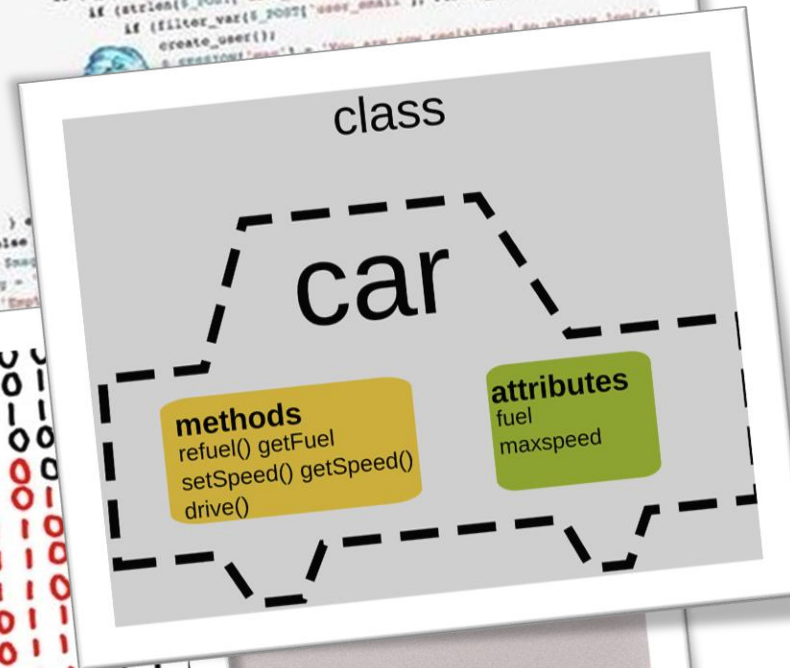
Programming Fundamentals (in JavaScript) 1: Basics

Stefano Balietti

Center for European Social Science Research at Mannheim University (MZES)
Alfred-Weber Institute of Economics at Heidelberg University

@balietti | stefanobalietti.com | @nodegameorg | nodegame.org



Variables

Variables

http://javascript.info/

Variables

```
let message = 'Hello!';
```

http://javascript.info/

Variables

```
let message = 'Hello!';
```

Keyword announcing that what follows is the name of a *new* variable

Variables

http://javascript.info/

Variables

```
let message = 'Hello!';
```

The name of the variable. It is *case sensitive.*
It references the value throughout the rest of the code.
Depending on the type of its value, it might expose other methods/properties.

Variables

http://javascript.info/

Variables

```
let message = 'Hello!';
```

Keyword that assigns what is to its right to the variable to the left.
Other programming language use <- to indicate the directionality.

http://javascript.info/

Variables

```
let message = 'Hello!';
```

The value of assignment: a string wrapped in quotes

http://javascript.info/

Variables

```
let message = 'Hello!';
```

The semicolon signals that the command is finished.

http://javascript.info/

Variables

```
let message =
'Hello!';
```

Is this valid?

Variables

http://javascript.info/

Variables

```
let message =
'Hello!';
```

Is this valid? YES.
Commands can span over multiple lines, therefore it is important to use the semicolon to specify where they end.

Variables

http://javascript.info/

Variables

```
let message;
message = 'Hello!';
```

❓ **Is this valid?**

Variables

http://javascript.info/

Variables

```
let message;
message = 'Hello!';
```

Is this valid? YES.

When do you want to separate creation and assignment?

Variables

http://javascript.info/

Variables

```
let message;
```

Creation

```
... THINGS HAPPENS ...
```

Value to assign not available immediately
Uncertainty about which code block will assign it
Need to be available across different code blocks
(more on variable scoping later)

```
message = 'Hello!';
```

Assignment

Variables

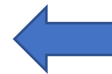
http://javascript.info/

Variables

```
let message;
```

Creation

```
... THINGS HAPPENS ...
```



Value to assign not available immediately
Uncertainty about which code block will assign it
Need to be available across different code blocks
(more on variable scoping later)

```
message = 'Hello!';
```

Assignment

Notice we don't use **let** again, otherwise it will throw an error.

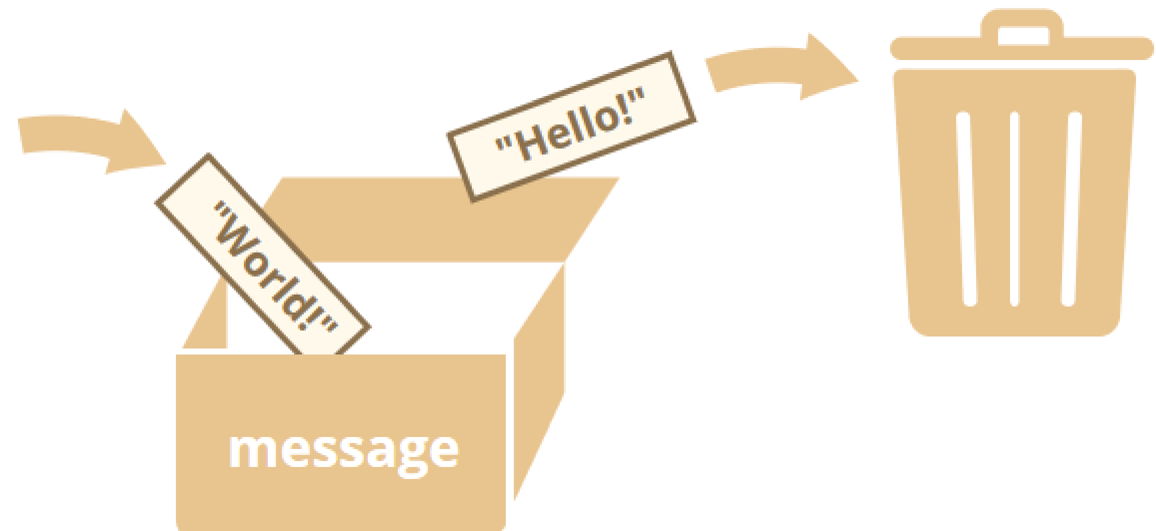
http://javascript.info/

Variables

```
let message = 'Hello!';
```

```
// value changed.
message = 'World!';

alert(message);
```



