Programming Fundamentals

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

Building Digital Skills: 12-13 March 2020, University of Luzern

Your Instructor: Stefano Balietti

http://stefanobalietti.com

Currently

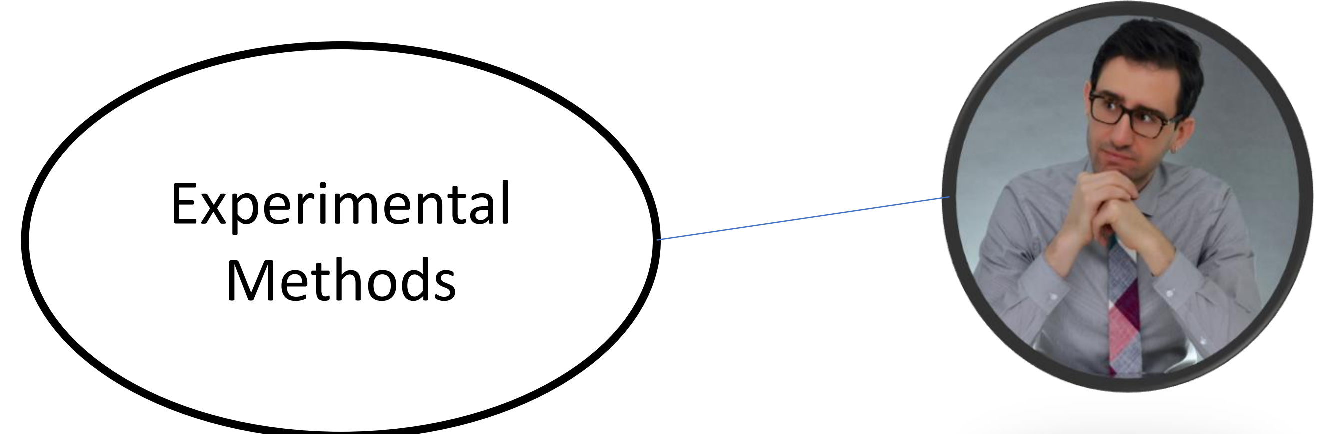
- Fellow in Sociology Mannheim Center for European Social Research (MZES)
- Postdoc at the Alfred Weber Institute of Economics at Heidelberg University

Previously

- Microsoft Research - Computational Social Science New York City
- Postdoc Network Science Institute, Northeastern University
- Fellow IQSS, Harvard University
- PhD, Postdoc, Computational Social Science, ETH Zurich

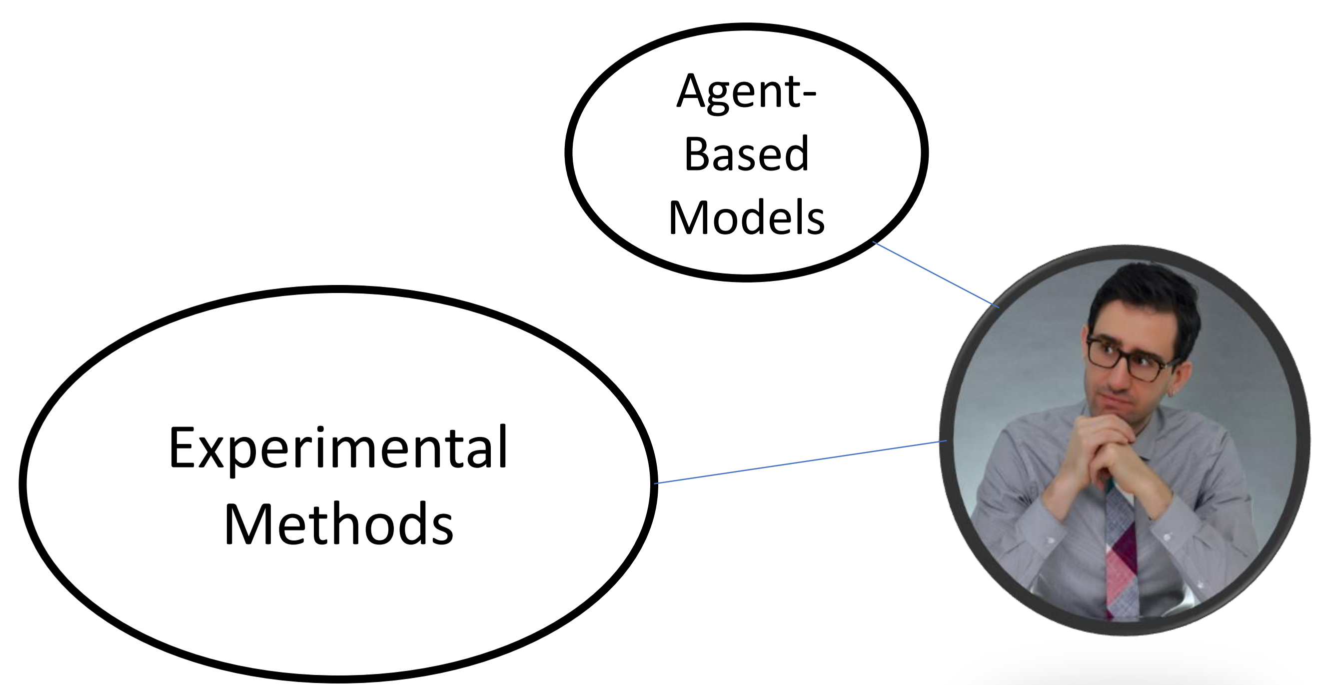
My Methodology

Interface of computer science, sociology, and economics



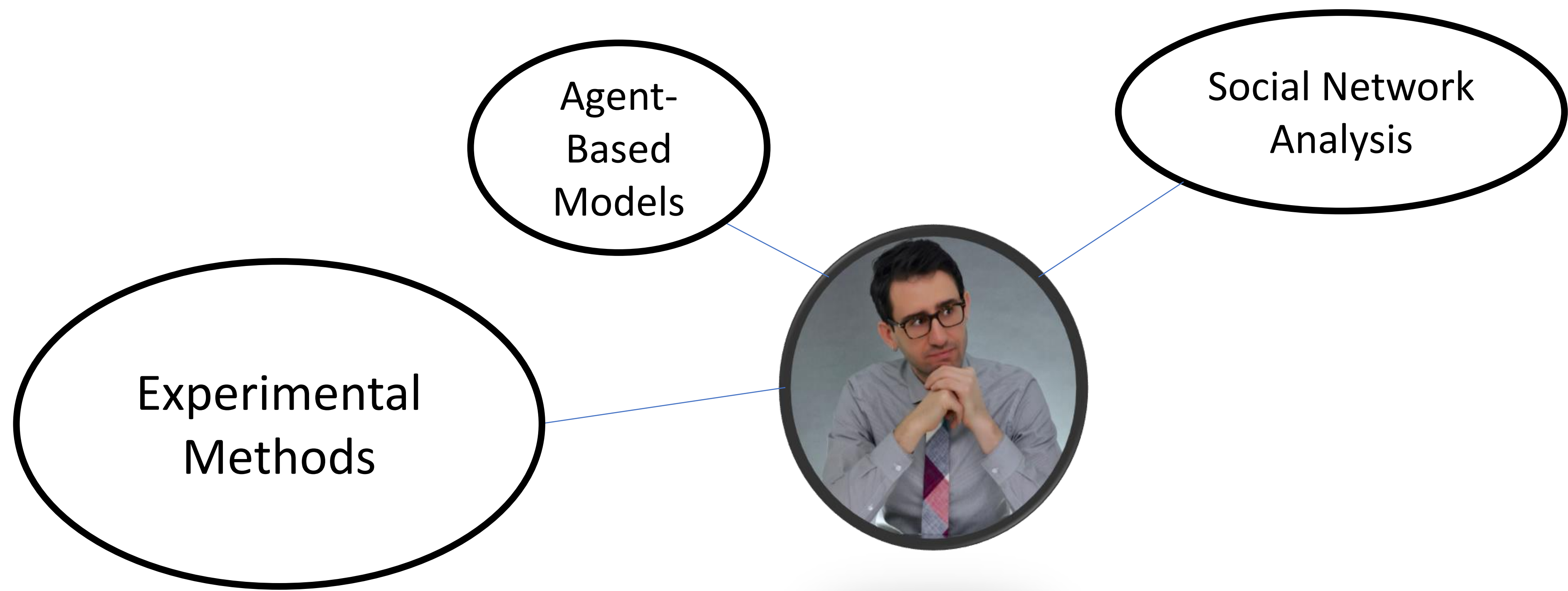
My Methodology

Interface of computer science, sociology, and economics



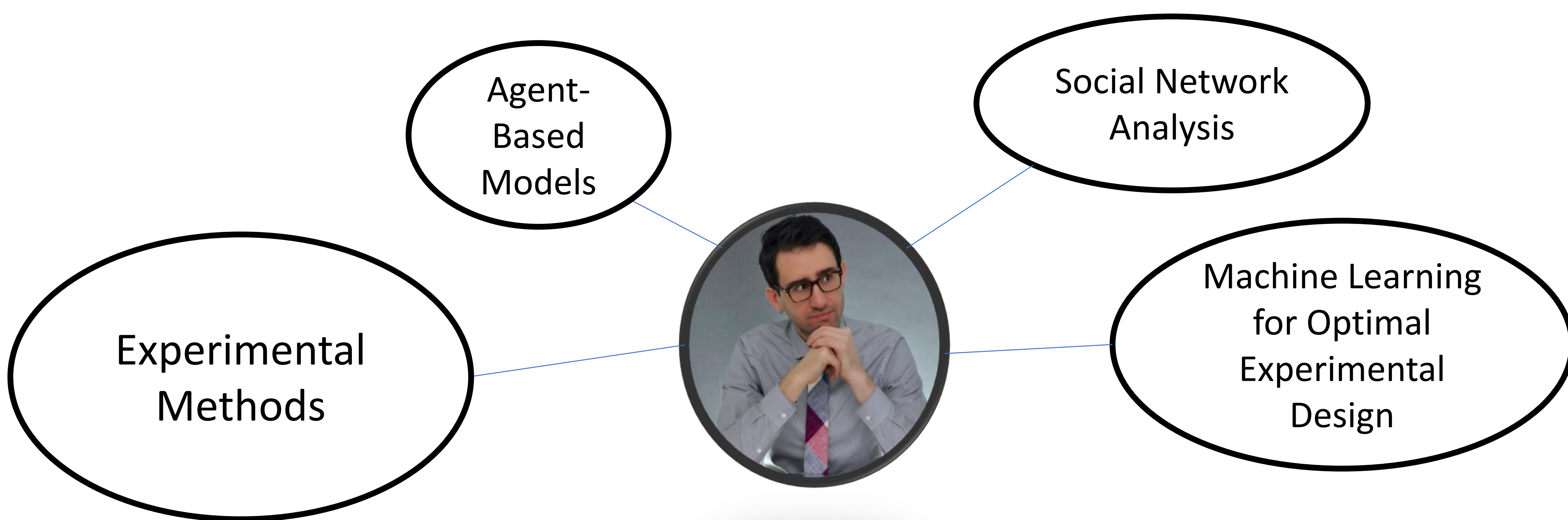
My Methodology

Interface of computer science, sociology, and economics



My Methodology

Interface of computer science, sociology, and economics



Vision
CONSUMPTION TAX
ESTATE TAX
SOCIAL WELFARE
Simulating Societal Processes in Virtual Labs
• Consensus, social influence, and polarization
• Group fairness, inequality, redistribution
• Incentives schemes for collective intelligence
• Optimal experimental design

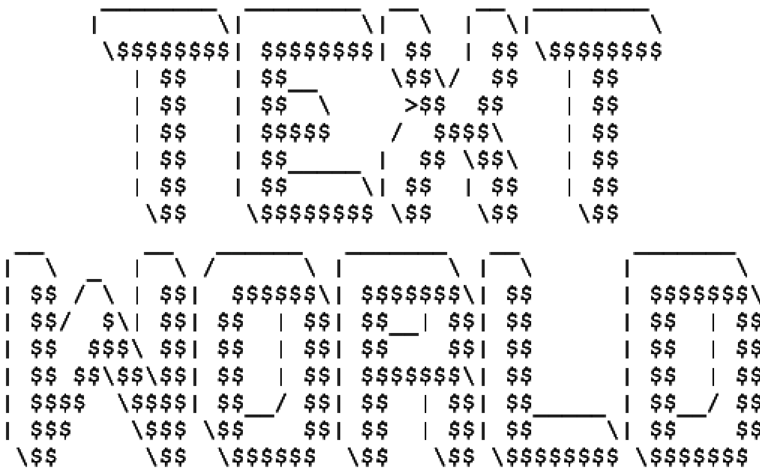
Building Platforms

Garch-in-Gretl (GiG) for econometrics Gretl software

~**5000** weekly downloads

Patterns Configuration Module for Drupal Web Content Management System

2,622 active users, **30,448** downloads



Fast, scalable JavaScript for large-scale real-time online experiments

v5

www.nodegame.org

Goals of the Seminar: Fundamentals of Programming

1. **General programming notions:** variables, data structures, operators, conditional logic, and recursion.

2. **Object oriented programming:** classes, objects, interfaces and inheritance, encapsulation, and abstraction.

3. **Writing high-quality code**: well-establish design patterns, unit-testing, linting, documentation, version control system Git and GitHub, and continuous integration.

