Javascript Crash Course

November 2018

Programming

Defining a structured data set and different types of deterministic operations (e.g. logical, arithmetic, conditional, repetitive) being executed on it in particular order.

Just like writing a cookbook.

Programming

tl;dr:

data + operations

Javascript

A widely used, single-threaded scripting programming language.

Although it can be used for multiple purposes, the major use case for it is web domain. ~99% modern web platforms have been written with the usage of Javascript.

Interactive webpage = Javascript

Representing the data

Primitive types	Reference types
- usually carry context-less information - “small” piece of data - each function call with argument being a primitive results in copying the input	- often linked with other data (a meaningful bag of primitives) - represents complex objects - only the reference (memory address) is passed during the function call

Declaring the data

There are two main ways to declare the variables (data) in Javascript:

`const <variable_name> = <variable_value>`	`let <variable_name> [= <variable_value>]`
Means that the value (the reference) to this variable cannot change over time; must be initialized during the declaration	The value can change often; may/may not be initialized when declaring

Declaring the data

	`const <variable_name> = <variable_value>`	`let <variable_name> [= <variable_value>]`
1	`const name = “Janusz”`	`let age`
2	`name = “Grażyna” // TypeError: Assignment to constant variable.`	`age = 26`
3		`age = 27`
4	`console.log(name) // “Janusz”`	`console.log(age) // 27`

Variables scopes

Both const and let are block-scoped.

It means that they are valid in the block (within the curly braces) they were declared.

Variables scopes

```
1  const fruit = “apple”
2  console.log(fruit) // “apple”
3
4  {
5    const fruit = “orange”
6    console.log(fruit) // “orange”
7  }
8
9  console.log(fruit) // “apple”
```

Data types

Each variable can represent different type of data. Javascript is dynamically typed - it means than in opposite to other common programming languages (like C, C#, Java and more), type annotation is not a part of the variable declaration. One variable can represent various types of data in different phases of the application lifecycle.

Data types

```
1  let data // variable “data” represents undefined
2  data = 199.99 // now “data” is a number

3  let otherData = data + 2 // “otherData” is also a number
4  console.log(otherData) // 201.99

5  data = “JS” // now “data” is a string
6  otherData = data + “2018” // now “otherData” is also a string
7  console.log(otherData) // “JS2018”
```

boolean

/ primitive type /

Represents only the logical value. Not more.

```
1  const areWeInGdynia = true
2  const isItSunnyToday = false
```

Handy while executing logical and conditional operations on the data.

number

/ primitive type /

Because digits are everywhere.

```
1  const milkPrice = 3.99
2  const breadPrice = 1.99
3  const totalPrice = milkPrice + breadPrice // 5.98
```

Used for arithmetic operations (obviously).

string

/ primitive type /

Characters. Words. Sentences. Prose.

```
1  const firstName = “Karol”
2  console.log(‘You look handsome today, ‘ + firstName)
```

“Words, once they are printed, have a life of their own”

~Carol Burnett

null

/ primitive type /

Expresses an emptiness.

```
1  const myData = null
2  console.log(myData) // null
```

Represents a variable with no meaningful value.

undefined

/ primitive type /

When you (un)intentionally forget to assign a value to a declared variable.

```
1  let personalId
2  console.log(personalId) // undefined
```

Object

/ reference type /

Data set containing contextual informations.

```
1  const person = {
2    lastName: “Doe”,
3    age: 61
4  }
5
6  console.log(person) // { lastName: “Doe”, age: “61” }
7
8  person.firstName = “John”
9  console.log(person) // { firstName: “John”, lastName: “Doe”, age:
   “61” }
```

Represented as a set of key-value pairs. May be dynamically extended.

Object

/ reference type /

```
1   const myDog = {
2     age: 10,
3     legsNumber: 4
4   }
5
6   myDog.name = “Reksio”
7   myDog.name = “Azor”
8
9   myDog = {
10    age: 1,
11    legsNumber: 4
12  } // TypeError: Assignment to
    constant variable.
```

```
1   let myDog = {
2     age: 10,
3     legsNumber: 4
4   }
5
6   myDog.name = “Reksio”
7   myDog.name = “Azor”
8
9   myDog = {
10    age: 1,
11    legsNumber: 4
12  }
```

Arrays

/ reference type - also objects /

A finite bag for data.

```
1   const scores = [ 19.5, 19, 19.5, 18.5, 18 ]
2
3   console.log(scores.length) // 5
4   console.log(scores[3]) // 18.5
5
6   scores[3] = 16.5
7   console.log(scores[3]) // 16.5
8
9   scores.push(“twenty”)
10  console.log(scores) // [ 19.5, 19, 19.5, 16.5, 18, “twenty” ]
```

The list can contain mixed primitives and objects of different types.

Date

/ reference type - also objects /

Guess what’s its purpose :)

```
1  const now = new Date()
2
3  const hour = now.getHours()
4  const month = now.getMonth()
5  // etc.
```

The Date objects contain information about the timezone too.