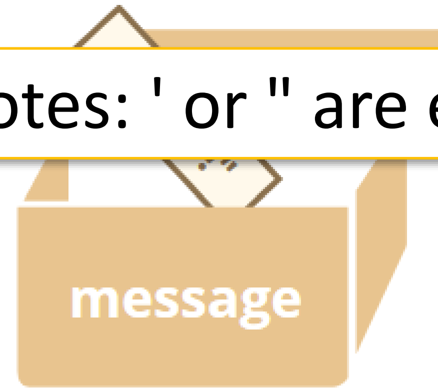
Variables

http://javascript.info/

Variables

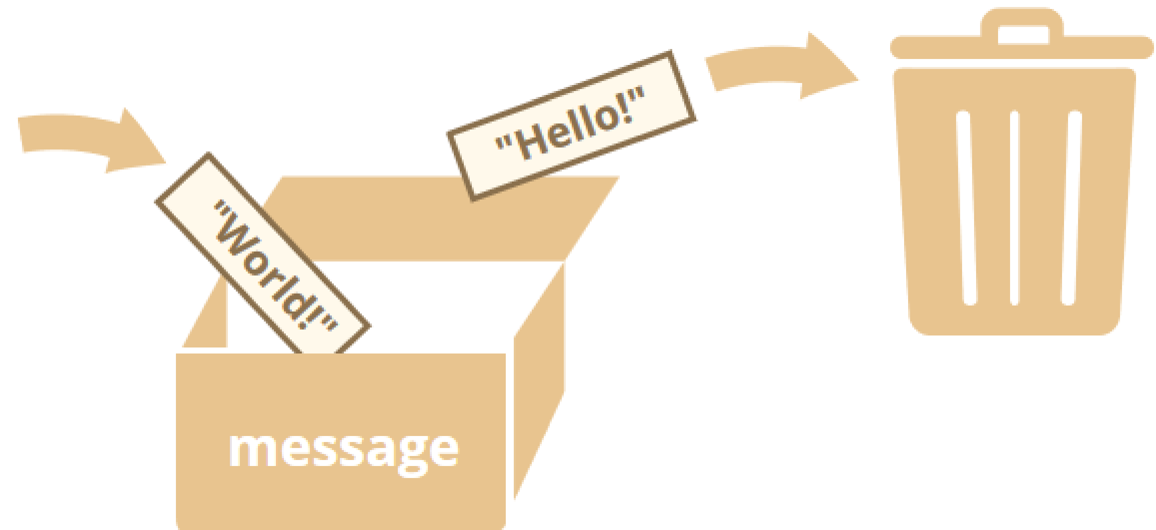
Strings must be wrapped in quotes: ' or " are equivalent.

```
let message = 'Hello!';
```




```
// value changed.
message = 'World!';

alert(message);
```



Variables

http://javascript.info/

Variables

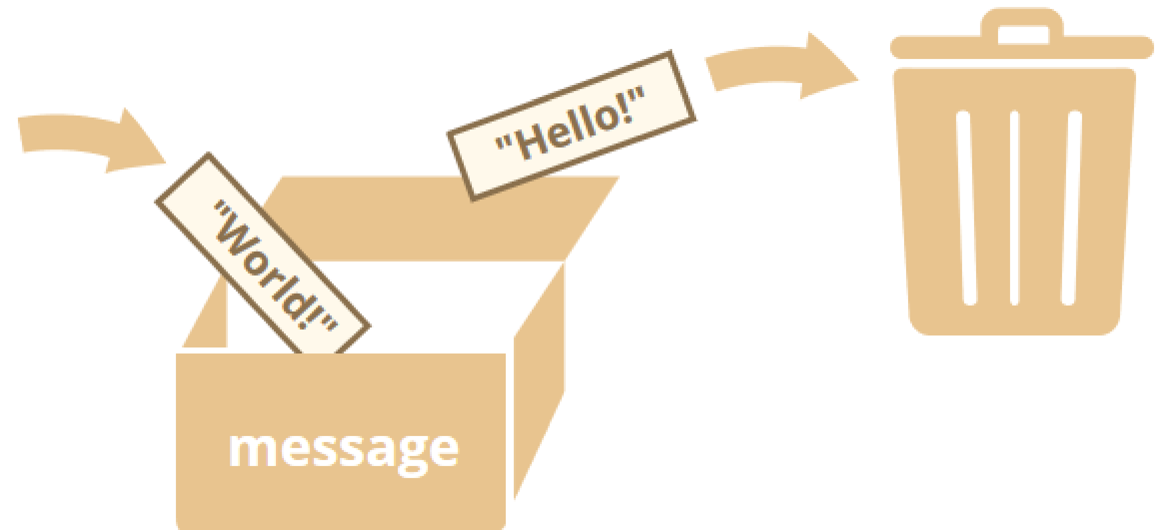
Strings must be wrapped in quotes: ' or " are equivalent.

```
let message = 'Hello!';
```

Text following // is a comment and it is not read by JavaScript

```
// value changed.
message = 'World!';


alert(message);
```



Variables

http://javascript.info/

Variables

Strings must be wrapped in quotes: ' or " are equivalent.

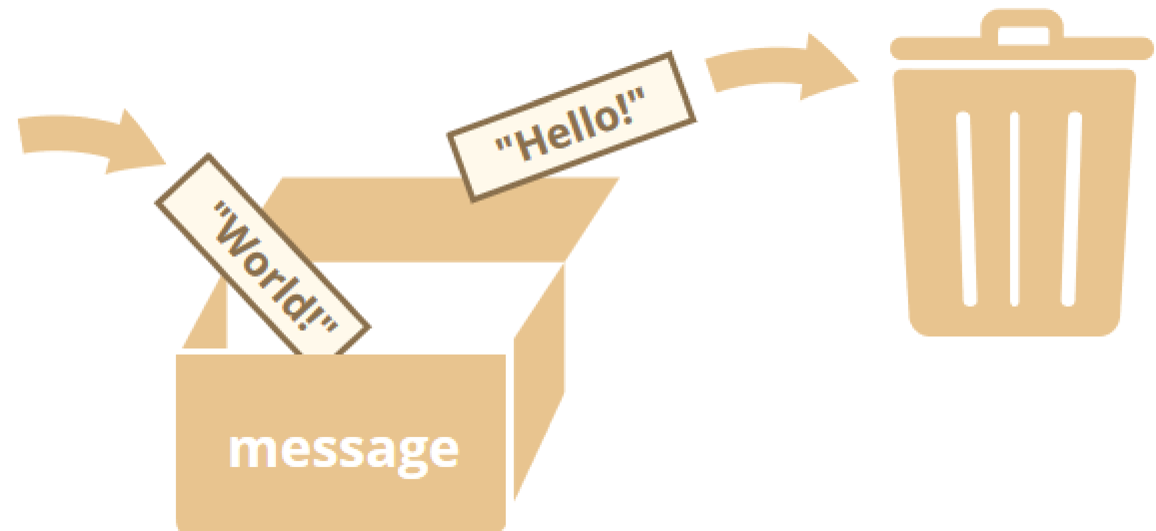
```
let message = 'Hello!';
```

Text following // is a comment and it is not read by JavaScript

```
// value changed.
message = 'World!';
```

Opens a popup in the Browser

```
alert(message);
```



Main Variable Types in JS

```
let a = 1; // number
let b = 'Hello world!'; // string
let c = false; // boolean
let d = function(p) { return p+1; }; // function
let e = { key: 'value' }; // object
let f = [ "value1", 3, c ]; // array (type is object)
```

Main Variable Types in JS

```
let a = 1; // number
let b = 'Hello world!'; // string
let c = false; // boolean
let d = function(p) { return p+1; }; // function
let e = { key: 'value' }; // object
let f = [ "value1", 3, c ]; // array (type is object)
```

Main Variable Types in JS

```
let a = 1; // number
let b = 'Hello world!'; // string
let c = false; // boolean
let d = function(p) { return p+1; }; // function
let e = { key: 'value' }; // object
let f = [ "value1", 3, c ]; // array (type is object)
```

TWO IMPORTANT CONCEPTS:

- Variables are **loosely** (or **"dynamically"**) **typed**
- Variables are scoped within the **block** in which they are declared

Variables Are Dynamically Typed

```
var message = 'Hello!';
// value changed.
message = 'World!';
alert(message);

// type of value changed to number
message = 2019;

// string concatenation (works also with numbers).
alert('This is year ' + message);
```

Variables Are Dynamically Typed

```
var message = 'Hello!';
// value changed.
message = 'World!';
alert(message);

// type of value changed to number
message = 2019;

// string concatenation (works also with numbers).
alert('This is year ' + message);
```

Variables are "loosely typed," that is their type (string, number, etc.) can be changed after assignment

Variables Are Dynamically Typed

```
var message = 'Hello!';
// value changed.
message = 'World!';
alert(message);

// type of value changed t
message = 2019;

// string concatenation (works also with numbers).
alert('This is year ' + message);
```

Plus is used to concatenate strings. Variable are converted on-the-fly when they are manipulated together with others of a different type. *Need to be careful because it can create unexpected behavior.*

Type conversions

```
"7" + 3;

"7" – 3;
```

Type conversions

```
"7" + 3;    "73";  // Converted to String

"7" - 3;    4;     // Converted to Number
```

Type conversions

```
"7" + 3;    "73";  // Converted to String

"7" - 3;    4;     // Converted to Number
```

Why is that?