Get the Certificate If Attending

1/4 Half Days

Get the Certificate If Attending

2/4 Half Days

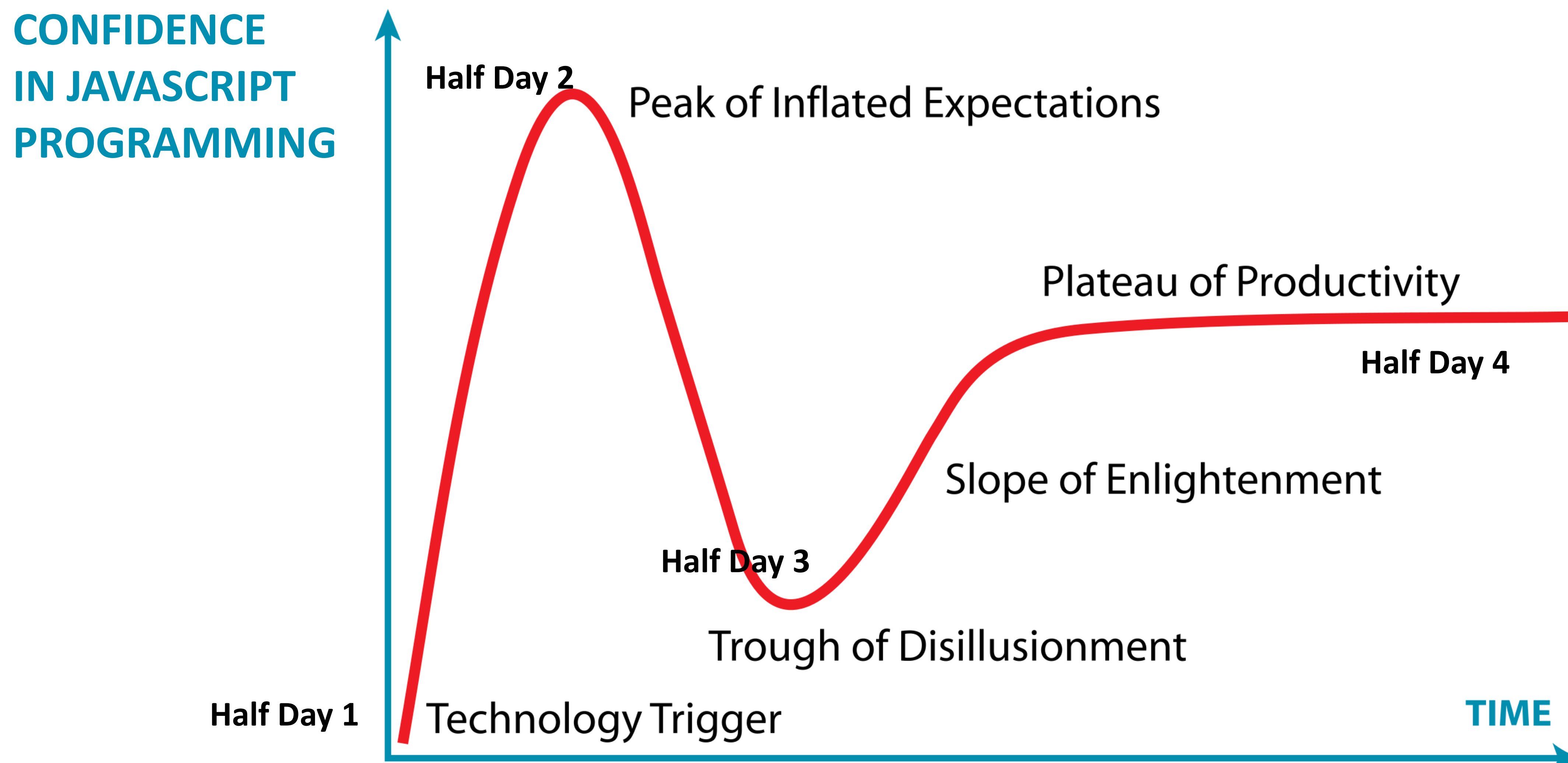
Get the Certificate If Attending

3/4 Half Days

Maximize Personal Knowledge

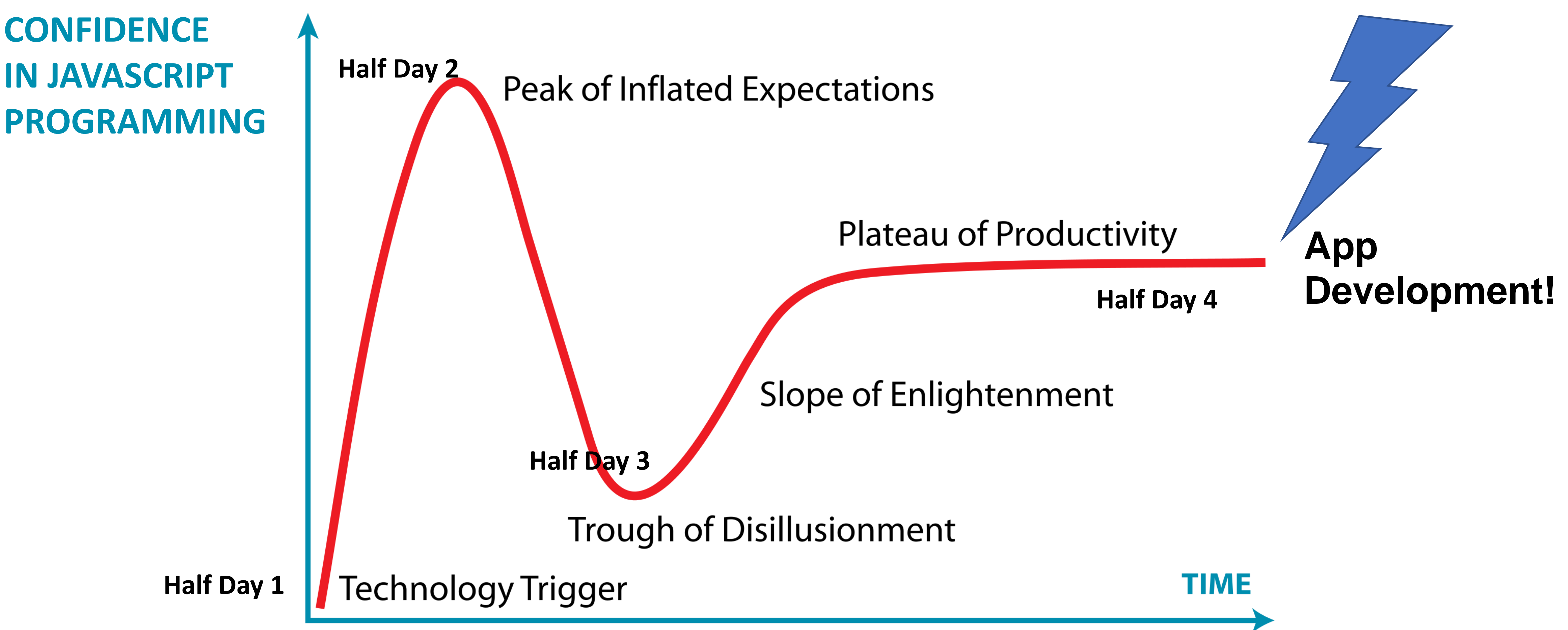
4/4 Half Days

Learning Curve



Adapted from: https://en.wikipedia.org/wiki/Hype_cycle

Learning Curve



Adapted from: https://en.wikipedia.org/wiki/Hype_cycle

This Seminar is Preparatory for the Next One: App Development

1. **Web App:** the golden triad: HTML, CSS, and JavaScript; asynchronous programming, Node.JS and NPM, REST API calls; introduction to Web frameworks: JQuery, Twitter Bootstrap, SASS; cloud providers.

This Seminar is Preparatory for the Next One: App Development

1. **Web App:** the golden triad: HTML, CSS, and JavaScript; asynchronous programming, Node.JS and NPM, REST API calls; introduction to Web frameworks: JQuery, Twitter Bootstrap, SASS; cloud providers.

   1. *Mobile App*: transform the Web App into a Mobile App with Apache Cordova (https://cordova.apache.org)
   2. *Browser Extension*: create a simple Chrome-based extension (https://developer.chrome.com/extensions)
   3. *Behavioral Experiment*: create a simple game theory experiment with the nodeGame platform (https://nodegame.org).
   4. *Decentralized App*: Introduction to Solidity (JavaScript-based language) to program blockchain applications on the Ethereum platform (https://ethereum.org)

Programming Fundamentals for You

- Briefly introduce yourself

- What is your level of computer programming?

- Do you know JavaScript already?

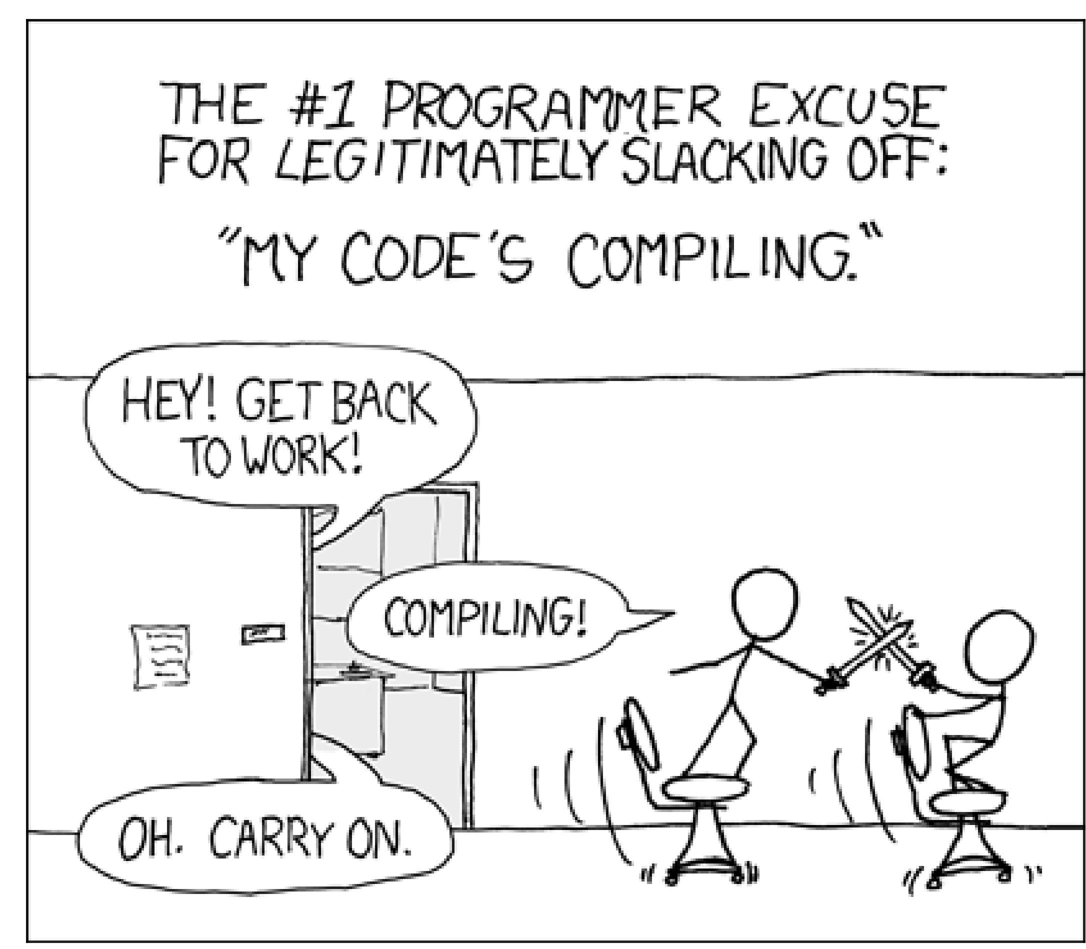
- What are you looking to learn in this course?

What is JavaScript?

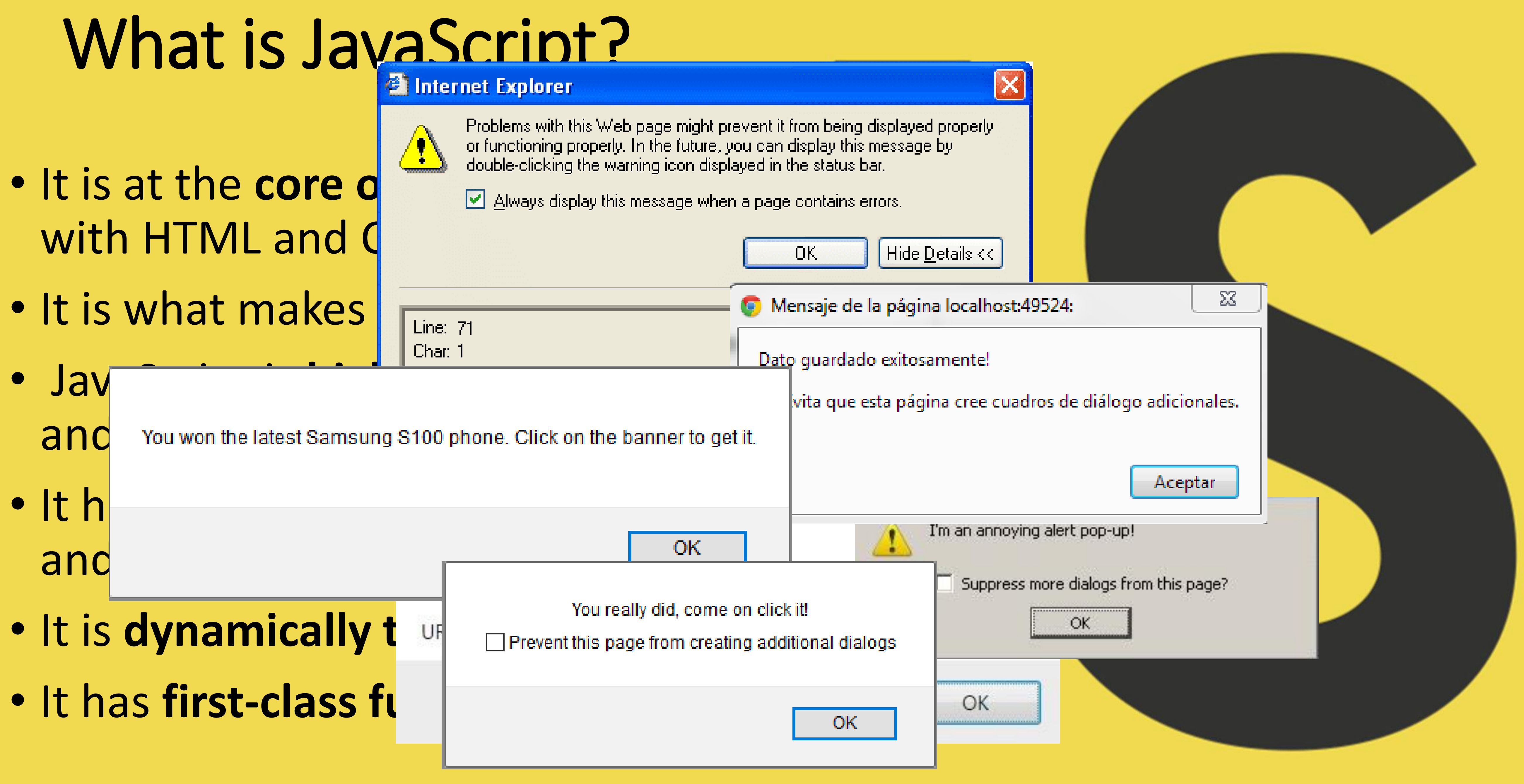
What is JavaScript?

Let's Start with What is NOT

- **JavaScript is NOT Java**, similar names, but for the rest rather different
- JAVA is a **compiled** language and while JavaScript is **interpreted**
- JAVA is generally more complex
- JAVA is fading (?)



https://www.xkcd.com/378/


What is JavaScript?
• It is at the core o
with HTML and C
• It is what makes
• Jav
and
• It h
and
• It is dynamically t
• It has first-class fu
Internet Explorer
Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.
Always display this message when a page contains errors.
OK
Hide Details <<
Line: 71
Char: 1
Mensaje de la página localhost:49524:
Dato guardado exitosamente!
vita que esta página cree cuadros de diálogo adicionales.
Aceptar
You won the latest Samsung S100 phone. Click on the banner to get it.
OK
I'm an annoying alert pop-up!
Suppress more dialogs from this page?
OK
UR
You really did, come on click it!
Prevent this page from creating additional dialogs
OK
OK

What is JavaScript?

- It is at the **core of Web technologies**, with HTML and CSS
- It is what makes the pages interactive
- JavaScript is **high-level**, **scripted,** and **multi-paradigm**.
- It has **prototypical object-orientation** and **curly-bracket** syntax
- It is **dynamically typed**
- It has **first-class functions**

JavaScript

- JavaScript was developed in May 1995 by *Brendan Eich* for Netscape Communications Corp
- Was created in **10 days** in order to accommodate the Navigator 2.0 Beta release
- Initially called **Mocha**, later renamed **LiveScript** in September, and later **JavaScript** in the same month

https://en.wikipedia.org/wiki/Brendan_Eich

JavaScript

- Microsoft introduced **JScript** as reverse-engineered implementation of Netscape's JavaScript in 1996 in Internet Explorer 3

- In 1996 Netscape submitted JavaScript to European Computer Manufacturers Association (ECMA) to create and industry standard

- In 1997 **ECMAScript** was released

- Between 1997 and 2009 5 standard have been released

- ***July 2015 ECMASCRIPT V6 released.***

JavaScript Is Constantly Updated

- ES2016 a.k.a. ES7
- ES2017 a.k.a. ES8
- ES2018 a.k.a. ES9
- ES2019 a.k.a. ES10
- ES2020 a.k.a. ES11

Node.JS

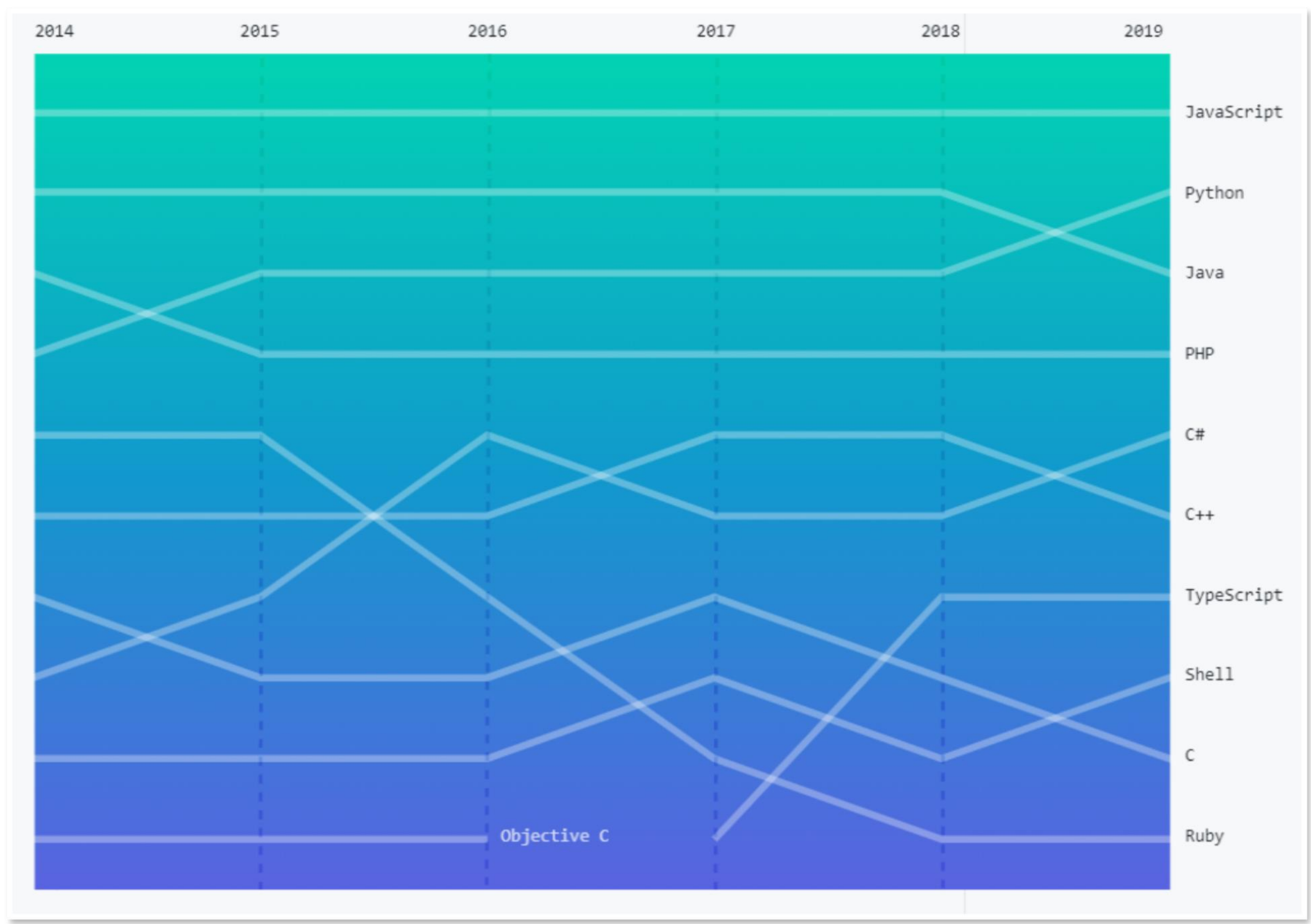
- **Node.JS** was created by *Ryan Dahl* and other developers working at Joyent in 2009
- Combination of Google's V8 JavaScript engine, an event loop, and a low-level I/O API
- **npm**, the node package manager, in 2011
- Versions: 0.10, 0.12, 4.0 … 12.0!

