a = a + 5;

a = "hello world";
a = a + "everyone";
```

Performs *implicit type coercion* at runtime.

Always prefer *strict equality* (===) over loose equality (==).

==

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	■		■					■												■	
false		■		■					■		■					■		■	■		
1	■		■					■												■	
0		■		■					■		■					■		■	■		
-1					■					■											
"true"						■															
"false"							■														
"1"	■		■					■												■	
"0"		■		■					■										■		
"-1"					■					■											
""		■		■							■					■		■			
null												■	■								
undefined												■	■								
Infinity														■							
-Infinity															■						
[]		■		■							■										
{}																					
[[]]		■		■							■										
[0]		■		■					■												
[1]	■		■					■												■	
NaN																					

https://dorey.github.io/JavaScript-Equality-Table/

==

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	■		■					■												■	
false		■		■					■		■					■		■	■		
1	■		■					■												■	
0		■		■					■		■					■		■	■		
-1					■					■											
"true"						■															
"false"							■														
"1"	■		■					■												■	
"0"		■		■					■										■		
"-1"					■					■											
""		■		■							■					■		■			
null												■	■								
undefined												■	■								
Infinity														■							
-Infinity															■						
[]		■		■							■										
{}																					
[[]]		■		■							■										
[0]		■		■					■												
[1]	■		■					■													
NaN																					

===

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	■																				
false		■																			
1			■																		
0				■																	
-1					■																
"true"						■															
"false"							■														
"1"								■													
"0"									■												
"-1"										■											
""											■										
null												■									
undefined													■								
Infinity														■							
-Infinity															■						
[]																					
{}																					
[[]]																					
[0]																					
[1]																					
NaN																					

https://dorey.github.io/JavaScript-Equality-Table/

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.
Note: This is not the same as making the variable immutable.

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.
Note: This is not the same as making the variable immutable.

```
const COLOR = "blue";
COLOR = "yellow"; // error
const COLOR = "green"; // error
```

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.
Note: This is not the same as making the variable immutable.

```
const COLOR = "blue";
COLOR = "yellow"; // error
const COLOR = "green"; // error
```

```
const PERSON = {name: "arvind"};
PERSON = {name: "daniel"}; // error
const PERSON = {name: "daniel"}; // error
```

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.
Note: This is not the same as making the variable immutable.

```
const COLOR = "blue";
COLOR = "yellow"; // error
const COLOR = "green"; // error
```

```
const PERSON = {name: "arvind"};
PERSON = {name: "daniel"}; // error
const PERSON = {name: "daniel"}; // error
PERSON.name = "daniel"; // a-ok
```

Scopes & Declaring Variables

By default, variables are defined in the *global* scope (code examples so far).

Use the `let` keyword to declare local variables, with *block* scope.

The `const` keyword declares *block* scoped variables that *cannot be reassigned*.
Note: This is not the same as making the variable immutable.

Pre ES6, only the `var` keyword was available to declare local variables.
Has *function* scope, and declarations are *hoisted* to the top of the function.

Functions

Functions

Functions are declared with a name, arguments, and a body.

```
function multiply(a, b) {
  return a * b;
}
```

Functions

Functions are declared with a name, arguments, and a body.

```
function multiply(a, b) {
  return a * b;
}
```

Functions can take a variable number of arguments. The special `arguments` variable lists them as an array.

```
function multiply() {
  return arguments[0] *
    arguments[1];
}
```

Functions

Functions are declared with a name, arguments, and a body.

```
function multiply(a, b) {
  return a * b;
}
```

Functions can take a variable number of arguments. The special `arguments` variable lists them as an array.

```
function multiply() {
  return arguments[0] *
    arguments[1];
}
```

With ES6, arguments can have default values.

```
function multiply(a, b = 1) {
  return a * b;
}
```

Control Flow Syntax

Control Flow Syntax

Conditional statements

```
if (a > b) {
  return a - b;
}
```

Control Flow Syntax

Conditional statements

```
if (a > b) {
  return a - b;
}
```

For loops

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Control Flow Syntax

Conditional statements

```
if (a > b) {
  return a - b;
}
```

For loops

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Iterating through objects

```
let map = {name: “Ben”, work: “student”}
for (key in map) {
  console.log(map[key]);
}
```

Note: This does not work for arrays