Type conversions

```
"7" + 3;    "73";  // Converted to String

"7" - 3;    4;     // Converted to Number
```

Why is that?

JavaScript made its best guess. Plus is the operator for string concatenation, hence everything became a string. Minus can only be for arithmetic operations, hence the conversion to number.

Do the Math

Operator	Operation	Example
+	Addition	`1+1;      // 2`
-	Subtraction	`1-1;      // 0`
/	Division	`1/10;     // 0.1`
*	Multiplication	`2*2;      // 4`
%	Remainder	`7%4;      // 1`

Do the Math

Operator	Operation	Example
+	Addition	`1+1;     // 2`
-	Subtraction	`1-1;     // 0`
/	Division	`1/10;    // 0.1`
*	Multiplication	`2*2;     // 4`
%	Remainder	`7%4;     // 1`
++	Add 1 to the current value (also --)	`let a = 1; a++;   // 2`
+=	Add something to current value (also *=, -=, /=)	`let a = 1; a+=2;  // 3`

Do the Math

Operator	Operation	Example
+	Addition	`1+1;      // 2`
-	Subtraction	`1-1;      // 0`
/	Division	`1/10;     // 0.1`
*	Multiplication	`2*2;      // 4`
%	Remainder	`7%4;      // 1`
++	Add 1 to the current value (also --)	`let a = 1; a++;   // 2`
+=	Add something to current value (also *=, -=, /=)	`let a = 1; a+=2;  // 3`
**	Exponentiation	`3**2;     // 9`
Math	The Math object offers several operations	`Math.random();      // 0.1231` `Math.floor(3.451);  // 3`

Do the Math

Operator	Operation	Example
+	Addition	`1+1;      // 2`
-	Subtraction	`1-1;      // 0`
/	Division	`1/10;     // 0.1`
*	Multiplication	`2*2;      // 4`
%	Remainder	`7%4;      // 1`
++	Add 1 to the current value (also --)	`let a = 1; a++;  // 2`
+=	Add something to current value (also *=, -=, /=)	`let a = 1; a+=2; // 3`
**	Exponentiation	`3**2;     // 9`
Math	The Math object offers several operations	**`Math.random();`** `// 0.1231` **`Math.floor(3.451);`** `// 3`

The round parentheses signal a *method invocation*, what is inside the parentheses is an *input parameter*. More on this later…

Conditional Operators: If/Else Statements

```
if ( CONDITION ) {
   // Execute if condition is TRUE
}
else {
  // Execute if condition is FALSE
}
```

You say that if/else statements are **"branching off"** your code, because only one of the two branches will be executed at run-time.

Conditional Operators: If/Else Statements

```
if ( CONDITION ) {
   // Execute if condition is TRUE
}
else {
  // Execute if condition is FALSE
}
```

If/Else can be chained and the order matters.

Conditional Operators: If/Else Statements

```
if ( CONDITION1 ) {
   // Execute if condition is TRUE
}
else if ( CONDITION2 ){
   // Execute if condition1 is FALSE and
   // condition2 is TRUE.
}
```

Will one of the two branches *always* be executed?

Conditional Operators: If/Else Statements

```
if ( CONDITION1 ) {
   // Execute if condition is TRUE
}
else if ( CONDITION2 ){
   // Execute if condition1 is FALSE and
   // condition2 is TRUE.
}
```

Will one of the two branches *always* be executed?
Not if both conditions are false.

Conditional Operators: If/Else Statements

```
if ( CONDITION1 ) {
   // Execute if condition is TRUE
}
else if ( CONDITION2 ){
   // Execute if condition1 is FALSE and
   // condition2 is TRUE.
}
else {
   // If both conditions above are FALSE,
   // I will be executed.
}
```

Logical Operators

AND: &&

```
if ( CONDITION1 && CONDITION2 ) {
   // Executed only if both conditions are TRUE
}
```

OR: ||

```
if ( CONDITION1 || CONDITION2 ) {
   // Executed if either condition is TRUE
}
```

NOT: !

```
if ( !CONDITION ) {
   // Executed only if condition is FALSE
}
```

Logical Operators

AND: &&

```
if ( CONDITION1 && CONDITION2 ) {
   // Executed only if both conditions are TRUE
}
```

OR: ||

```
if ( CONDITION1 || CONDITION2 ) {
   // Executed if either condition is TRUE
}
```

"Short-circuit" operators. The second condition is evaluated only if needed.

NOT: !

```
if ( !CONDITION ) {
   // Executed only if conditio
}
```

Comparisons

Like assignments, comparisons have an operator which separates a left-hand side term and right-hand side term, e.g., 3 > 1, and they return a Boolean value (true or false).

Operator	Operation	Example
>	Greater than	`2>1;      // true`
>=	Greater or equal than	`1>=1;     // true`
<	Less than	`10<1;     // false`
<=	Less or equal than	`3<=3;     // true`
==	Equals to	`2==2;     // true`
===	Strictly equals to	`2===2;    // true`

Comparisons

Like assignments, comparisons have an operator which separates a left-hand side term and right-hand side term, e.g., 3 > 1, and they return a Boolean value (true or false).

Operator	Operation	Example
>	Greater than	`2>1;      // true`
>=	Greater or equal than	`1>=1;     // true`
<	Less than	`10<1;     // false`
<=	Less or equal than	`3<=3;     // true`
==	**Equals to**	**`2==2;     // true`**
===	**Strictly equals to**	**`2===2;    // true`**

Why do we need two types of equals?

Comparisons

Like assignments, comparisons have an operator which separates a left-hand side term and right-hand side term, e.g., 3 > 1, and they return a Boolean value (true or false).

Operator	Operation	Example
`>`	Greater than	`2>1;      // true`
`>=`	Greater or equal than	`1>=1;     // true`
`<`	Less than	`10<1;     // false`
`<=`	Less or equal than	`3<=3;     // true`
`==`	**Equals to**	**`2==2;     // true`**
`===`	**Strictly equals to**	**`2===2;    // true`**

Why do we need two types of equals? **Because of type conversions**

Variable Comparison: === vs ==

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																					

- If the cell is filled, it means the result of a comparison is true, otherwise false
- The table on the diagonal reads:

```
if (true === true)   // true
if (false === false) // true
...
```

Variable Comparison: === vs ==

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																					

Variable Comparison: === vs ==

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

===

==

```
// Using double equal.
if (1 == true) {
    console.log('This can't be true!');
}
```

Variable Comparison: === vs ==

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																					

```
// Using double equal.
if (1 == true) {
    console.log("This can't be true!");
}
// will print "This can't be true!"
```

Variable Comparison: === vs ==

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

===

==

```
// Using triple equal.
if (1 === true) {
    console.log("This can't be true!");
}
```

Variable Comparison: === vs ==

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

===

==

```
// Using triple equal.
if (1 === true) {
    console.log("This can't be true!");
}
// will print nothing
```