Advantages of Node.JS

- **Easy to Learn.** If you know JavaScript...otherwise easy to get started, but careful of pitfalls!
- **Full Stack JS**. Single programming language for client side (e.g., browser) and backend (i.e., server)
- **Freedom to Develop Apps.** Web apps and mobile apps, browser extensions, games, decentralized apps…
- **Higher performance.** Process several request *simultaneously* thanks to the *asynchronous non-blocking paradigm;* highly scalable *horizontally* and *vertically*
- **Many Frameworks and Testing tools.** Bootstrap, jQuery, React, Mocha, Ganache, nodeGame…
- **Huge and Active Community.**

https://www.quora.com/What-are-the-advantages-of-node-js-1

JavaScript is #1 Language on Github



https://octoverse.github.com/

GitHub.com

40 m+

developers on GitHub, including 10M new users in 2019.*

44 m+

repositories created in the last year—and 44% more developers created their first repository in 2019 than in 2018.*

87 m+

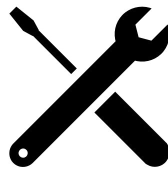
pull requests merged in the last year—and 28% more developers opened their first pull request in 2019 than in 2018.*

20 m+

issues closed in the last year. That's a lot of decisions made, bugs fixed, and boxes checked.*

https://octoverse.github.com/

Hands On

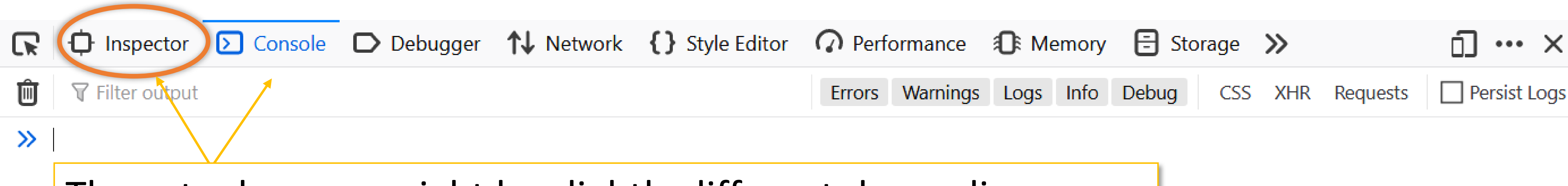


Open the JavaScript console of your browser:

(ctrl+shift+I or Right Click/Inspect Element)

Hands On

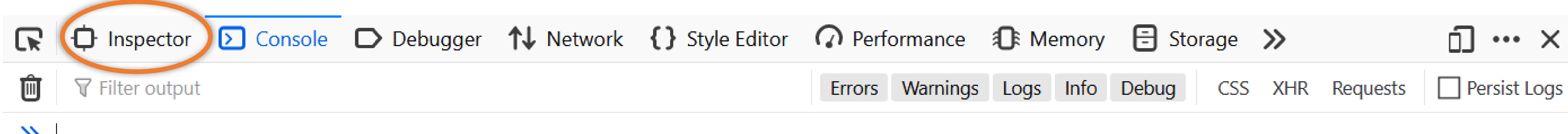
The Inspector is where you can visualize the live DOM (Document Object Model) and make changes, including CSS (Cascading Style Sheets) changes.



The actual names might be slightly different depending on the browser version and language. For instance, "Inspector" is sometimes called "Elements."

Hands On

The Inspector is where you can visualize the live DOM (Document Object Model) and make changes, including CSS (Cascading Style Sheets) changes.



Hands On

Try to open different web sites, how does the content of the console changes?

Hands On

bahn.de: lots of messy output, including your first and last name

```
▼ Incoming message 'load'                                              common.js:46:37
  ▼ load(Kunde)                                                        common.js:46:37
  ⓘ [iLogic] Connectivity is  Connected                               common.js:137:46
    [Cache] Read  Kunde  from cache (storage):  { "khash" :            common.js:114:51
    "48fef9ed13945b104be7f743779d182f76de070da█████" , "name" : { "nachname" : "Balietti"
    , "vorname" : "Stefano" , "anrede" : "0" , "titel" : "1" , "login" : "█████}}
  ⓘ [iLogic] Data is in cache but outdated/expired.                    common.js:137:46
  ⓘ [iLogic] -> loading it from server.                                common.js:137:46
  ⓘ [iLogic] Ajax call load(Kunde).                                    common.js:137:46
▼ Processing AJAX response for load(Kunde)                             common.js:46:37
   [iLogic]   response =                                               common.js:114:51
   ▶ Object { status: 200, content: "{ \"khash\" :
   \"48fef9ed13945b104be7f743779d182f76de070da█████\" , \"name\" : { \"nachname\" :
   \"Balietti\", \"vorname\" : \"Stefano\" \"anrede\" : \"0\" , \"titel\" : \"1\" , \"login\" :
   \█████\"}}", etag: "jfu-ET6GJJDvpkdIkd8█████" }
   [Cache] Wrote  Kunde  to cache (storage):  { "khash" :              common.js:114:51
   "48fef9ed13945b104be7f743779d182f76de070da█████" , "name" : { "nachname" : "Balietti" ,
   "vorname" : "Stefano" , "anrede" : "0" , "titel" : "1" , "login" : █████}}
```

Hands On

facebook.com: a warning to not fall victim of social engineering phishing attacks

```
 .d8888b.  888                       888
d88P  Y88b 888                       888
Y88b.      888                       888
 "Y888b.   888888  .d88b.  88888b.   888
    "Y88b. 888    d88""88b 888 "88b  888
      "888 888    888  888 888  888  Y8P
Y88b  d88P Y88b.  Y88..88P 888 d88P
 "Y8888P"   "Y888  "Y88P"  88888P"   888
                           888
                           888
                           888
```

This is a browser feature intended for developers. If someone told you to copy and paste something here to enable a Facebook feature or "hack" someone's account, it is a scam and will give them access to your Facebook account.

See *https://www.facebook.com/selfxss* for more information.

Hands On

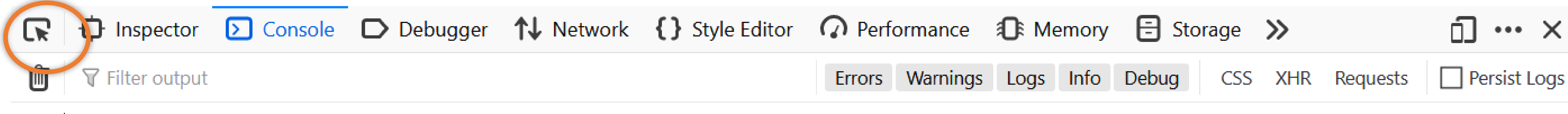
```
<!--
         0000000                         000        0000000
       111111111      11111111100          000      111111111
       00000        111111111111111111      00000      000000
       000        1111111111111111111111111100000         000
       000        1111       1111111111111111100          000
       000         11       0     1111111100              000
       000          1      00             1               000
       000               00      00       1               000
       000             000    00000       1               000
    00000            0000  00000000       1                00000
  11111            000 00    000000      000                 11111
    00000          0000      000000     00000              00000
       000        10000      000000      000              0000
       000        00000      000000       1               000
       000        000000     10000        1     0         000
       000        1000000 00              1    00         000
       000         1111111                1 0000          000
       000          1111111100           000000           000
       0000          111111111111111110000000            0000
       111111111        111111111111100000          111111111
         0000000              00000000              0000000

      NYTimes.com: All the code that's fit to printf()
      We're hiring: https://nytimes.wd5.myworkdayjobs.com/Tech
   -->
```

nytimes.com: a job offer!

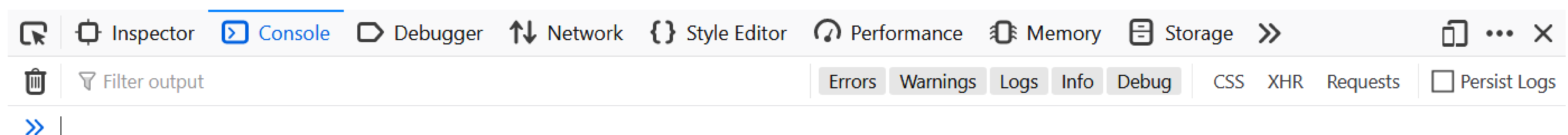
Hands On

Try to open different web sites, how does the content of the console changes?

Clear any pre-existing output: click on button or type `clear()`



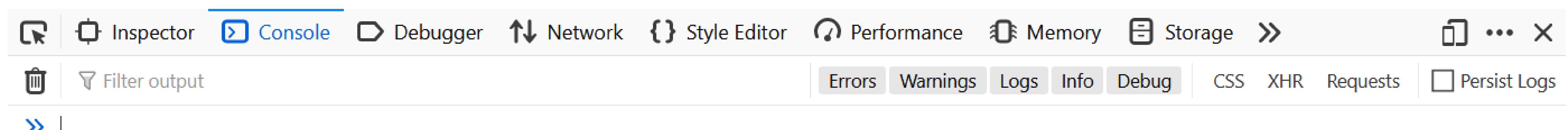
Hands On



Type something in the console using the command:

```
console.log('This is my very own text');
```

Hands On



Now do it... feel the power of the alert!

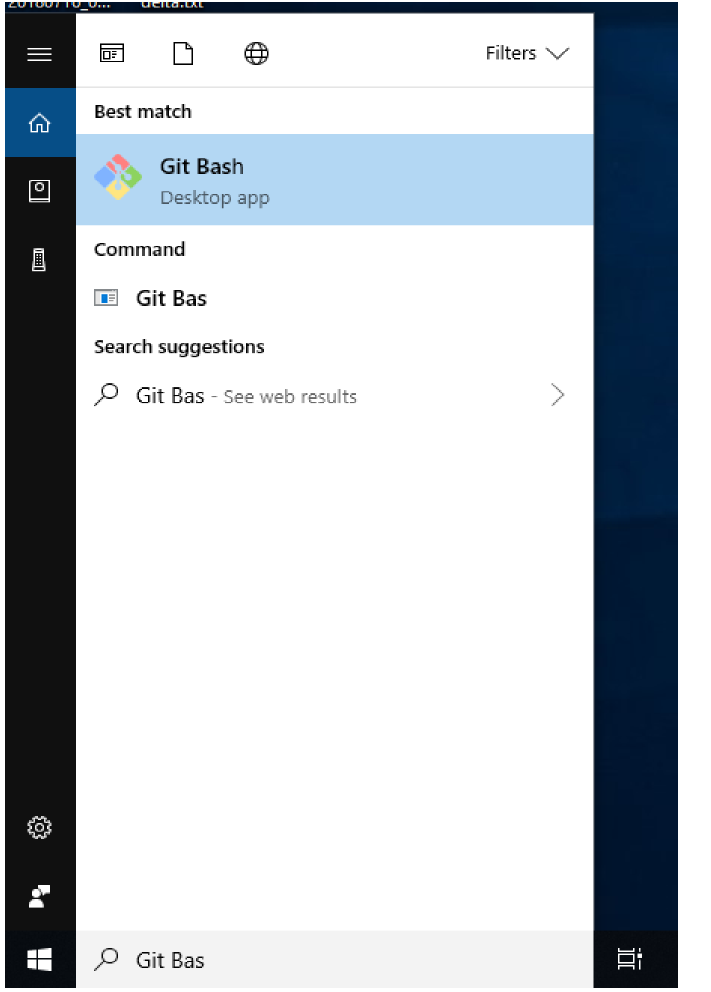
```
alert('This is my revenge!');
alert('again and again...');
```

Preparation: Have You Got?

- An account on **GitHub**: https://github.com/
- The text editor **Atom** https://atom.io/
- The environment **Node.JS** https://nodejs.org/en/
- The version control system **Git** https://git-scm.com/

Are Git and Node.JS Installed Properly?

Open Git Bash (Win) or a Terminal (OSX/Linux)



Can you reproduce the following or a similar output?

Git installed

Node.JS installed

```
MINGW64:/c/Users/balistef

balistef@mzes072 MINGW64 ~
$ git --version
git version 2.22.0.windows.1

balistef@mzes072 MINGW64 ~
$ node --version
v12.15.0

balistef@mzes072 MINGW64 ~
$
```

Seminar Structure

Seminar Structure

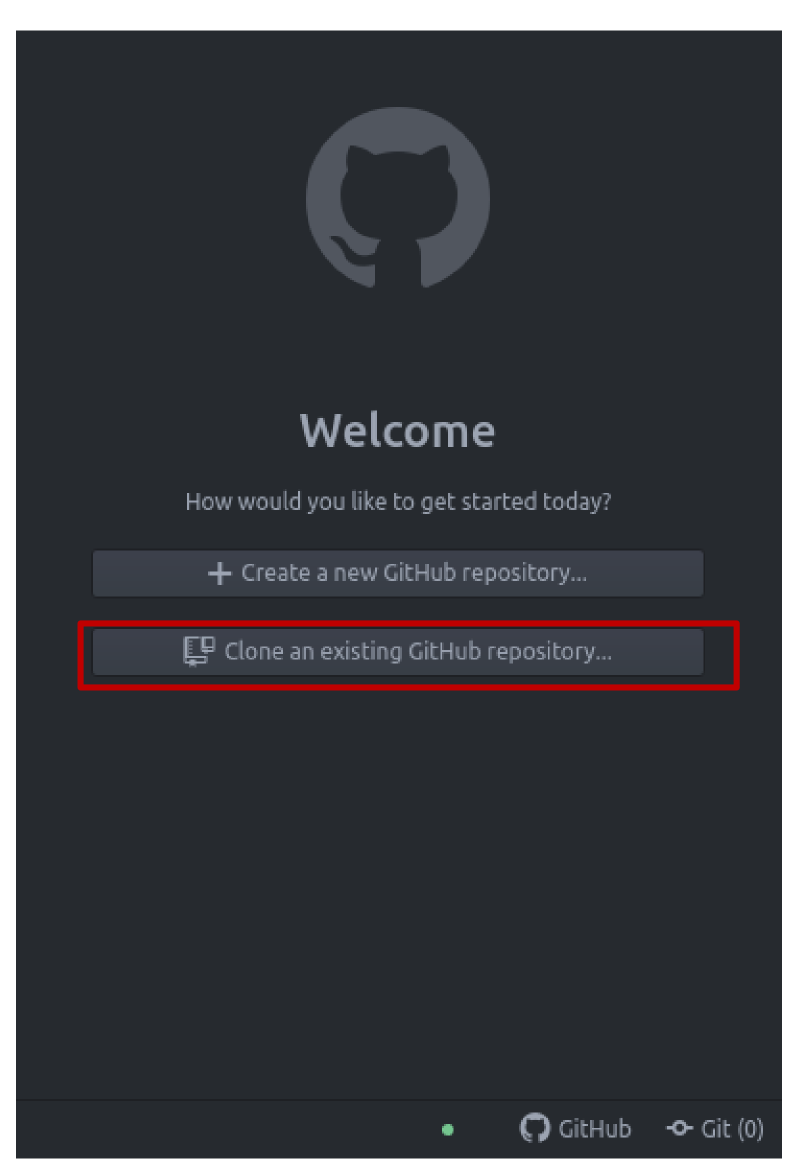
Exercises extend what is covered in the slides. Some of you will find some exercises easy and others more difficult. Don't worry if you don't finish them all:
Do them at your own pace!

Exercise 0: Download the exercises

Or better said:

clone the GitHub repository of the exercises

Exercise 0: Download the exercises



Exercise 0B: Configure Atom

Let's open the slide deck "Configure Atom"

How I Learnt JavaScript

Great tutorial from novice to JavaScript Ninja:
http://javascript.info/

10+years ago…

Part 1: Basics

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.