Arithmetic operations

Used to perform arithmetic operations on numbers

```
1   let x = 5
2
3   console.log(x + 3) // addition
4   console.log(x - 3) // subtraction
5
6   console.log(x * 2) // multiplication
7   console.log(x / 5) // division
8
9   console.log(0.2 + 0.3) // 0.5
10  console.log(0.1 + 0.2) // ???
```

Arithmetic operations

Incrementation and decrementation - operators that add or subtract one from the variable.

```
1   let y = 5
2
3   console.log(y++) // 5
4   console.log(y) // 6
5   console.log(++y) // 7
6
7   y = 5
8   console.log(y--) // 5
9   console.log(y) // 4
10  console.log(--y) // 3
```

Arithmetic operations

Modulo - remainder after division of one number by another.

```
1  let z = 5
2
3  console.log(z % 1) // 0
4  console.log(z % 2) // 1
5  console.log(z % 3) // 2
```

String operators

```
1  let str1 = “Hello”
2  let str2 = “World”
3
4  console.log(str1 + str2) // “HelloWorld”
5  console.log(str1 + “ “ + str2) // “Hello World”
6  console.log(str1, str2) // “Hello World”
7  console.log(1 + “Hello”) // “1Hello”
```

String literals

String can contain a variable with usage of string templates.

```
1  const str1 = “Hello”
2  const str2 = “World”
3
4  const helloWorld = `${str1} ${str2}`
5
6  console.log(helloWorld) // “Hello World”
```

Expressions are indicated by dollar sign and curly braces, e.g. ${expression}

String templates are created using back-tick (``).

Assignment

Assign value to variable

```
let x = 5

x += 1 // 6
x -= 2 // 4

x *= 3 // 12
x /= 4 // 3

x %= 5 // 3
```

```


x = x + 1
x = x - 2

x = x * 3
x = x / 4

x = x % 5
```

Comparison

=== , == and type coercion

```
1  console.log(5 === 5) // true
2  console.log(5 == 5) // true
3
4  console.log(21 == ‘21’) // true
5  console.log(21 === ‘21’) // false
```

=== checks value and types

== performs type coercion - convert values into a common type

In most cases we use triple equals.

Comparison

To check that things are not equal.

```
1  console.log(5 !== 6) // true
2  console.log(5 != 6) // true
3
4  console.log(21 != ‘21’) // false
5  console.log(21 !== ‘21’) // true
6
7  console.log(“Hello” != “World”) // true
8  console.log(“Hello” !== “World”) // true
```

Comparison

Greater than, less than

```
1  console.log(5 > 6) // false
2  console.log(5 >= 5) // true
3
4  console.log(5 < 6) // true
5  console.log(5 <= 6) // true
```

Conditions

Conditional expression checks the condition and take an action depending on the result.

```
1  if (5 > 6) {
2    console.log(“5 is greater than 6”)
3  } else if (5 === 6) {
4    console.log(“5 is equal 6”)
5  } else {
6    console.log(“5 is lower than 6”)
7  }
```

else if and else are optional.

Truthy and falsy values

The following values are defined as falsy:

- false
- 0
- “” (empty string)
- null
- undefined
- NaN (not a number)

Everything else is truthy.

Logical operators

And, or, not

```
1   const num1 = 5
2   const num2 = -2
3
4   if (num1 === 5 && num2 === 5) {
5     console.log(“Both variables are equal 5”)
6   } else if (num1 === 5 || num2 === 5) {
7     console.log(“At least one value is equal 5”)
8   } else if (!(num1 === 5 || num2 === 5)) { // the same as num1 !== 5 && num2 !== 5
9     console.log(“Both values are different than 5”)
10  }
```

Ternary operator

Assign value to variable depending on some condition.

```
1  const person = {
2    age: 16
3  }
4
5  const majority = person.age >= 18 ? “adult” : “underage”
6  console.log(majority) // “underage”
```

Switch operator

Perform different actions based on different conditions.

```
1   const num = 1
2
3   switch (num) {
4     case 1:
5       console.log(“one”)
6       break
7     case 2:
8       console.log(“two”)
9       break
10    default:
11      console.log(“unrecognized number”)
12  }
```

Functions

Function is an ordered set of instructions that will be executed during single call. Defining functions helps programmers to avoid repeating themselves. Functions can have names, so with good naming it is crystal clear what is their main purpose.

The operations defined without function body can be executed on an input data (function's arguments). The final effect (output) of the function call can be returned to the caller by a return statement.