Variable Comparison: === vs ==

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

==

Value	Executes	Code
true	■	if (true) { /* executes */ }
false	□	if (false) { /* does not execute */ }
1	■	if (1) { /* executes */ }
0	□	if (0) { /* does not execute */ }
-1	■	if (-1) { /* executes */ }
"true"	■	if ("true") { /* executes */ }
"false"	■	if ("false") { /* executes */ }
"1"	■	if ("1") { /* executes */ }
"0"	■	if ("0") { /* executes */ }
"-1"	■	if ("-1") { /* executes */ }
""	□	if ("") { /* does not execute */ }
null	□	if (null) { /* does not execute */ }
undefined	□	if (undefined) { /* does not execute */ }
Infinity	■	if (Infinity) { /* executes */ }
-Infinity	■	if (-Infinity) { /* executes */ }
[]	■	if ([]) { /* executes */ }
{}	■	if ({}) { /* executes */ }
[[]]	■	if ([[]]) { /* executes */ }
[0]	■	if ([0]) { /* executes */ }
[1]	■	if ([1]) { /* executes */ }
NaN	□	if (NaN) { /* does not execute */ }

Use always ===
(unless you have a good reason)

Block Scope

```
let favoriteFood = 'lasagne';

if (favoriteFood === 'lasagne') {
    console.log('Well Done!');
    favoriteFood += ' with a lot of cheese';
    let secondFavorite = 'pizza';
}
```

What will it print?

```
console.log(favoriteFood);
console.log(secondFavorite);
```

Block Scope

```
let favoriteFood = 'lasagne';

if (favoriteFood === 'lasagne') {
    console.log('Well Done!');
    favoriteFood += ' with a lot of cheese';
    let secondFavorite = 'pizza';
}
```

What will it print?

```
console.log(favoriteFood); // 'lasagne with a lot of cheese';
console.log(secondFavorite); // undefined (error is thrown)
```

Block Scope

```
let favoriteFood = 'lasagne';

if (favoriteFood === 'lasagne') {
    console.log('Well Done!');
    favoriteFood += ' with a lot of cheese';
    secondFavorite = 'pizza';
}
```

secondFavorite lives only within the block in which it is defined. Blocks are delimited by curly brackets.

```
console.log(favoriteFood); // 'lasagne with a lot of cheese';
console.log(secondFavorite); // undefined (error is thrown)
```

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';
```

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';

let length = favoriteFood.length; // 28
```

The dot operator grants access to the property of objects. Wait wasn't `favoriteFood` a string? Yes, but it exposes methods and properties like an object.

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';

let length = favoriteFood.length; // 28
```

The dot operator grants access to the property of objects. Wait wasn't `favoriteFood` a string? Yes, but it exposes methods and properties like an object.

Here we learn that there are 28 characters in the string. That is a bit long for a single favorite food. *Let's investigate*

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';

let length = favoriteFood.length; // 28

let index = favoriteFood.indexOf('with a lot of cheese');
```

The method indexOf returns the index of the first occurrence of the string passed as input parameter, or -1 if not found.

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';

let length = favoriteFood.length; // 28

let index = favoriteFood.indexOf('with a lot of cheese');

if (index !== -1) {
    console.log('Uhm...are you American?');
    favoriteFood = favoriteFood.substring(0, index).trim();
}
```

substring returns a portion of the original string as specified by its input parameters.

String Methods

```
favoriteFood // 'lasagne with a lot of cheese';

let length = favoriteFood.length; // 28

let index = favoriteFood.indexOf('with a lot of cheese');

if (index !== -1) {
    console.log('Uhm...are you American?');
    favoriteFood = favoriteFood.substring(0, index).trim();
}
```

Trim removes white beginning and trailing white spaces. We *chained* it to the results of the previous method.

Other Ways to Declare Variables

```
var message = 'I am an old-timer!';
```

```
const MESSAGE = 'I am immutable';
```

Other Ways to Declare Variables

```
var message = 'I am an old-timer!';
```

Var variables are prior to ES6, still *valid, but* its usage is not recommended any more.

```
const MESSAGE = 'I am immutable';
```

Other Ways to Declare Variables

```
var message = 'I am an old-timer!';
```

Var variables are prior to ES6, still *valid, but* its usage is not recommended any more.

```
const MESSAGE = 'I am immutable';
```

Constants are variables that will throw an error if you attempt to re-assign them. *But not if you change them!*

More on vars https://levelup.gitconnected.com/stop-using-var-to-declare-variables-in-javascript-6c0caec16f43

Exercises

Part_1_Basics/1_primitive_types.js

Objects

Objects

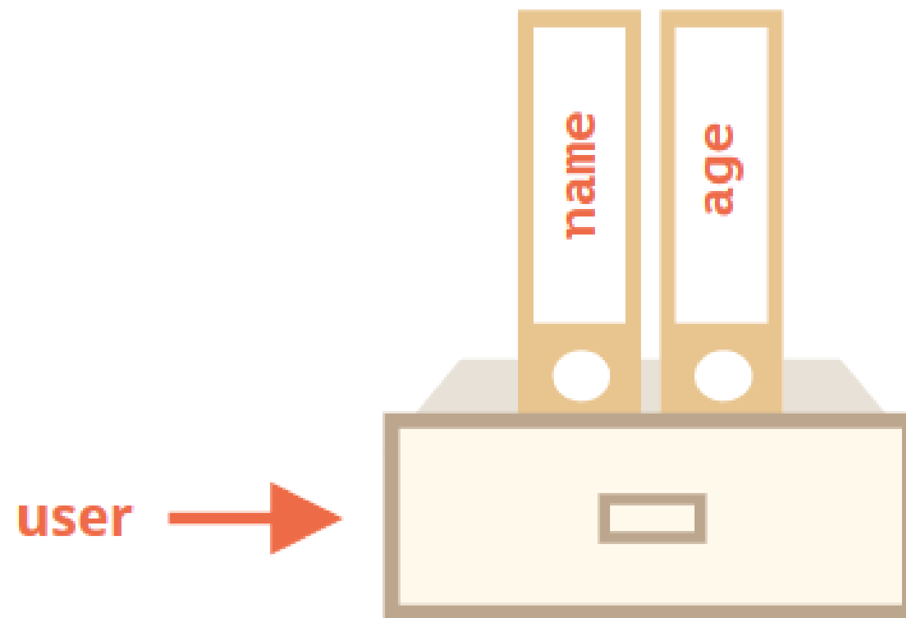
- Objects are containers for variables indexed by a key (in other programming languages they may be called maps or dictionaries)

- They can contain variables of any type inside

Objects

http://javascript.info/

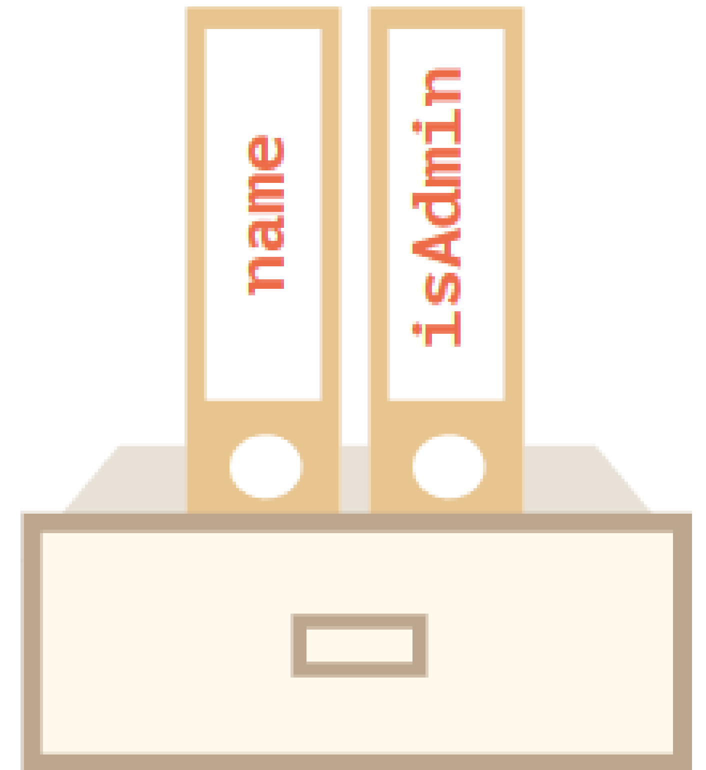
```
var user = {
  name: "John", // by key "name" store value "John"
  age: 30       // by key "age" store value 30
};
```



Objects

http://javascript.info/