Variables

http://javascript.info/

Variables

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

The value of assignment: a string wrapped in quotes

Variables

http://javascript.info/

Variables

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

The semicolon signals that the command is finished.

http://javascript.info/

Variables

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

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!';
```

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***

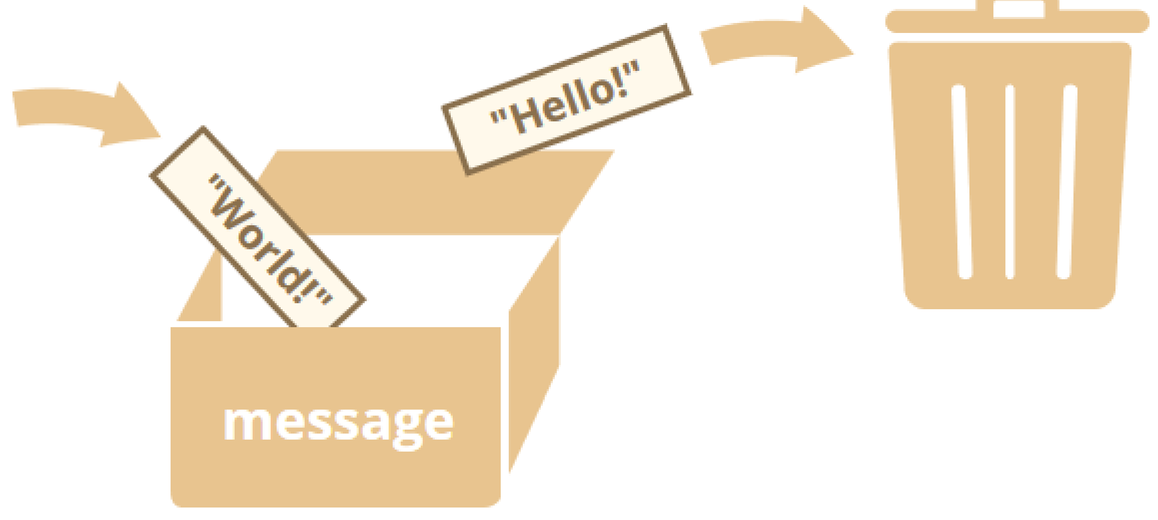
Notice we don't use **let** again, otherwise it will throw an error.

Variables

http://javascript.info/

Variables

```
let message = 'Hello!';


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


alert(message);
```



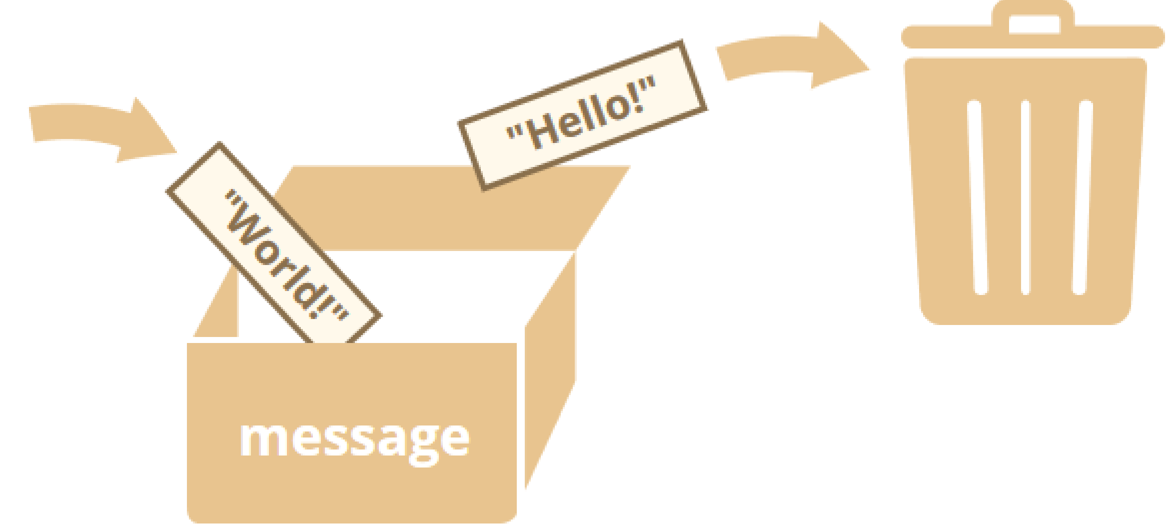
Variables

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/

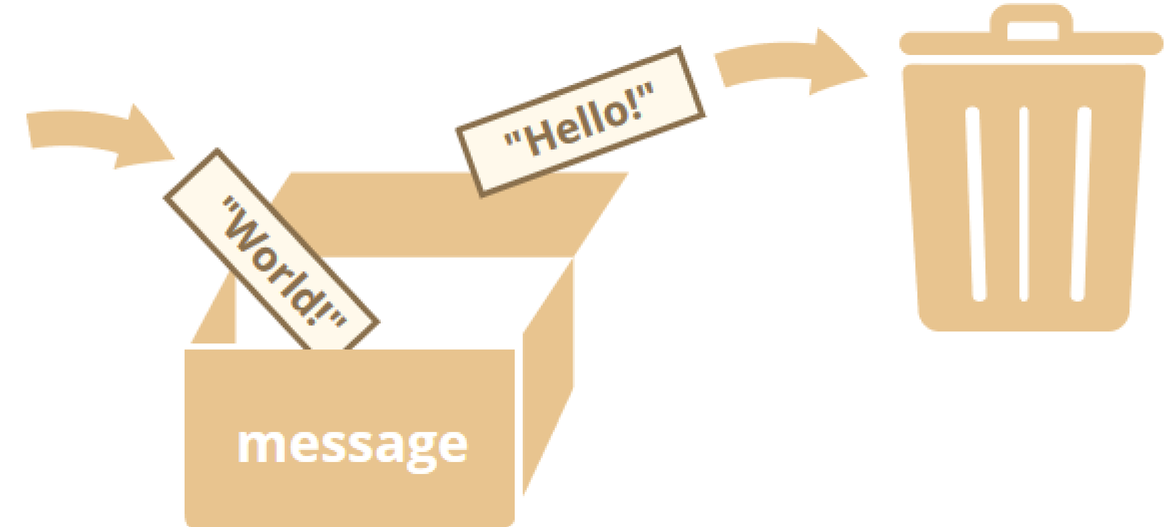
Variables

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);
```



http://javascript.info/

Variables

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

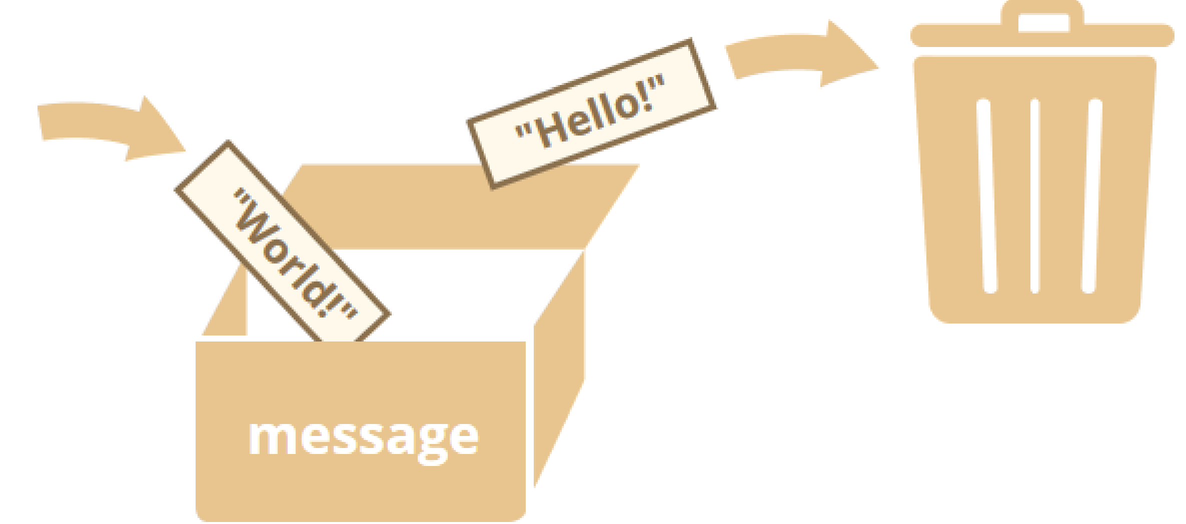
```
let message = 'Hello!';
```

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

These are primitive types

```
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)
```

These are composite types

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);
```

Variable 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
}
```

! (NOT)

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

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

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

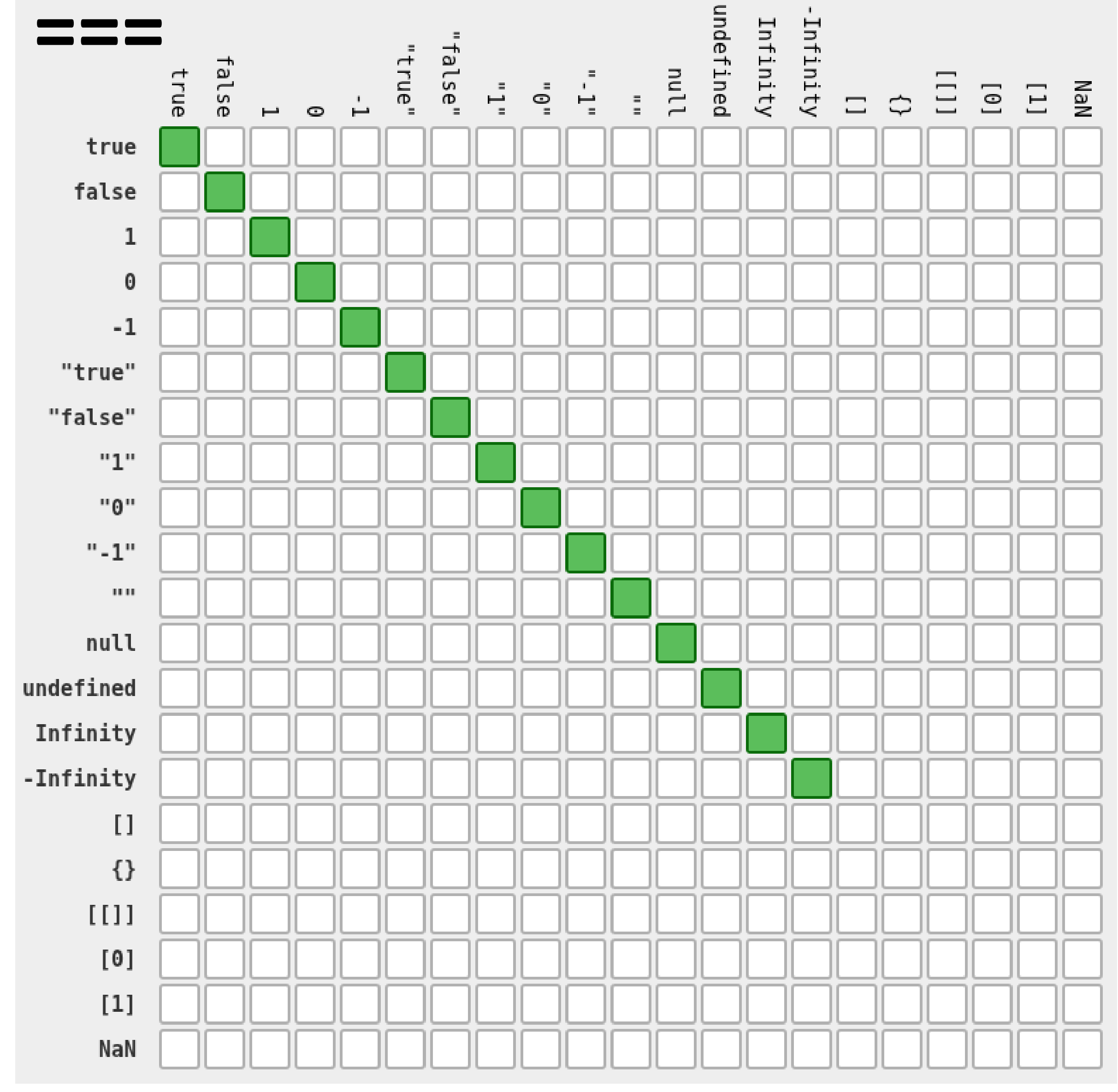
Like assignments, comparisons have an operator which separates a left-hand side term and right-hand side term, e.g., 3 > 1, and they return a Boolean value (true or false).

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/

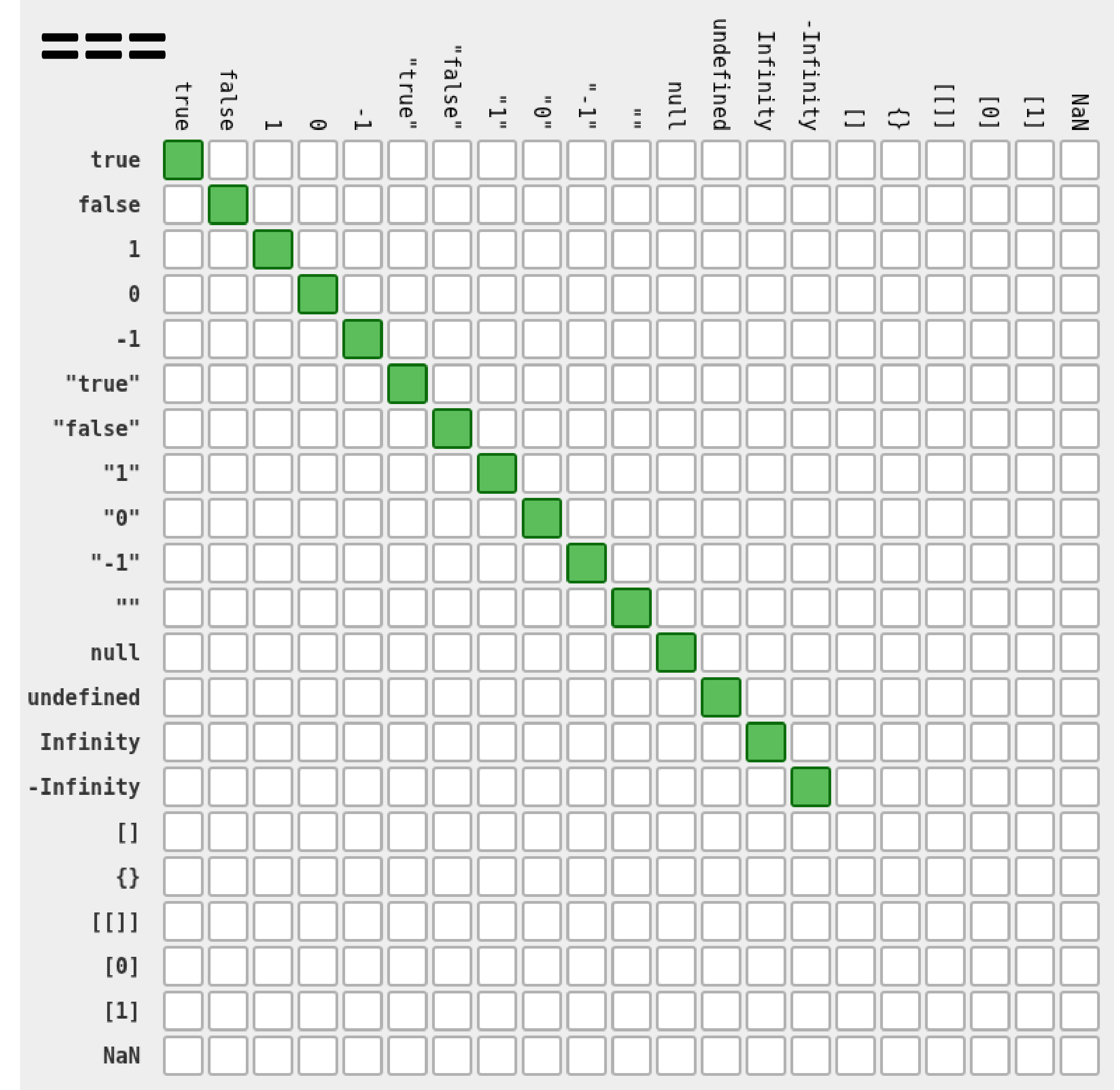


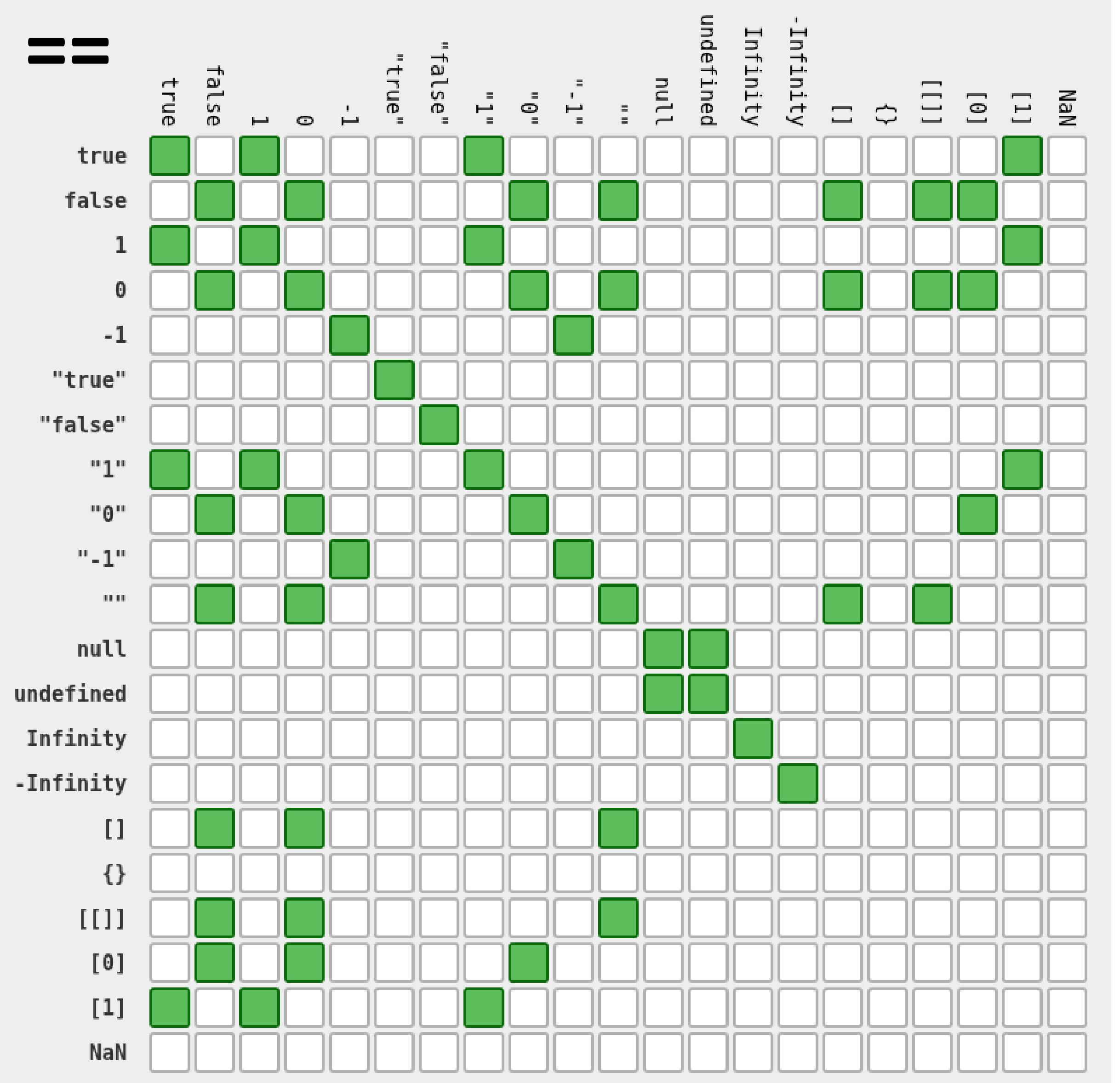
- If the cell is filled, it means the result of a comparison is true, otherwise false
- The table on the diagonal reads:

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

Variable Comparison: === vs ==

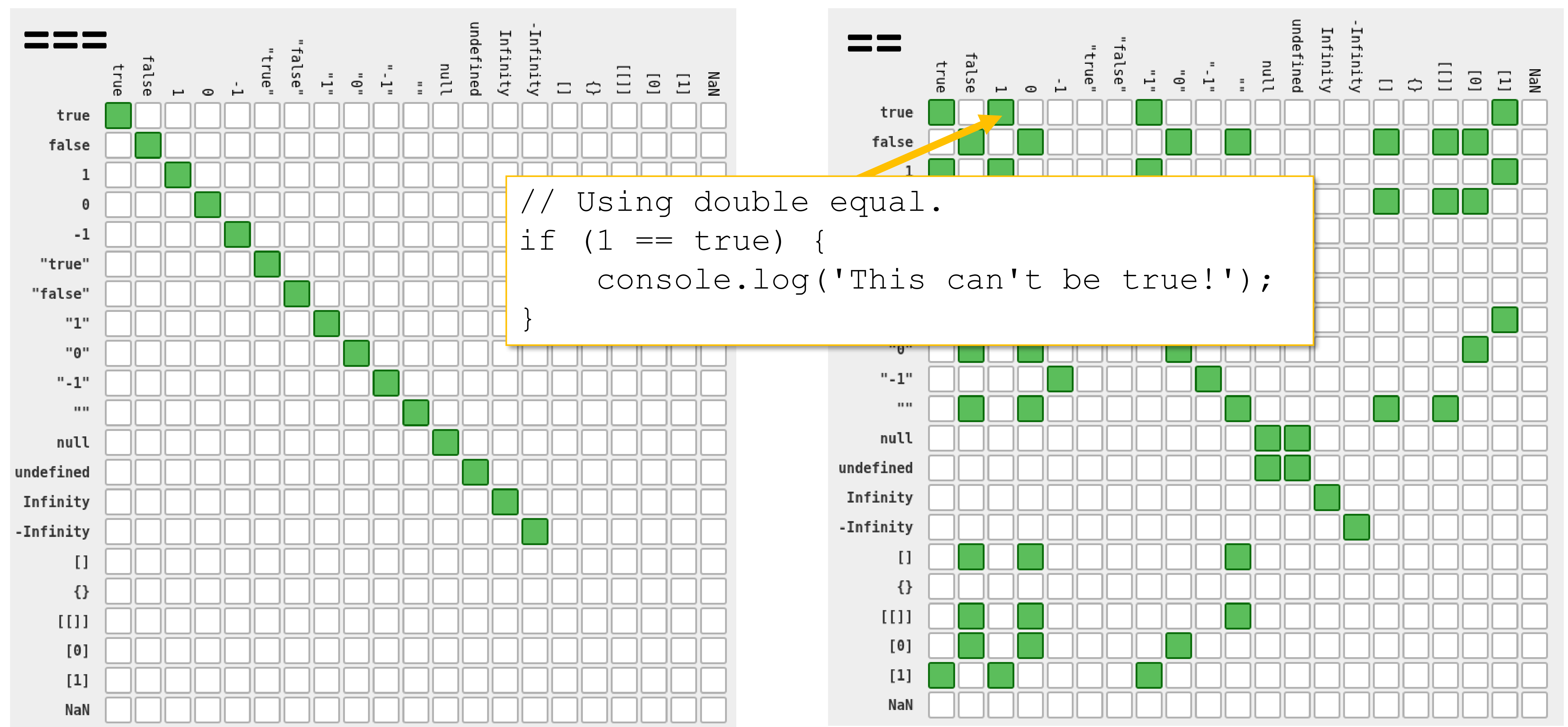
https://dorey.github.io/JavaScript-Equality-Table/





Variable Comparison: === vs ==

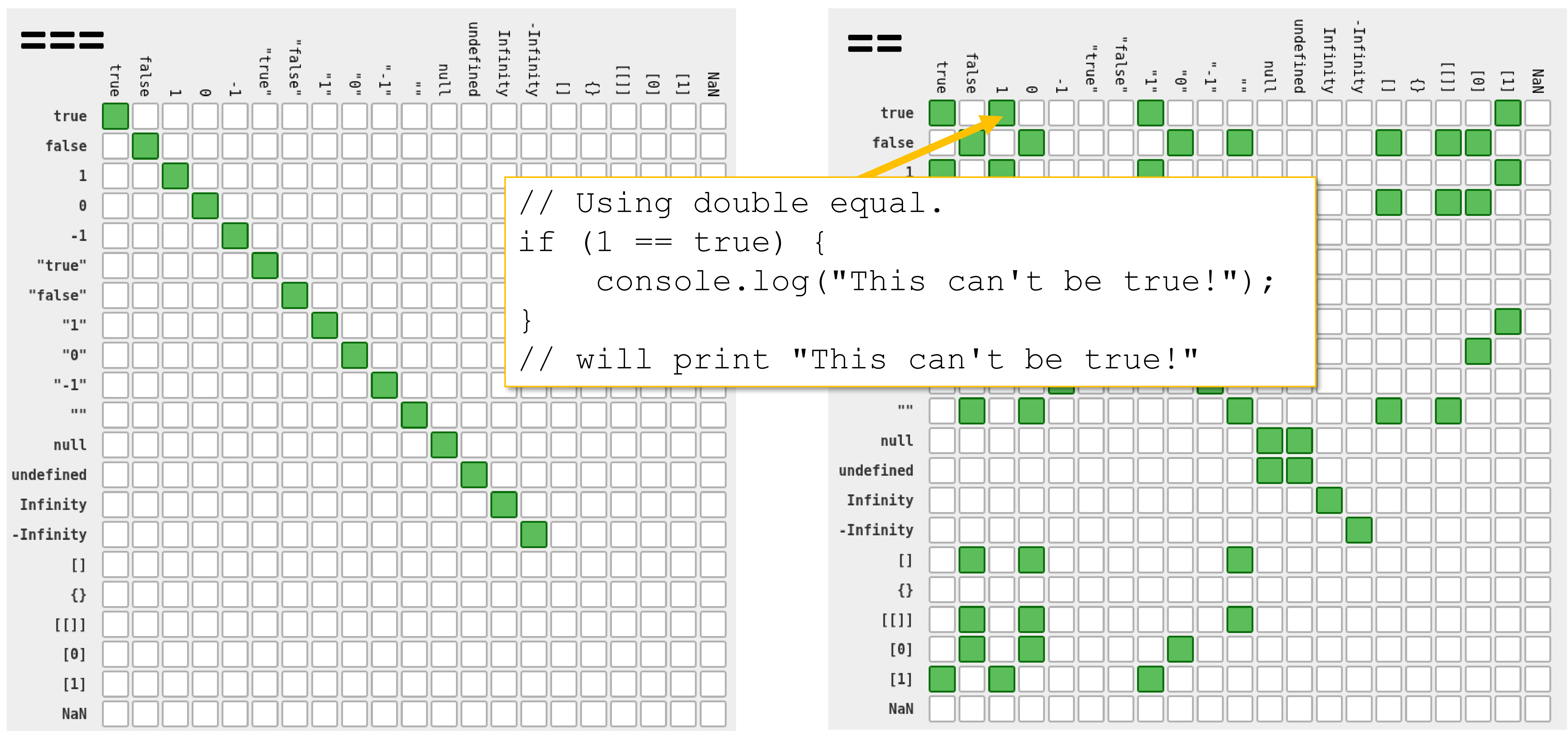
https://dorey.github.io/JavaScript-Equality-Table/




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

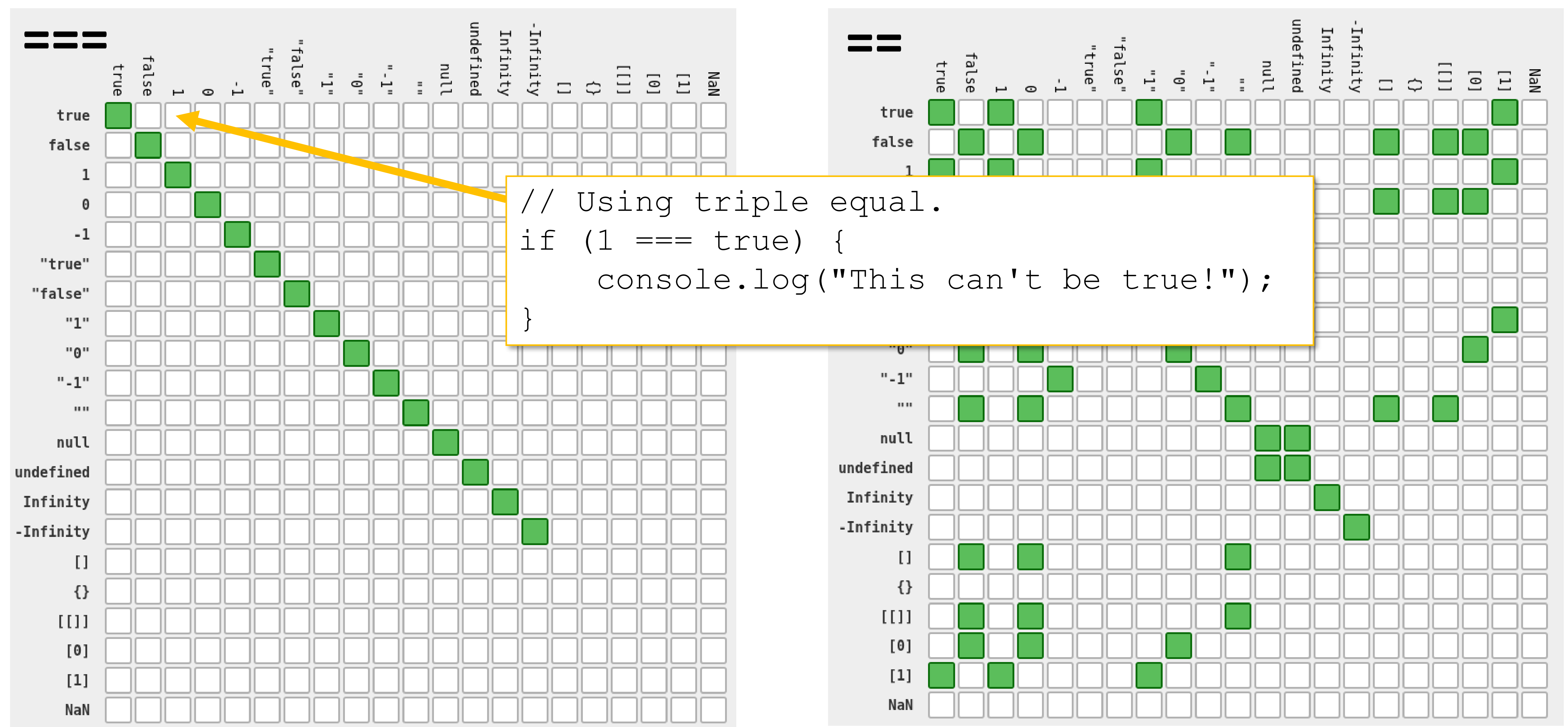
Variable Comparison: === vs ==

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



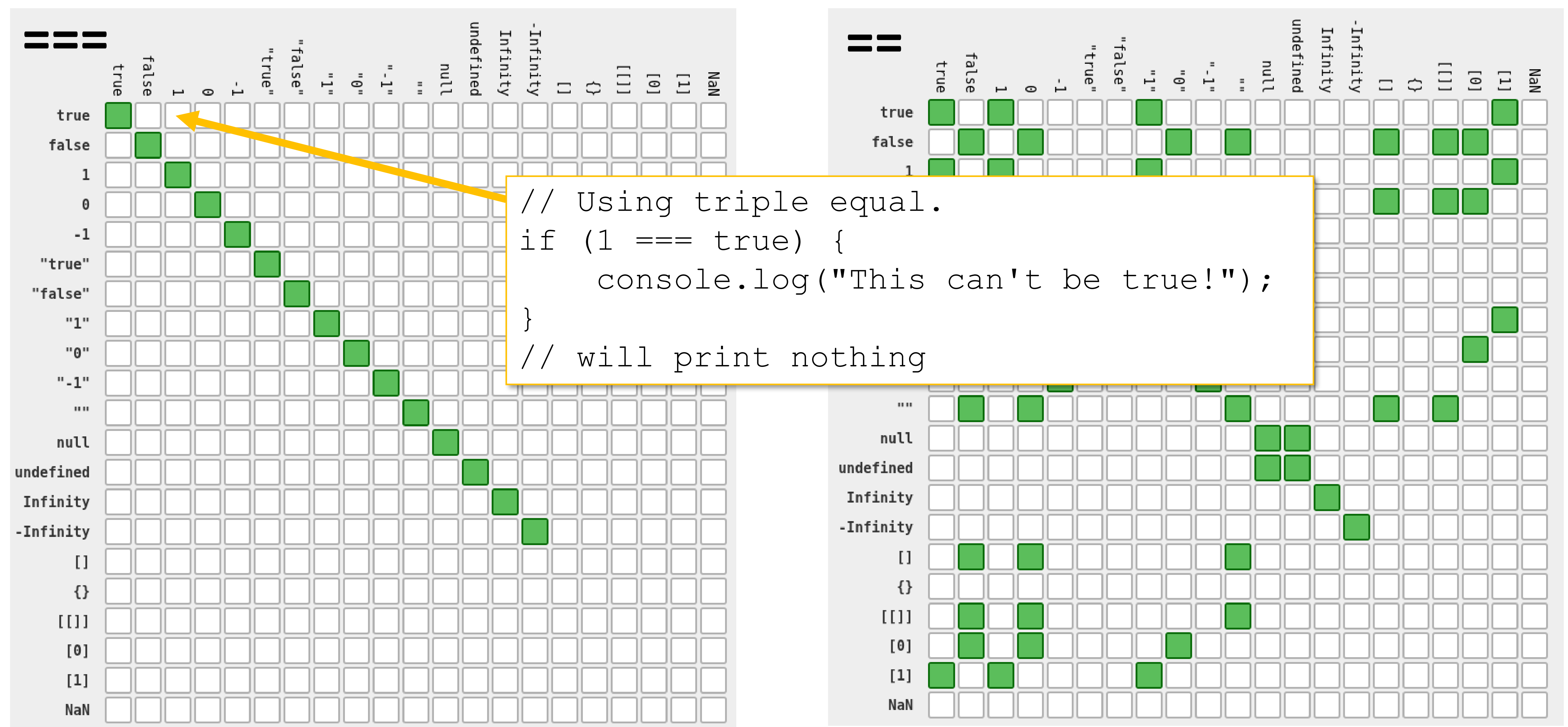
Variable Comparison: === vs ==

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



Variable Comparison: === vs ==

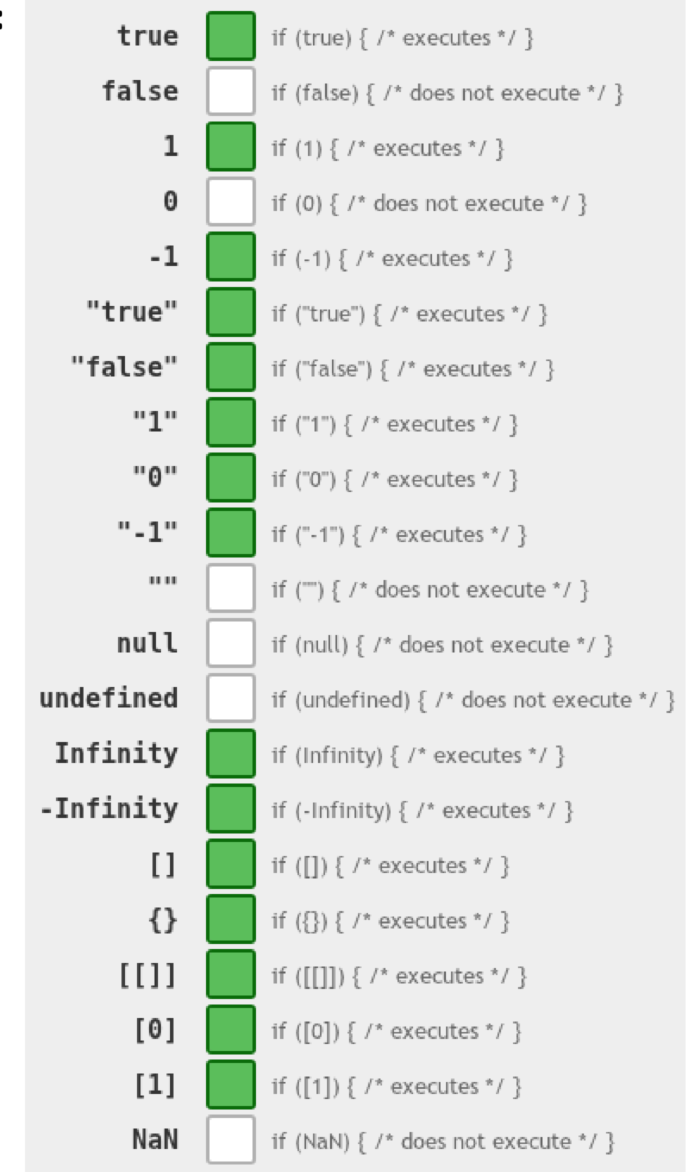
https://dorey.github.io/JavaScript-Equality-Table/



Variable Comparison: === vs ==

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

==



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';
    let 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/

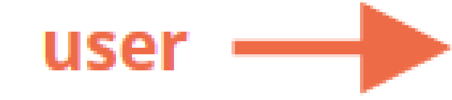
```
// We now add a new property
// Note! JavaScript is case sensitive
user.isAdmin = true;
// Delete an existing one.
delete user.age;
```

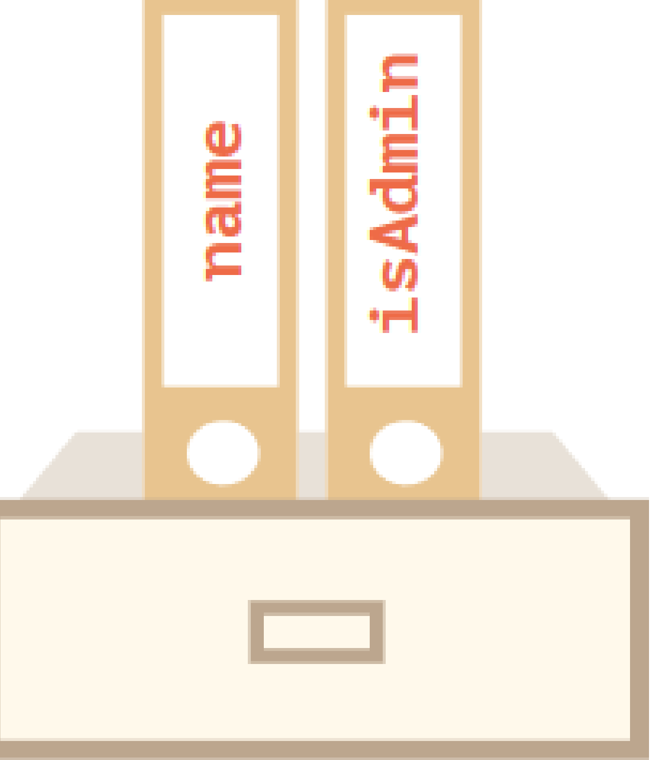
Objects

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.