```
// We now add a new property
// Note! JavaScript is case sensitive
user.isAdmin = true;
// Delete an existing one.
delete user.age;
```

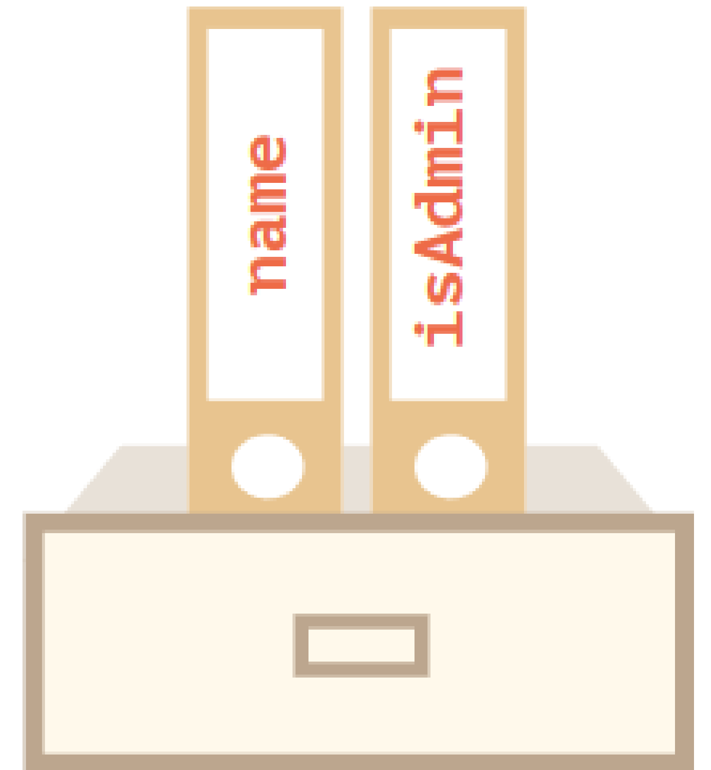


Objects

http://javascript.info/

```
// We now add a new property
// Note! JavaScript is case sensitive
user.isAdmin = true;
// Delete an existing one.
delete user.age;
```

The dot operator accesses the value of a given property inside the object.

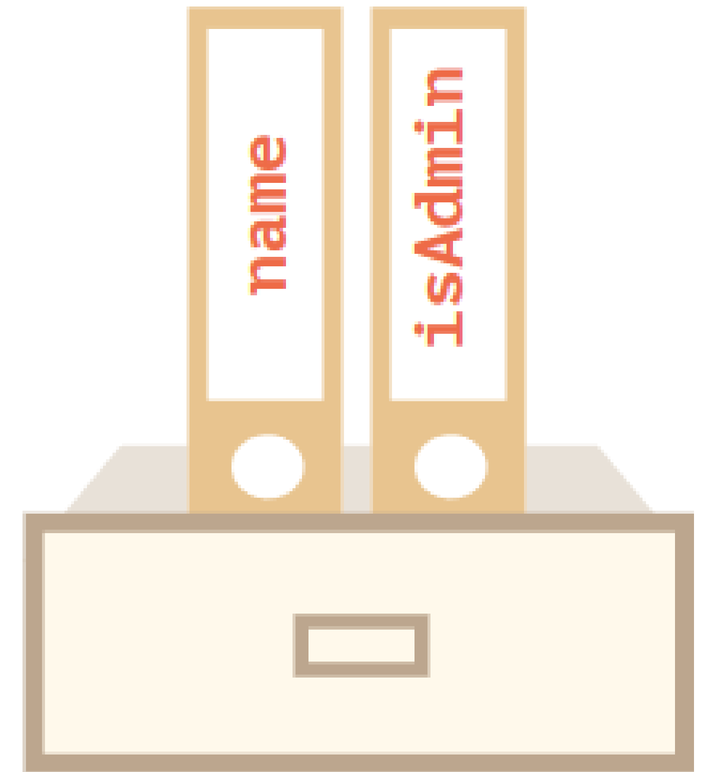


Objects

http://javascript.info/

```
// We now add a new property
// Note! JavaScript is case sensitive
user.isAdmin = true;
// Delete an existing one.
delete user.age;
```

The dot operator accesses the value of a given property inside the object.
If the property was not previously defined (as in this case), it will be simply created.



Looping in Objects (For In)

```
for (let property in user) {
    console.log(property + ': ' + user[property]);
}
```

Looping in Objects (For In)

```
for (let property in user) {
  if (user.hasOwnProperty(property)) {
    console.log(property + ': ' + user[property]);
  }
}

// Output.
// name: John
// isAdmin: true
```

- **hasOwnProperty** is necessary to avoid contamination of other properties belonging to the object and not added by the user
- **MUST USE ALWAYS.**

Looping in Objects (For In)

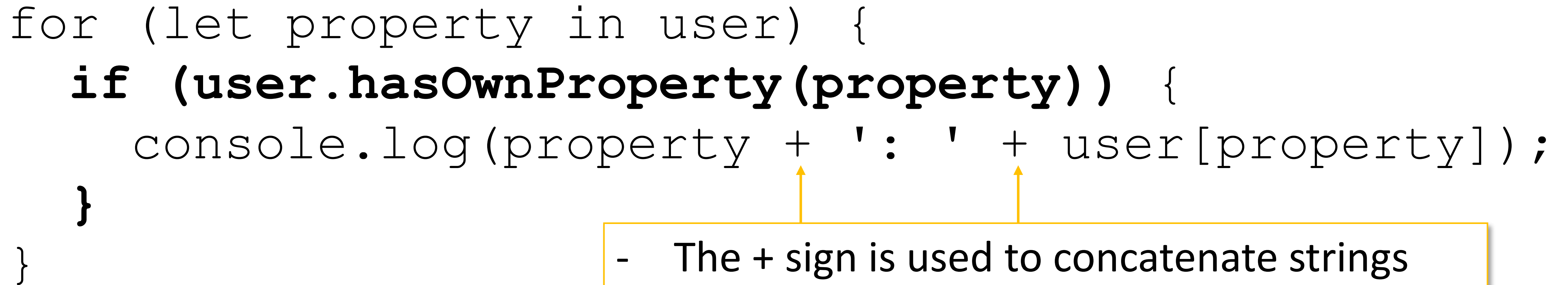
```
for (let property in user) {
   if (user.hasOwnProperty(property)) {
     console.log(property + ': ' + user[property]);
   }
}

// Output.
// name: John
// isAdmin: true
```

- The square parentheses allows one to access the value of the property of an object, when the property name is contained in a variable.
- The following notations are equivalent:

```
user.name;    // John
user['name']; // John;
var property = "name";
user[property]; // John
```

Looping in Objects (For In)




```
// Output.
// name: John
// isAdmin: true
```

Arrays

- Arrays are containers for variables indexed by a number
- They are faster to iterate through than objects
- Like objects, they can contain variables of any type

Arrays

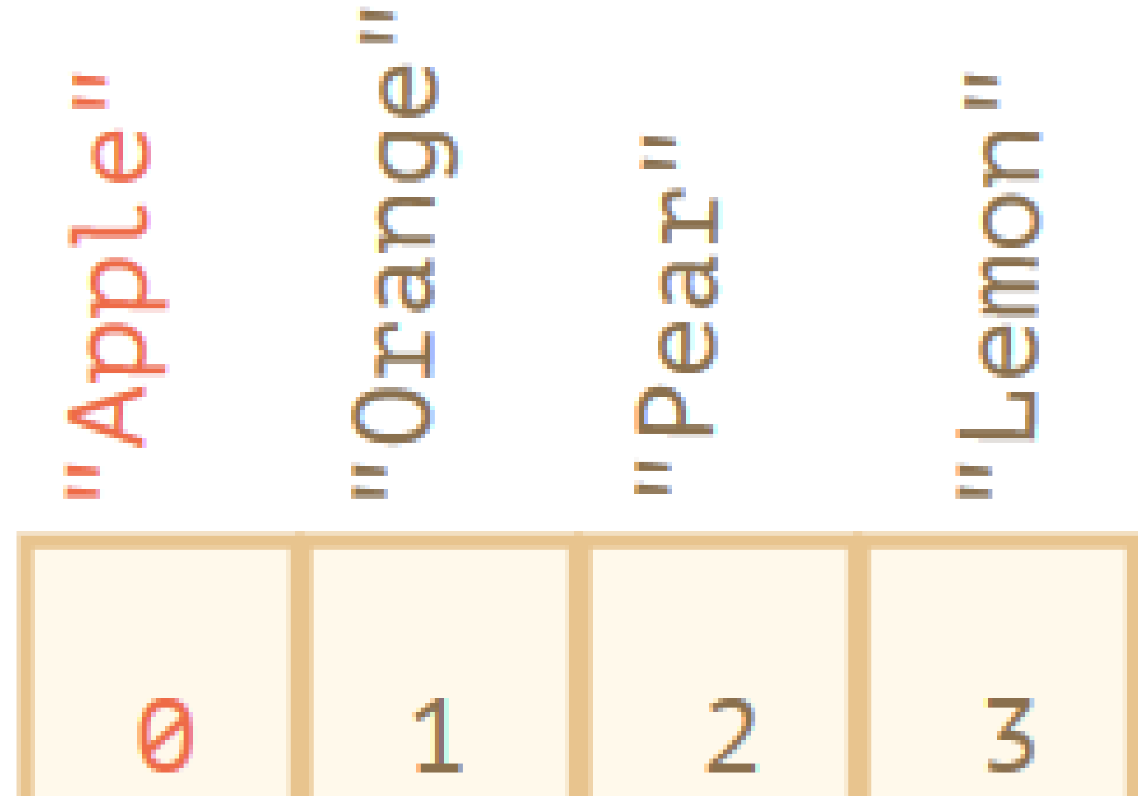
http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```

Arrays

http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```



Arrays are collections of items indexed by a number.
The first item has index 0, the second item has index 1, and so on…
Arrays can contain items of any type (string, number, etc.) and also mix them.

Arrays

http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```

```
fruits.length;
```

Arrays

http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```

```
fruits.length; // 4
```

Arrays

http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```

```
fruits.length; // 4
fruits[2];
```

Arrays

http://javascript.info/

```
var fruits = [
"Apple",
"Orange",
"Pear",
"Lemon"
];
```

```
fruits.length; // 4
fruits[2]; // "Pear"
```

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   // Code to be added here.
}
```

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];



var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   // Code to be added here.
}
```

A for loop repeats the code inside the parenthesis as long as a condition is true (we will add the code later).

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {

}
```

It is divided in 3 parts, separated by ; (semicolon).

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {

}
```

It is divided in 3 parts, separated by ; (semicolon).
Initialization

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {

}
```

It is divided in 3 parts, separated by ; (semicolon).
Initialization ; **Condition**

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];



var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {

}
```

It is divided in 3 parts, separated by ; (semicolon).
Initialization ; Condition ; **Increment (i++ means i = i + 1)**

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
    message += fruits[i] + ',';
}
alert(message);
```

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
    message += fruits[i] + ',';
}
aler
```

The first iteration i = 0, the second iteration i = 1, the third iteration i = 2, and the fourth and last iteration i = 3. In this way, we can access all the items in the array and create a text with all the fruits we like.

Arrays and For Loops

```
var fruits = [ "Apple", "Orange",
               "Pear", "Lemon" ];


var message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
    message += fruits[i] + ',';
}
alert(mess
```

However, there is a grammatical problem! The text will end with a comma, instead that with a dot. **Do you know how to fix it?**

Exercises

Part_1_Basics/2_objects_and_loops.js

Functions

- Functions are reusable blocks of codes
- They may take input parameters and may return an output value
- Functions abstract the complexity of code operations inside their body

Functions

```
// Standard function.
// Functions are reusable blocks of codes.
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.name;
    alert(message);
}
```

Functions

Note! Functions are also called "methods" or "callbacks." The definition is always the same.

```
// Standard function.
// Functions are reusable blocks of codes.
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.name;
    alert(message);
}
```

Functions

```
// Standard function.
// Functions are reusable block
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.name;
    alert(message);
}
```

This line is the **function declaration.** It specifies the name of the function as well as input parameters

Functions

```
// Standard function.
// Functions are reusable block
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.na
    alert(message);
}
```

This line is the **function declaration.** It specifies the name of the function as well as input parameters

person is the input parameter

Functions

```
// Standard function.
// Functions are reusable blocks of codes.
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.name;
    alert(message);
}
```

Functions

```
// Standard function.
// Functions are reusable blocks of codes.
function showPerson(person) {
    let message = 'Hello, ';
    message = message + person.name;
    alert(message);
}
```

The part wrapped in curly brackets is called the **"body"** of the function, it specifies what the it actually does internally

Functions

```
// Execute the function.
// Remember! We have already defined
// the variable user before.
showPerson(user);
```

Function Invocation

```
// Execute the function.
// Remember! We have already defined
// the variable user before.
showPerson(user);
```

Note! Functions are "**invoked**" or "**executed**" or "**called.**" The terms are synonymous.

Function Invocation

```
// Standard function.
function showPerson2(person) {
    let message = 'Hello, ';
    message = message + 'person.name';

    if (person.isAdmin === true) {
        message += 'I notice that you are an admin';
    }
    alert(message);
}
```

Functions

```
// Standard function.
function showPerson2(person) {
    let message = 'Hello, ';
    message = message + 'person

    if (person.isAdmin === true) {
        message += 'I notice that you are an admin';
    }
    alert(message);
}
```

This is an "if statement." If the condition is true, it will execute the text inside the parentheses

Functions

```
// Standard function.
function showPerson2(person) {
    let message = 'Hello, ';
    message = message + 'person

    if (person.isAdmin === true) {
        message += 'I notice that you are an admin';
    }
    alert(message);
}
```

The number of equals matters
- 1 equal for assignment to variables
- 2 equals for comparison
- 3 equals for **strict** comparison

Input Parameters

```
// Internally modifies input.
function doSomething(obj, num, str) {
    obj.a = 10;
    num = 1;
    str = 'a';
}
let obj = {}, num = 0, str = '';
doSomething(obj, num, str);

console.log(obj);
console.log(num);
console.log(str);
```

What will the final values of the object, the string, and the number be, after they have been modified by the function?

Input Parameters