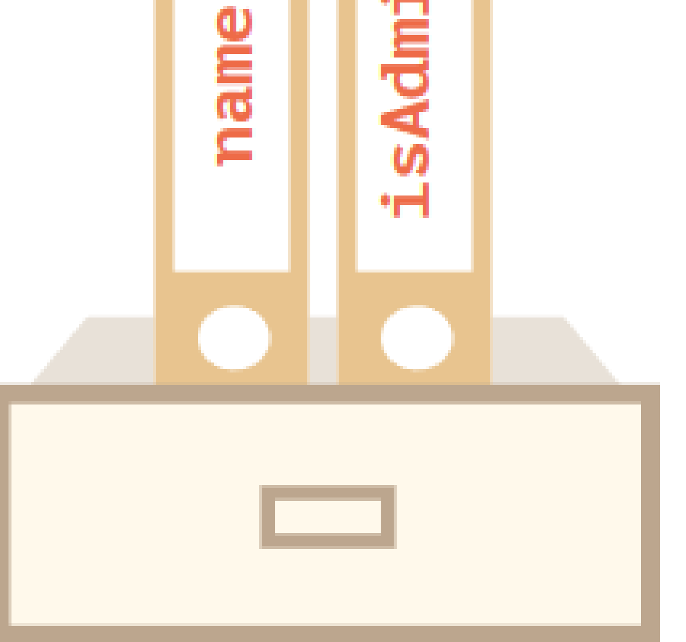


Objects

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)

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

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

- The + sign is used to concatenate strings

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.r
    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 + 'per

    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 + 'per
    
    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';
}
var 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';
}
var 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

```
var message  = 'I like ';
// This is a "for loop".
for (var 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

```
var message  = 'I like ';
// This is a "for loop".
for (var 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

```
var 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;
}

var 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;
}

var 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;
}

var 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;
}

var 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;
}

var 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;
}

var 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;
}

var 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;
}

var 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
```

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

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`

Private Variables

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

(The `let` keyword is crossed out.)

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
```

(`let` and `undefined` are crossed out.)

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

Variable *leaking* into the global scope

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

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.');
}
```

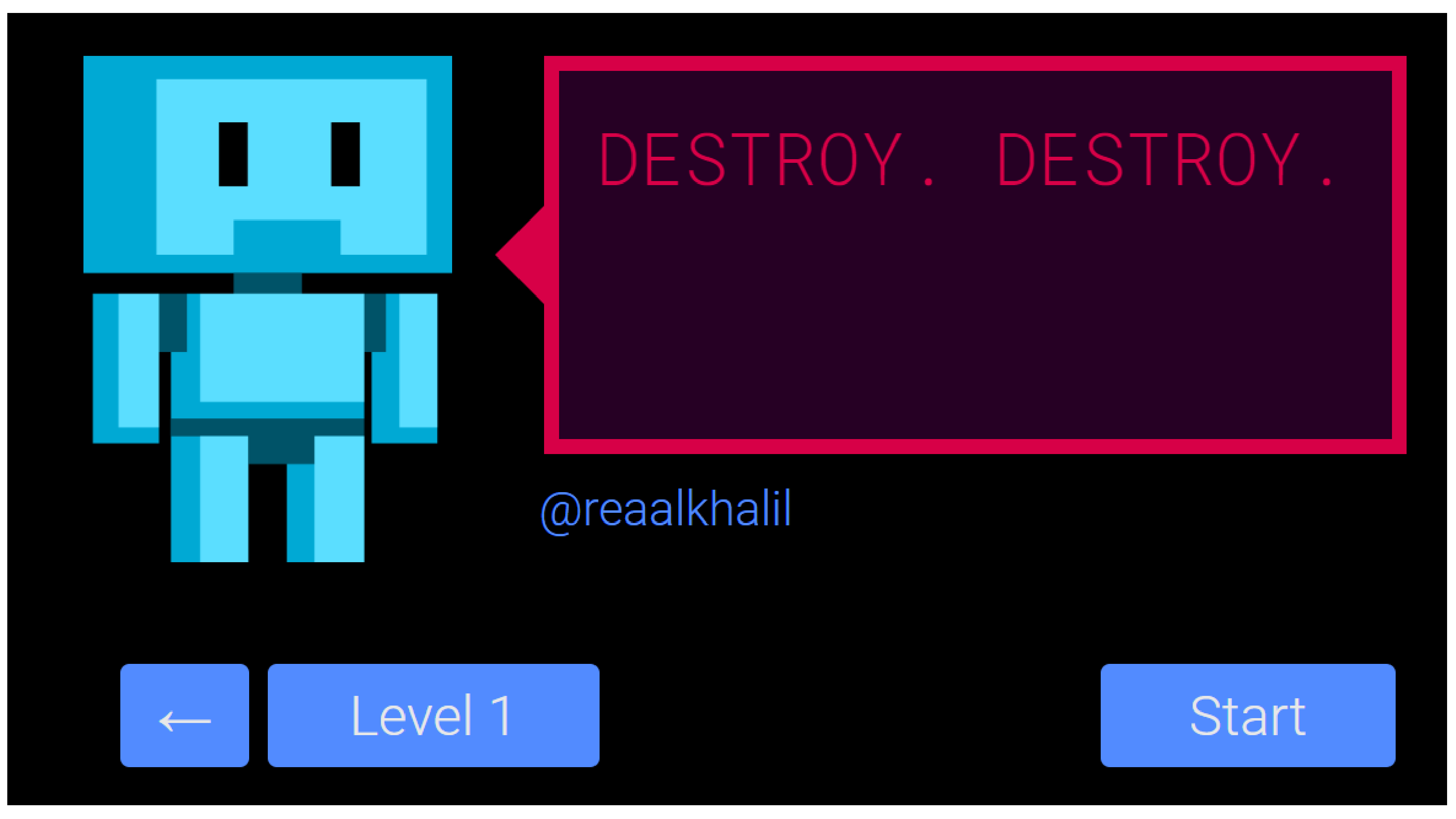
Main JS Operators Cheatsheet

	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	{ key : 1 } function() { ... } for (...) { ... }

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)



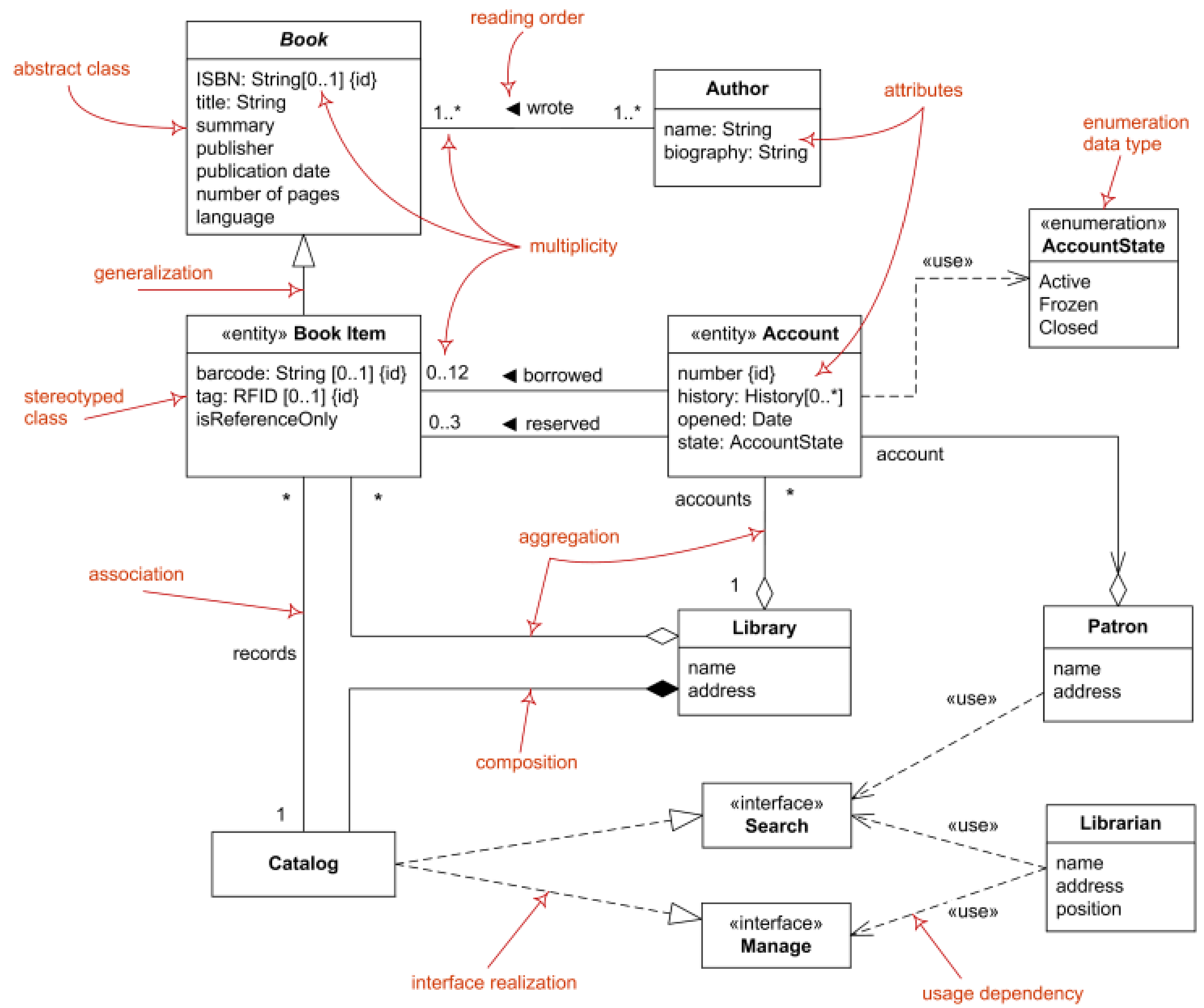
https://lab.reaal.me/jsrobot/

Part 2: Object Oriented Programming (OOP)

Object Oriented Programming (OOP)

- JavaScript is **multi-paradigm**, it has features of the OOP paradigm and of the procedural programming (PP) paradigm
- OOP and PP are two conceptually opposite coding philosophy
- PP revolves *stateless* **procedures** (functions)
- OOP revolves around *stateful* **objects** and **classes**, and on precise relationships between them.

Objects and Classes Diagram



https://www.uml-diagrams.org/class-diagrams-overview.html

Objects and Classes Diagram

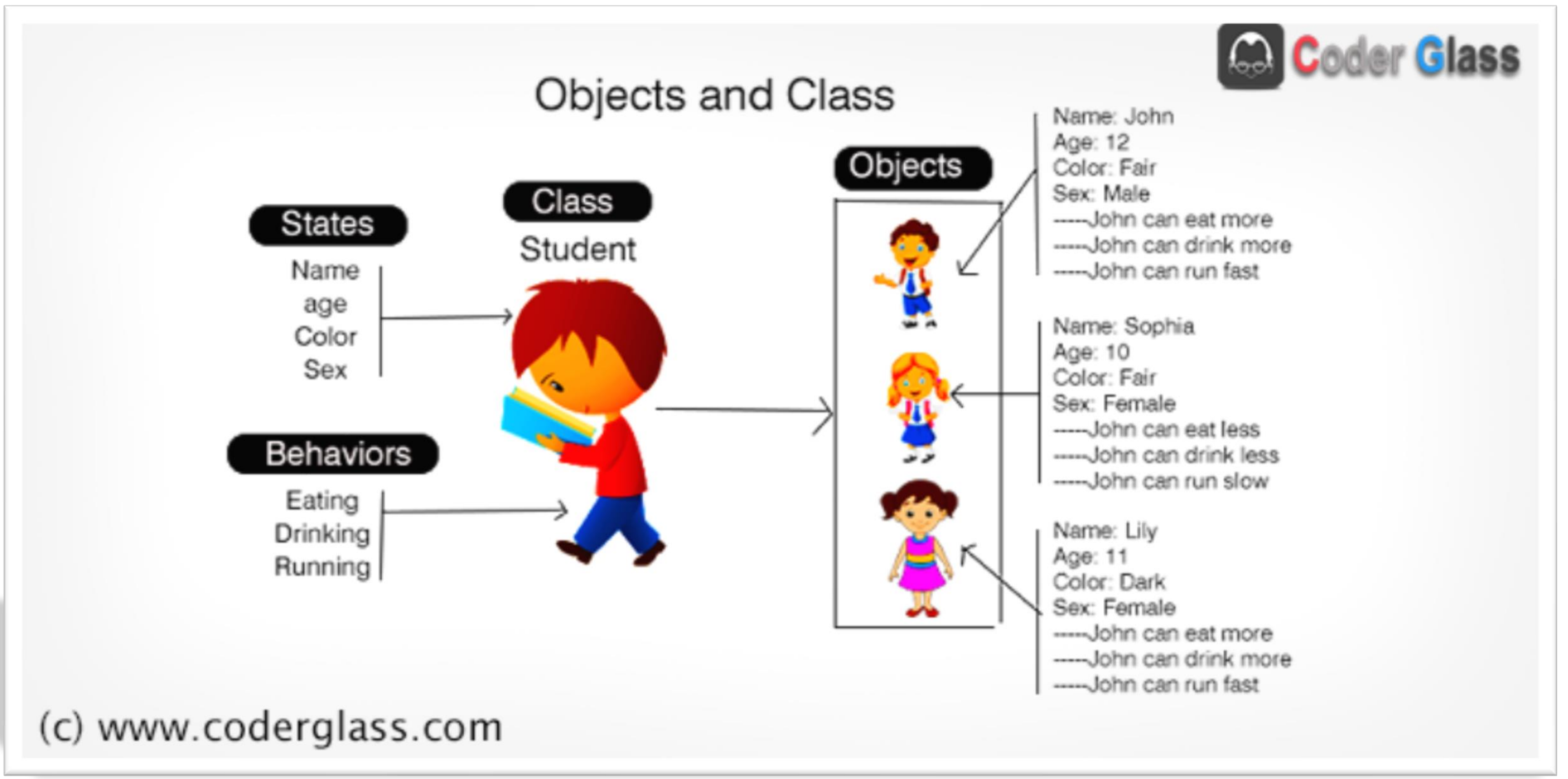
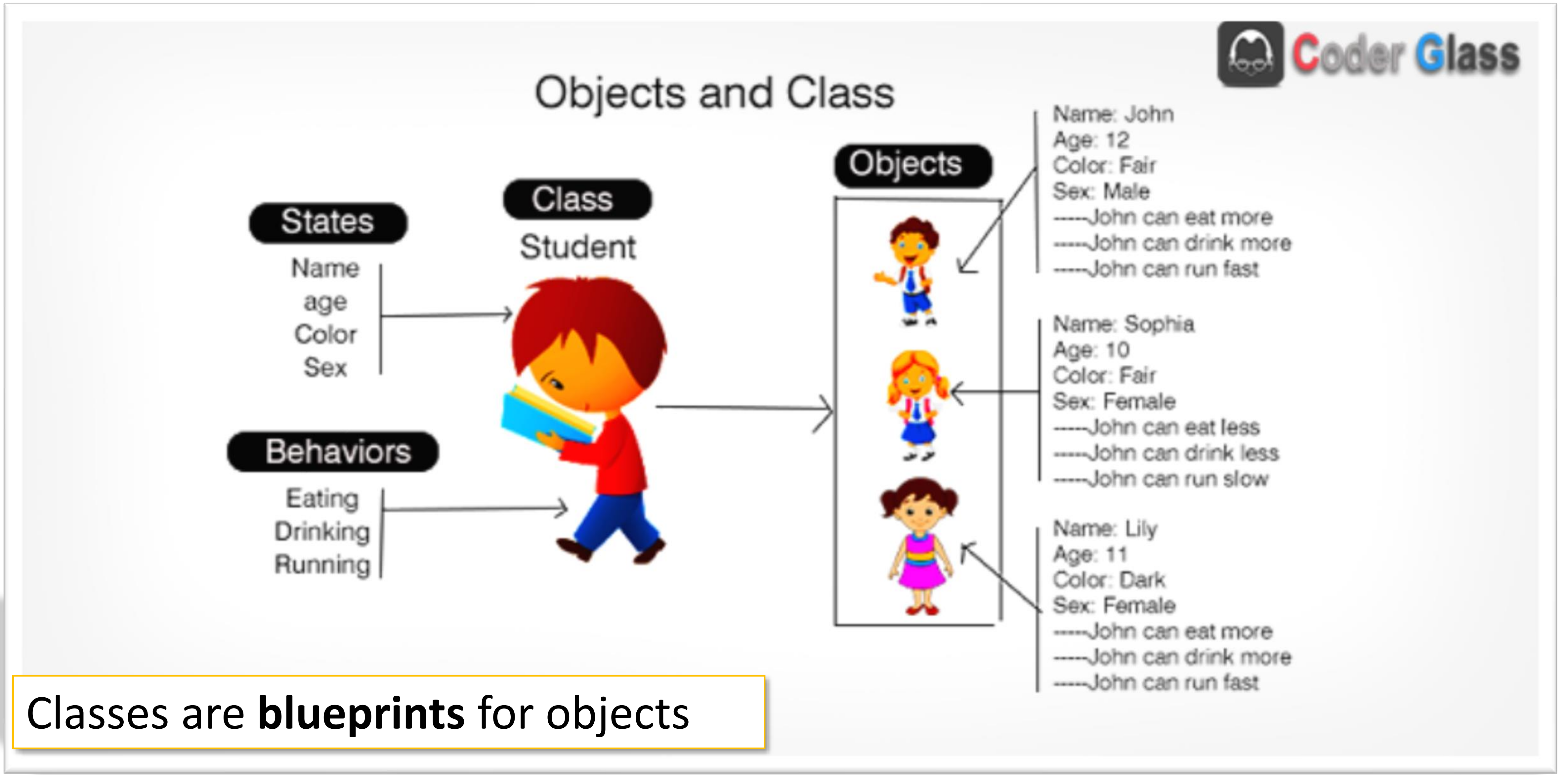