```
// Internally modifies input.
function doSomething(obj, num, str) {
    obj.a = 10;
    num = 1;
    str = 'a';
}
let obj = {}, num = 0, str = '';
doSomething(obj, num, str);

console.log(obj); // { a: 10 }
console.log(num); // 0
console.log(str); // ''
```

Objects are passed as a *reference (to an address in memory),* while numbers and strings are *copies (primitive types cannot be referenced).*

Modifying a copy does not affect the value outside the function, modifying the reference does.

Our Previous Example: Arrays and For Loops

```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
    message += fruits[i];
    if (i < (fruits.length - 1 )) {
        message += ', ';
    }
    else {
        message += '.';
    }
}
alert(message);
```

Our Previous Example: Arrays and For Loops

```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
    message += fruits[i];
    if (i < (fruits.length - 1 )) {
        message += ', ';
    }
    else {
        message += '.';
    }
}
alert(message);
```

That's a lot of code inside the for-loop. How to make it more compact and more general with a function?

Functions with Returns

We create a function for joining words

```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}

let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

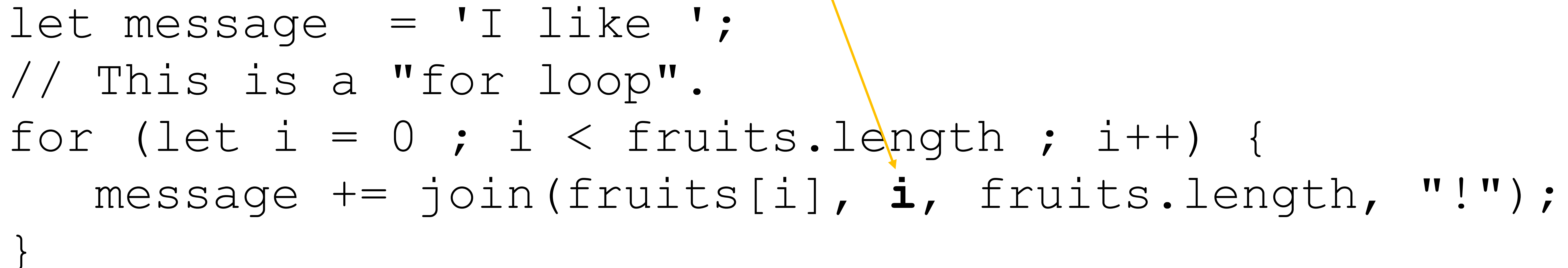
Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}

let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

Functions with Returns

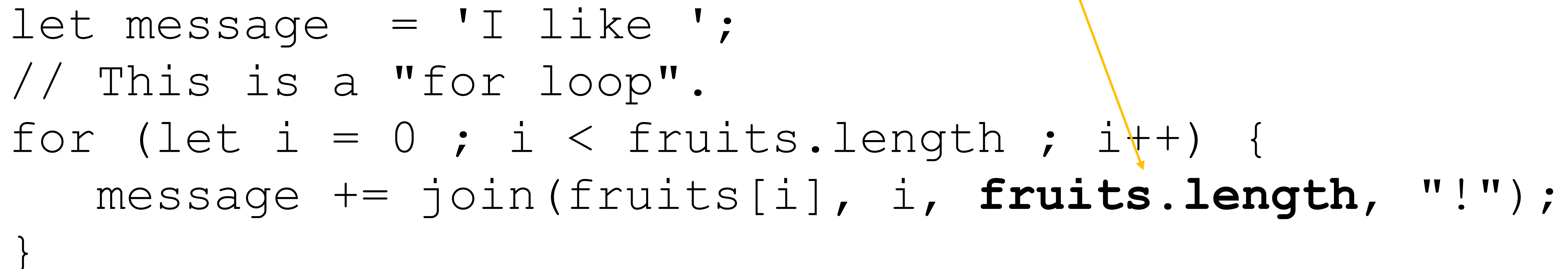
```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}
```



```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

Functions with Returns

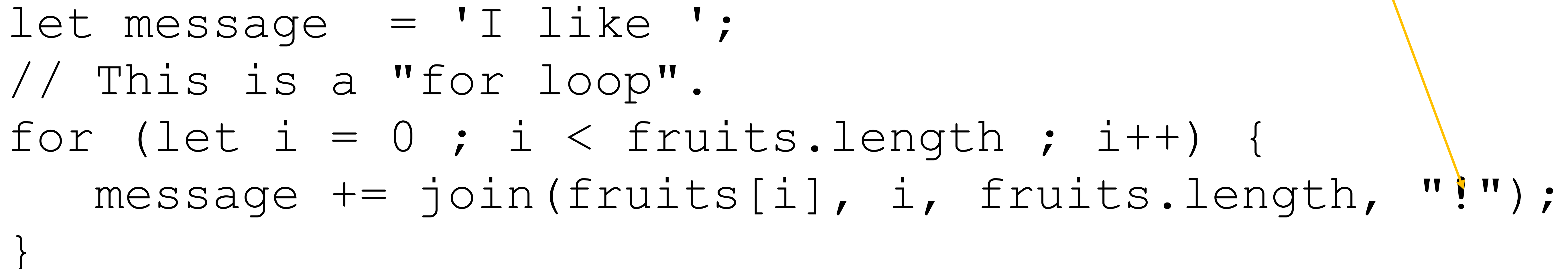
```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}
```



```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}
```



```
let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}

let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

This last value is *optional,* because the function defines a default parameter.

Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}

let message  = 'I like ';
// This is a "for loop".
for (let i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

If-else branches can be written without parentheses, and they apply to the next line, as delimited by semicolon (;).

Functions with Returns

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}

let message  = 'I like ';
// This is a "for loop".
for (var i = 0 ; i < fruits.length ; i++) {
   message += join(fruits[i], i, fruits.length, "!");
}
```

The return keyword makes available outside of the function the modified variable `word`.

Ternary Operator

We can make a new function **join2** even more compact. The ternary operator **?** merges together an if/else statement in one line, separating the two branches with **:**

```
function join(word, index, arraySize, endSign = '.') {
    if (index !== arraySize -1) word += ',';
    else word += endSign;
    return word;
}
function join2(word, index, arraySize, endSign = '.')
    word += index !== arraySize -1 ? ',' : endSign;
    return word;
}
```

Ternary Operator

We can make a new function **join3** even more compact by merging the ternary operator and the return statement in one line.

```
function join2(word, index, arraySize, endSign = '.')
    word += index !== arraySize -1 ? ',' : endSign;
    return word;
}
function join3(word, index, arraySize, endSign = '.') {
    return word += (index !== arraySize -1 ? ',' : endSign);
}
```

Is join3 better than join2?

Ternary Operator

We can make a new function **join3** even more compact by merging the ternary operator and the return statement in one line.

```
function join2(word, index, arraySize, endSign = '.')
    word += index !== arraySize -1 ? ',' : endSign;
    return word;
}
function join3(word, index, arraySize, endSign = '.') {
    return word += (index !== arraySize -1 ? ',' : endSign);
}
```

Is join3 better than join2? *NO*. join3 is much less readable and in the long-term it will increase the maintenance costs.

Private Variables

- Variables declared inside a function are expected to stay private, that is not accessible outside of the function.

Private Variables

- Variables declared inside a function are expected to stay private, that is not accessible outside of the function.

```
function foo(bar) {
    let a = bar;
}
foo(10);
console.log(a); // undefined
```

Private Variables

```
function foo() {
    let a = 1;
}
foo();
console.log(a); // undefined
```

(The word "let" is crossed out.)

What happens is we do not use the `let` keyword?

Private Variables

```
function foo() {
    let a = 1;
}
foo();
console.log(a); // undefined
```

What happens is we do not use the `let` keyword?

JS will try to access the *global variable* `a`

Private Variables

```
function foo() {
    let a = 1;
}
foo();
console.log(a); // undefined
```

(The `let` keyword is crossed out on the slide.)

What happens is we do not use the `let` keyword?

JS will try to access the *global variable* `a`
What if there is no *global variable* `a`?

Private Variables

```
function foo() {
    let a = 1;
}
foo();
console.log(a); // undefined 1
```

(In the code, `let` is crossed out, and `undefined` is crossed out and replaced by `1`.)

What happens is we do not use the `let` keyword?

JS will try to access the *global variable* `a`
What if there is no *global variable* `a`?

Variable *leaking* into the global scope

Exercises

Part_1_Basics/3_functions.js

Catching Errors

- When your code runs you do not generally have full controls on the value of all the variables
- For instance, a user may input a text instead of a number in a form, and this may cause errors

Catching Errors

- When your code runs you do not generally have full controls on the value of all the variables
- For instance, a user may input a text instead of a number in a form, and this may cause errors
- They look ugly:

```
Error: aaa
    at createPageRestructure (/home/capaj/git_projects/looop/project-alpha/back-end/src/controller/PageController.js:18:9)
    at /home/capaj/git_projects/looop/project-alpha/back-end/src/controller/PageController.js:151:18
    at /home/capaj/git_projects/looop/project-alpha/back-end/src/model/TopicModel.js:109:14
    at _fulfilled (/home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:854:54)
    at self.promiseDispatch.done (/home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:883:30)
    at Promise.promise.promiseDispatch (/home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:816:13)
    at /home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:624:44
    at runSingle (/home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:137:13)
    at flush (/home/capaj/git_projects/looop/project-alpha/back-end/node_modules/q/q.js:125:13)
    at _combinedTickCallback (internal/process/next_tick.js:95:7)
    at process._tickDomainCallback (internal/process/next_tick.js:198:9)
```

Catching Errors

- Try and Catch Statements prevent the errors to "bubble up" and let your system fail gracefully.
- Simply wrap the code that may raise an error in a try and catch clause

```
try {
    let a = null;
    a.length;
    // Throws an error and may cause your app to stop.

}
catch(error) {
    a = 'was supposed to be a string.';
    console.log('sorry my bad. Carry on.');
}
```

Main JS Operators Cheatsheet

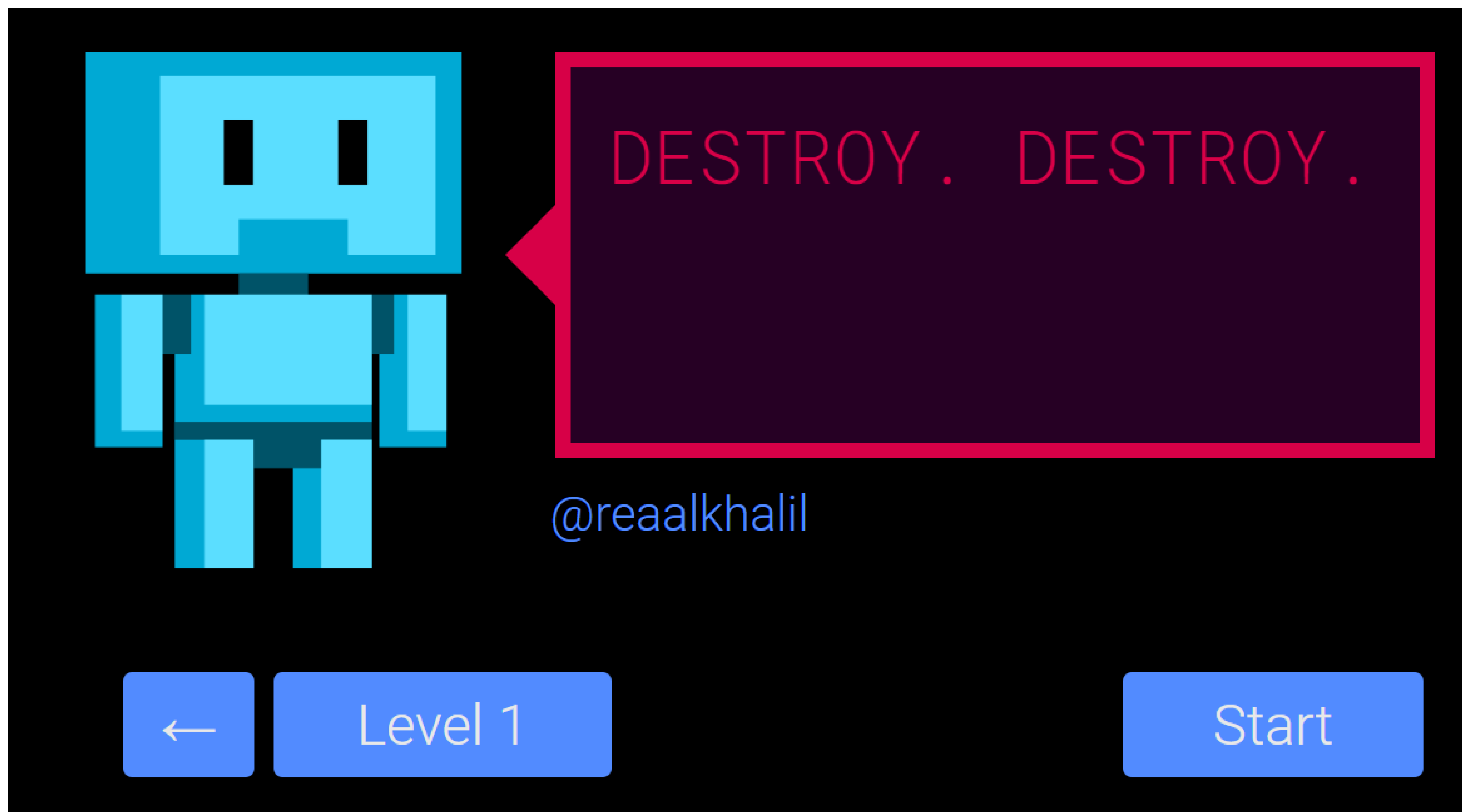
	English Name	Usage	Example
'	Single quote	Wraps strings	'hello'
"	Double quote	Wraps strings	"hello again"
/	Slash	Comments (two in a row)	// comment
;	Semicolon	Ends a line (not mandatory, but recommended)	'hello';
:	Colon	Separates a key and a value in an object	{ key : 1 }
.	Dot	Access an object property (or creates it if not found)	object.key // 1
,	Comma	Separate properties in objects	{ key1 : 1 , key2 : 2 }
()	Parentheses or Brackets	Invoke a function, wrap condition statements	alert('hello'); If (counter > 10) …
[]	Square Parentheses (or Brackets)	Define an array, access elements of the array	[1, 2, 3]; array[0]; // 1

{}	Curly Parentheses (or Brackets)	Define objects, function bodies, blocks of code, string substitutions with backticks strings	{ key : 1 }
			function() { … }
			for (…) { … }
			`I am ${age}`

Exercises

Part_1_Basics/4_try_catch.js
Part_1_Basics/5_final_exercise.js

If You Finish Everything (or if you need a break)



https://lab.reaal.me/jsrobot/