Image source: https://www.coderglass.com/java/java-object-and-class.php

Objects and Classes Diagram



Classes are **blueprints** for objects

Image source: https://www.coderglass.com/java/java-object-and-class.php

https://en.wikipedia.org/w

Objects and Classes Diagram

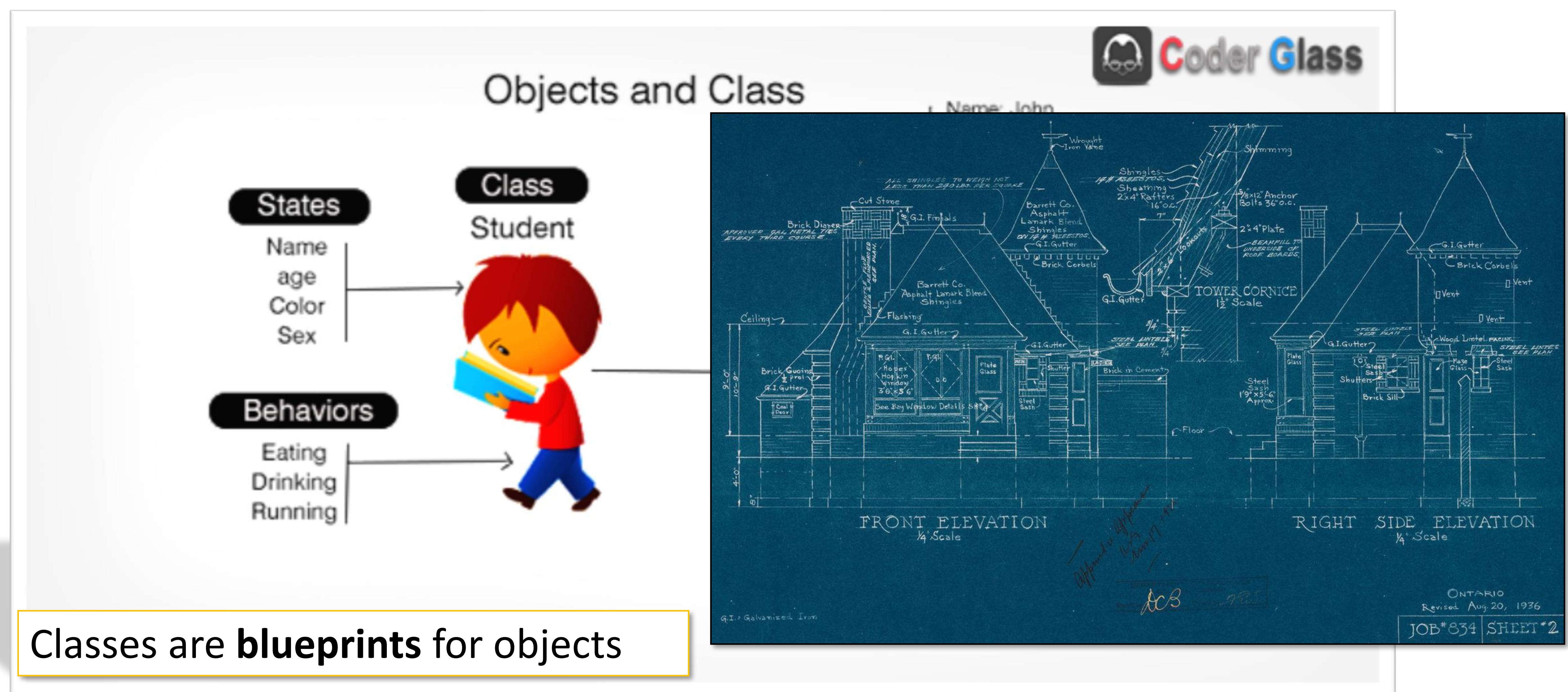
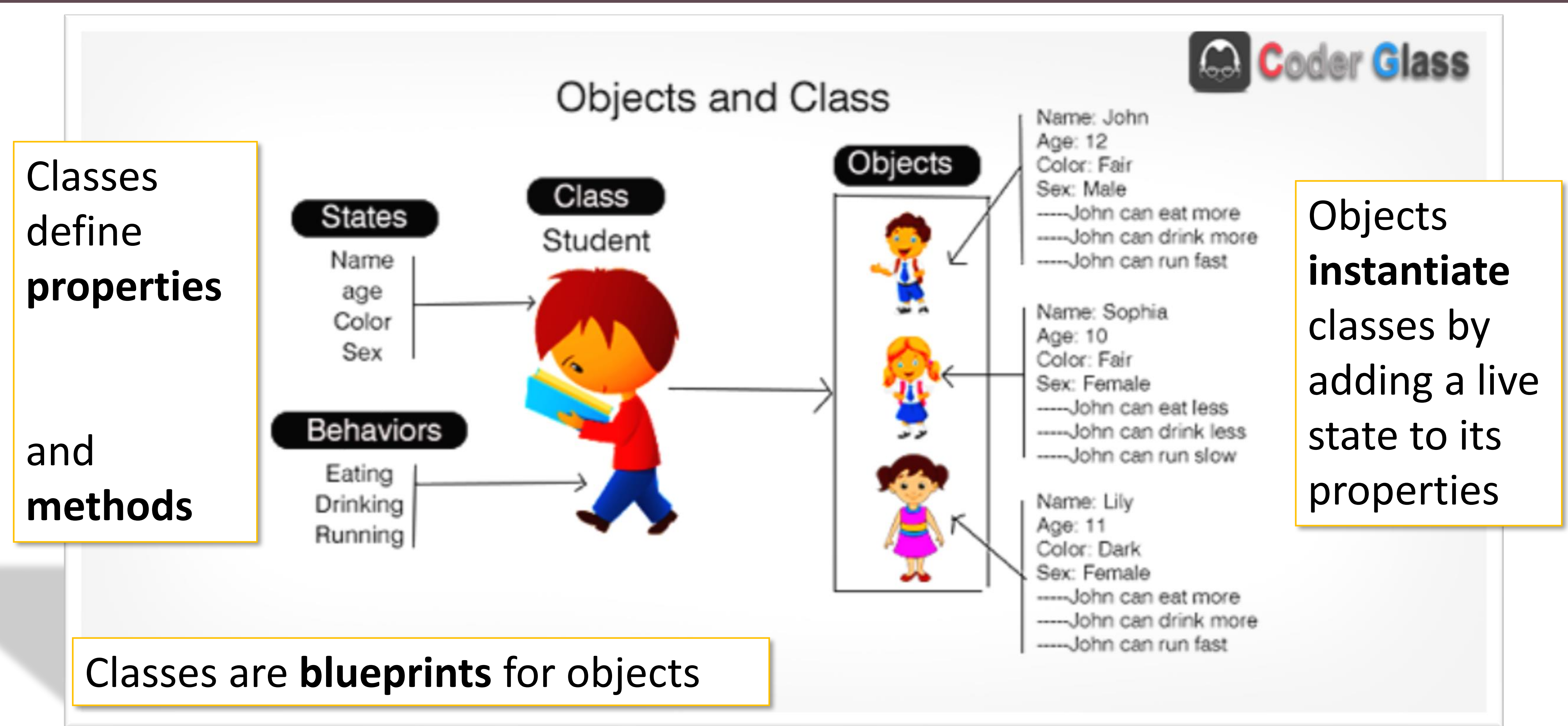


Image source: https://www.coderglass.com/java/java-object-and-class.php

Objects and Classes Diagram



Classes define **properties** and **methods**

Objects **instantiate** classes by adding a live state to its properties

Classes are **blueprints** for objects

Image source: https://www.coderglass.com/java/java-object-and-class.php

JavaScript Classes

```
class Person {

  constructor() {
    this.name = 'Stefano Balietti';
  }

  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}
```

JavaScript Classes

```
class Person {

  constructor() {
    this.name = 'Stefano Balietti';
  }

  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}
```

Notice! This is the news ES6 definition of a class.
It is much easier than using ES5 prototypical definition, even if behind the scenes it is exactly the same. *Exercise available!*

JavaScript Classes

```
class Person {
  constructor() {
    this.name = 'Stefano Balietti';
  }
  sayHi() {
    console.log('Hi! I am ' + this.name);
  }
}

// Create an object using the new operator
let stefano = new Person();
```

JavaScript Classes

```
class Person {
   constructor() {
      this.name = 'Stefano Balietti';
   }
   sayHi() {
      console.log('Hi!
   }
}
// Create an object using the new operator
let stefano = new Person();
```

The new operator invokes the **constructor** method of the class. The constructor is a special method which is executed only once, upon creation.

JavaScript Classes

```
class Person {
  constructor() {
    this.name = 'Stefano Balietti';
  }
  sayHi() {
    console.log('Hi!
  }
}

// Create an object
let stefano = new Pe
```

The new operator invokes the **constructor** method of the class. The constructor is a special method which is executed only once, upon creation.

In this case, it is adding the property 'name' with the value 'Stefano Balietti'.

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

```
constructor() {
  let person = {};
  person.name = 'Stefano Balietti';
  return person;
}
```

The Constructor

```
constructor() {
   this.name = 'Stefano Balietti';
}
```

The constructor is a compact way of creating new objects. What it does is the following:

```
constructor() {
  let this = {};
  this.name = 'Stefano Balietti';
  return this;
}
```

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
   name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
    name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

1. For complex object is faster because the blueprint is already loaded in memory
2. It allows for complex objects!

The Instantiated Object

```
// Create an object using the new operator
let stefano = new Person();
console.log(stefano)
```

```
{
    name: 'Stefano Balietti
}
```

In the technical language the variable stefano is the live "instance" of the class Person.

Couldn't we directly create the object? What is the advantage of using a constructor function?

1. For complex object is faster because the blueprint is already loaded in memory
2. It allows for complex objects! `stefano.sayHi();//I am Stefano Balietti`

A More Complex Person

```
class Person {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name;
      ', and I was born in ' + this.year;
  }
}
```

Here the constructor is accepting input parameters to customize the instance.

A More Complex Person

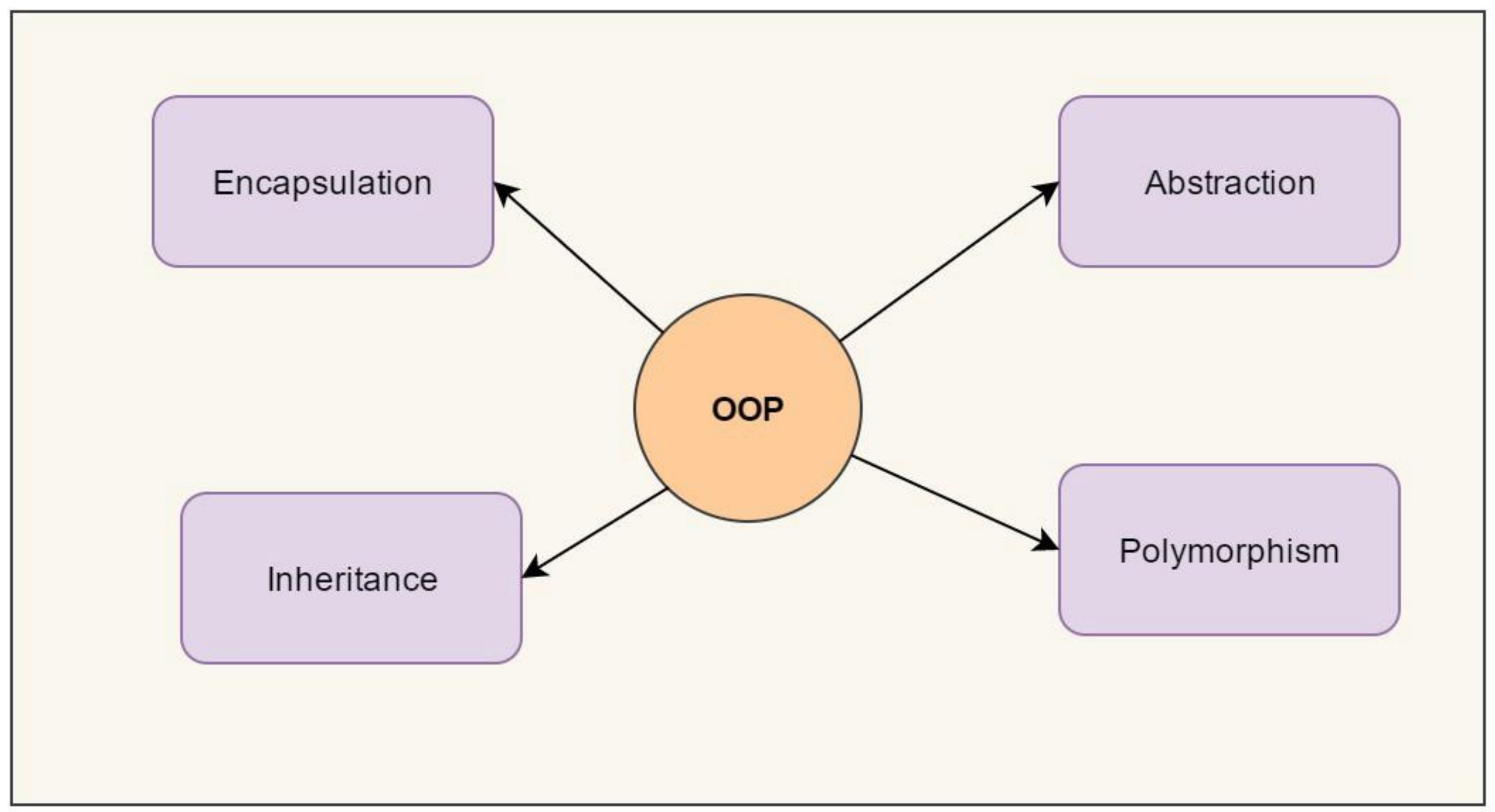
```
class Person {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name;
      ', and I was born in ' + this.year;
  }
}
let brendan = new Person('Brendan', 1961);
brendan.sayHi('Stefano');
// 'Hello Stefano. I am Brendan and I was born in 1961'
```

Here the constructor is accepting input parameters to customize the instance.

Exercises

Part_2_OOP/classes.js

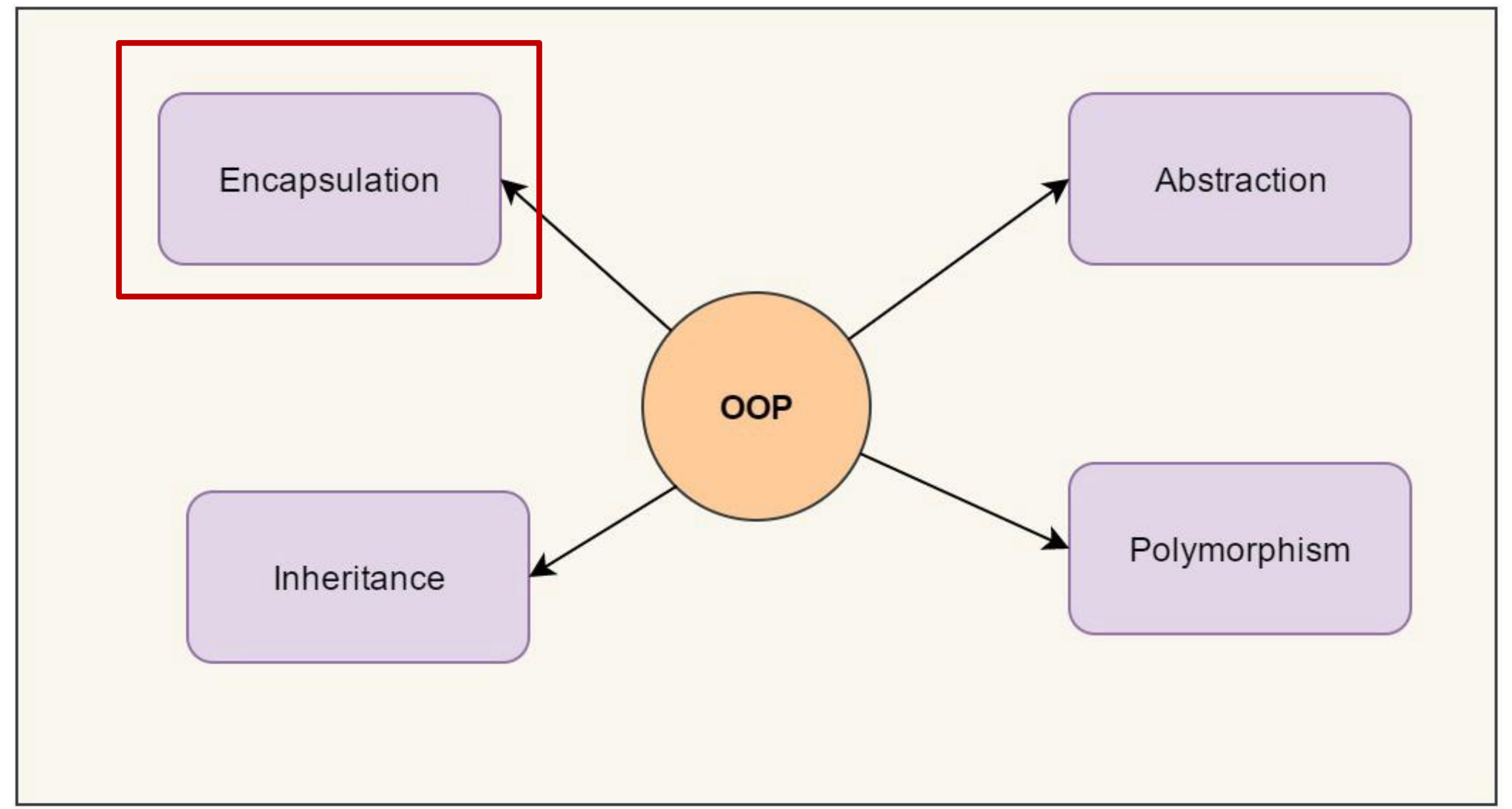
4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Encapsulation

- Encapsulation means that you can hide some of the methods and properties of a class declaring them as **private**, so they are *not* accessible outside of the class
- This prevents erroneous or malicious manipulation of the object by other entities
- It also reduces the complexity of the API for other external developers

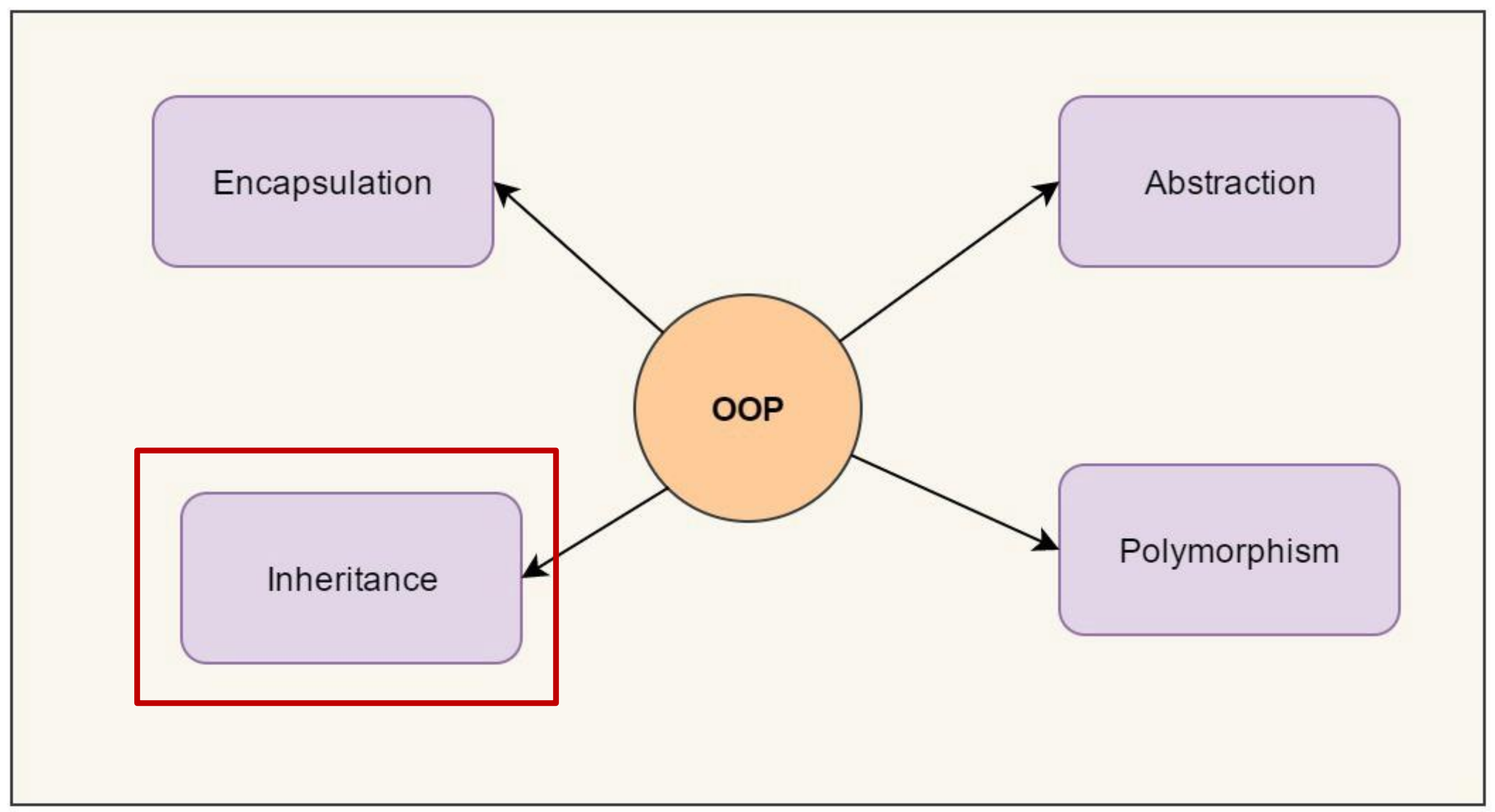
Encapsulation

- Encapsulation means that you can hide some of the methods and properties of a class declaring them as **private**, so they are *not* accessible outside of the class.
- This prevents erroneous or malicious manipulation of the object by other entities
- It also reduces the complexity of the API for other external developers

- *JavaScript does not natively support encapsulation*
- You can do it with **closures**, but it is complex topic, so we don't apply it here

- Here some references for the curious ones:
- https://medium.com/@luke_smaki/javascript-es6-classes-8a34b0a6720a
- https://www.intertech.com/Blog/encapsulation-in-javascript/

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Inheritance

- Inheritance means that classes can share portion of codes with each other, by defining directional relationships of dependence, such as Parent/Child
- *JavaScript has native support for this feature*

OOP Pillar 1: Inheritance

```
class Liar extends Person {

        // We are going to add code here.

}
```

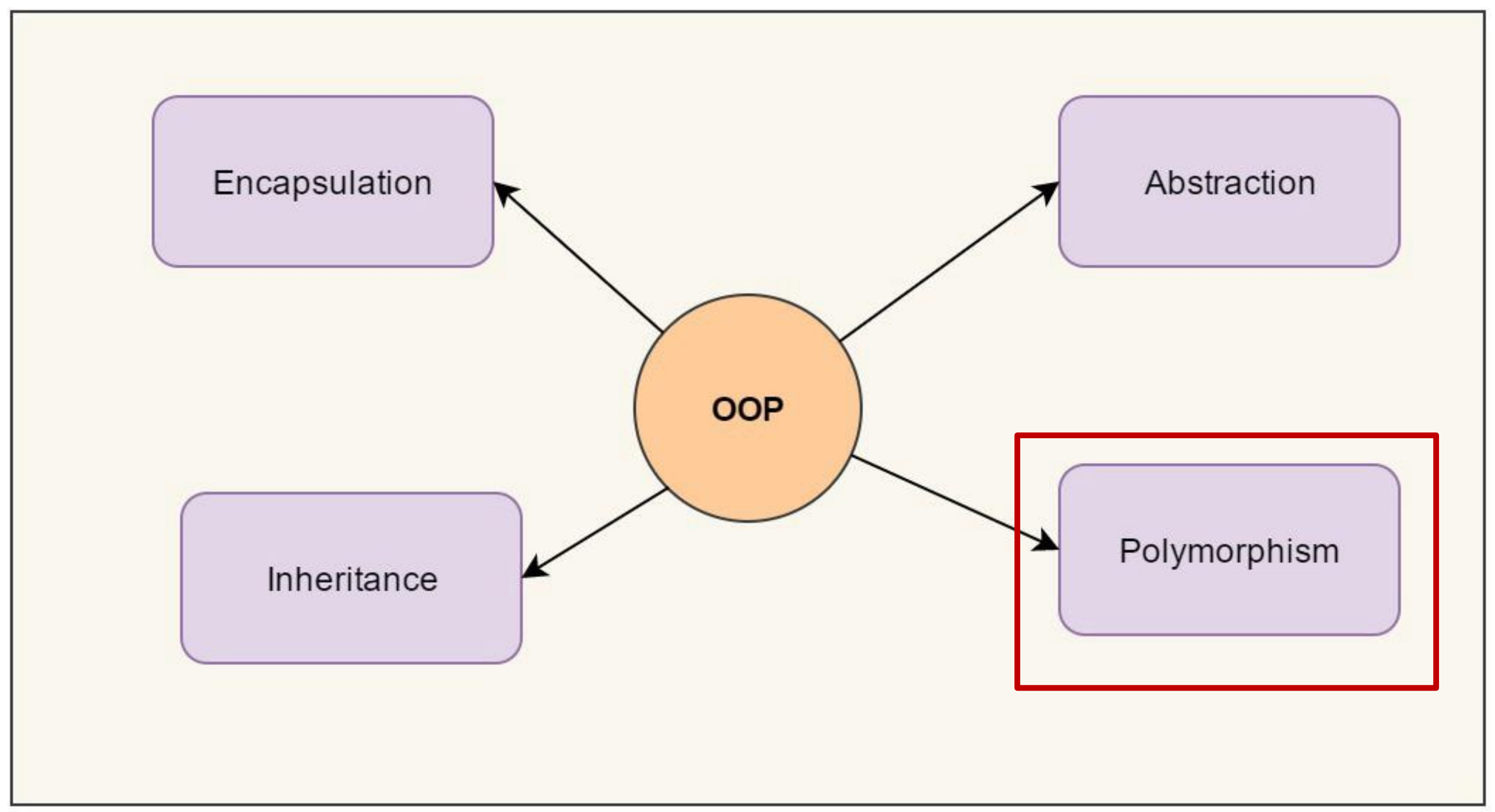
OOP Pillar 1: Inheritance

```
class Liar extends Person {

        // We are going to add code here.

}
```

Here we extend the previously defined Person class.

It means that the Liar class will have all the methods (including the constructor) and properties of the parent class.

4 Pillars of OOP



Four Pillars of Object Oriented Programming

Picture source: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

Polymorphism

- Inheritance means that classes can share portion of codes with each other, by defining directional relationships of dependence, such as Parent/Child
- *JavaScript has native support for this feature*

- You can't really separate polymorphism from inheritance
- It means one get take many forms
- More specifically, the same method can morph into another one

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

This person is faking to be 15 younger than he or she is.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + 15);
  }
}
```

Here we replace ("**override**") the body of the sayHi method with another one.

This person is faking to be 15 younger than he or she is.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }


}
```

15 can become a parameter

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }

}
```

Note for the Nerds! This type of polyphormism is called **"overloading":** the same method is accepting different combination of input parameters.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }

}
```

Note for the Nerds! This type of polymorphism is called **"overloading":** the same method is accepting different combination of input parameters. However, *JavaScript does not support overloading* and the method is technically **overridden**, so that only one method sayHi exists in the end. Other programming languages will generate two methods, distinguishing them by their input parameters.

OOP Pillar 2: Polymorphism

```
class Liar extends Person {

  sayHi(to, degree) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + degree);
  }
}
```

However, it is kind of weird that who is invoking the sayHi method gets to decide the degree of lying. It should rather be a fixed property of the person.
What is another approach?

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
```

Here we create a new constructor with three input parameters

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
let liar = new Liar('Rosie Ruiz', 1953, 5);
liar.sayHi('Stefano'); // Hello Stefano. I am Rosie Ruiz and I was born in 1953
```

Here we create a new constructor with three input parameters

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    this.name = name;
    this.year = year;
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
}
let liar = new Liar('Rosie Ruiz', 1953, 5);
liar.sayHi('Stefano'); // Hello Stefano. I am Rosie Ruiz and I was born in 1953
```

Here we create a new constructor with three input parameters

Can we do better?

More Polymorphism

```
class Liar extends Person {
   constructor(name, year, degree) {
     super(name, year);
     this.degree = degree;
   }
   sayHi(to) {
       return 'Hello ' + to + '. I am ' + this.name +
       ', and I was born in ' + (this.year + this.degree);
   }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

```
constructor(name, year) {
  this.name = name;
  this.year = year;
}
```

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

```
constructor(name, year) {
  this.name = name;
  this.year = year;
}
```

It's just two lines saved, what is the big advantage here?

More Polymorphism

```
class Liar extends Person {
  constructor(name, year, degree) {
    super(name, year);
    this.degree = degree;
  }
  sayHi(to) {
      return 'Hello ' + to + '. I am
      ', and I was born in ' + (this
  }
}
```

super means the super class, that is, the *parent* class. Here we are invoking its constructor.

```
constructor(name, year) {
  this.name = name;
  this.year = year;
}
```

It's just two lines saved, what is the big advantage here?

We avoid code duplication, this makes maintaining the code much easier.
Some constructors can set up many variables at the same time, even methods.

Exercises

Part_2_OOP/encapsulation.js
Part_2_OOP/inheritance_and_poly.js

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can extend extending classes.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
   sayHi(to) {
       if (Math.random() > 0.5) return 'Who am I?';
       else return super.sayHi(to);
   }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

We just 6 lines of code, we created a relatively complex personality thanks to inheritance and polymorphism: *a confused liar*! Isn't that amazing?

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

We can use *super* to access any method of the parent class.
Here, the confused liar with probably 0.5 will not remember who he or she is (or is it just faking?), otherwise he or she will lie as before.

We just 6 lines of code, we created a relatively complex personality thanks to inheritance and polymorphism: *a confused liar*! Isn't that amazing?

How can we do better?

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      else return super.sayHi(to);
  }
}
```

The else word is *not* needed here.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      if (Math.random() > 0.5) return 'Who am I?';
      return super.sayHi(to);
  }
}
```

Two return statements are not needed either.

More Polymorphism and Inheritance

```
class ConfusedLiar extends Liar {
  sayHi(to) {
      return Math.random() > 0.5 ? 'Who am I?' : super.sayHi(to);
  }
}
```

With the ternary operator we saved one extra line without losing readability.
5 lines! Amazing!

Advanced Topic: Context

- The value of this is called **context**

```
sayHi(to) {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
  }
```

Advanced Topic: Context

- The value of this is called **context**

```
sayHi(to) {
	return 'Hello ' + to + '. I am ' + this.name +
	', and I was born in ' + (this.year + this.degree);
}
```

- In JavaScript, surprisingly, it is not fixed, but it changes dynamically depending on where the function is executed

Advanced Topic: Context

- The setTimeout function lets you execute some code after a given amount of time (here 2 seconds).

```
setTimeout(function() {
  // Code to be added

}, 2000);
```

Advanced Topic: Context

- If you use the setTimeout function inside our sayHi method the result might be disappointing.

```
setTimeout(function() {
   // Code to be added

}, 2000);
```

- The context, i.e., the value of this, inside the setTimeout function is the setTimeout function itself.
- This is generally terribly confusing to JS beginners

Advanced Topic: Context

```
sayHi(to) {
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + this.name +
      ', and I was born in ' + (this.year + this.degree);
    }, 2000);
}

sayHi('Stefano');
// Hello Stefano. I am undefined, and I was born in undefined.
```

Advanced Topic: Context

- You can circumvent this problem, by storing the value of this inside another variable.
- For historical reason, it is customary to call this variable *that*

```
sayHi(to) {
    let that = this;
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + that.name +
      ', and I was born in ' + (that.year + that.degree);
    }, 2000);
}
```

Advanced Topic: Context

- You can circumvent this problem, by storing the value of this inside another variable.
- For historical reason, it is customary to call this variable *that*

```
sayHi(to) {
    let that = this;
    setTimeout(function() {
      return 'Hello ' + to + '. I am ' + that.name +
      ', and I was born in ' + (that.year + that.degree);
    }, 2000);
}
```

- Alternatively, you can use an *arrow function* as a parameter of the setTimeout function

Advanced Topic: Arrow Functions

- Introduced in ES6
- They look weird
- They can shorten function definitions

```
// Standard way.
function() {
   return 'I am a normal function';
}

// Arrow functions.
() => {
   return 'I am an arrow function';
}
```

It isn't much shorter though…

Advanced Topic: Arrow Functions

- Introduced in ES6
- They look weird
- They can shorten function definitions

```
// Standard way.
function() {
   return 'I am a normal function';
}

// Arrow functions.
() => {
   return 'I am an arrow function';
}
```

It isn't much shorter though…There are conditions in which parentheses can be omitted.

Exercises

Part_2_OOP/4_this.js

Objected Oriented Cooperation Tournament

Part_2_OOP/5_final_exercise.js

But first the theory!