Functions examples

```
1   // function definition
2   function greet() { // function definition
3     const now = new Date()
4     if (now.getHours() > 18) {
5       console.log(“Good evening”)
6     } else {
7       console.log(“Good morning”)
8     }
9   }
10
11  // function call
12  greet()
```

Functions examples

```
1   // function definition
2   function greetPerson(name) {
3     const now = new Date()
4     if (now.getHours() > 18) {
5       console.log(`Good evening, {name}`)
6     } else {
7       console.log(`Good morning, {name}`)
8     }
9   }
10
11  // function call
12  greetPerson(“Nicole”)
```

Functions examples

```
1   // function definition
2   function getTotalPrice(quantity, price) {
3     return quantity * price
4   }
5
6   // function calls
7   getTotalPrice(2, 2.50)
8
9   const onionPricePerKg = 0.42
10  const onionsKg = 11
11  const allOnionsValue = getTotalPrice(onionsKg, onionPricePerKg)
12
13  console.log(allOnionsValue) // 4.62
```

Functions examples

```
1   // function definition
2   function getBirthYear(person) {
3     const now = new Date()
4     const currentYear = now.getFullYear()
5
6     return currentYear - person.age
7   }
8
9   // function calls
10  const mrGoalScorer = { lastName: “Lewandowski”, age: 30 }
11  console.log(getBirthYear(mrGoalScorer)) // 1988
12
13  const mrUnknown = { lastName: “Doe” }
14  console.log(getBirthYear(mrUnknown)) // NaN
```

Functions examples

```
1   // function definition
2   const checkAge = function(age) {
3     return age >= 18
4       ? “Access Granted”
5       : “Access Denied”
6   }
7
8   const personDetails = {
9     firstName: “Anna”,
10    age: 14
11  }
12
13  // function call
14  const ageCheckOutput = checkAge(personDetails.age)
15  console.log(ageCheckOutput) // “Access Denied”
```

Functions examples

```
1   // function definition
2   const checkAge = (age) => age >= 18 ? “Access Granted” : “Access Denied”
3
4   const personDetails = {
5     firstName: “Marek”,
6     age: 22
7   }
8
9   // function call
10  const ageCheckOutput = checkAge(personDetails.age)
11  console.log(ageCheckOutput) // “Access Granted”
```

Loops

Allow programmers to execute the same set of operations (instructions or function calls) multiple times (until the certain condition is met).

Very useful when you have to process a collection of entities of the same type in the same manner (using the same algorithm).

For loop

```
1  const numbersArray = [1, 2, 3, 4, 5, 6, 7]
2
3  for(let i = 0; i < numbersArray.length; i++) {
4    console.log(numbersArray[i])
5  }
```

For-of

Used to iterate over all elements of an array.

```
1  const numbersArray = [1, 2, 3, 4, 5, 6, 7]
2
3  // for-of
4  for(number of numbersArray) {
5    console.log(number)
6  }
```

While

Used to repeat some action till the condition is truthy.

```
1  const num = 0;
2
3  while(num < 10) {
4    console.log(num)
5    num++
6  }
```

Do while

Similar to while loop, action is executed at least one time (even if the condition is not fulfilled).

```
1  const i = 0
2
3  do {
4    ++i
5    console.log(i)
6  } while (i < 10)
```

Putting some life into static HTML

Web applications that provides various functionalities (e.g. registering new users, logging in, adding to cart, and millions more) usually require users' interactions.

Handling them in raw HTML and CSS is impossible - both technologies are meant to develop static content.

Putting some life into static HTML

Complex action execution can be achieved by interacting with DOM elements - using mouse, keyboard, touchscreen and other controllers. Every action executed by these controllers on the DOM elements (clicking, hovering over, typing, moving the cursor) can be considered as an interaction.

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const paragraphs =
      document.querySelectorAll(“p”)
2
3   console.log(paragraphs.length) // 3
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const firstParagraph =
      document.querySelector(“p”)
2
3   console.log(paragraph)
      // <p class=”text”>1</p>
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const mainDiv =
      document.querySelector(“#main”)
2
3   console.log(mainDiv)
      // <div id=”main”>...</div>
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const mainDiv =
      document.getElementById(“main”)
2
3   console.log(mainDiv)
      // <div id=”main”>...</div>
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const textPs =
      document.querySelectorAll(“.text”)
2
3   console.log(textPs.length) // 2
4   console.log(textPs[0])
      // <p class=”text”>1</p>
5   console.log(textPs[1])
      // <p class=”text”>2</p>
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const textPs =
      document.getElementsByClassName(“text”)
2
3   console.log(textPs.length) // 2
4   console.log(textPs[0])
      // <p class=”text”>1</p>
5   console.log(textPs[1])
      // <p class=”text”>2</p>
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const mainDiv =
      document.querySelector(“#main”)
2   const divChildren =
      mainDiv.children
3
4   console.log(divChildren.length) // 3
```

Accessing DOM elements

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const firstParagraph =
      document.querySelector(“p”)
2   const content =
      firstParagraph.innerText
3
4   console.log(content) // 1
```

console.dir()

In comparison to console.log(), console.dir() prints the HTML element represented as a fully qualified Javascript object (instead of raw DOM element).

All properties of the selected element can be then inspected.

Modifying DOM tree programmatically

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   const firstParagraph =
      document.querySelector(“p”)
2   const mainDiv =
      document.querySelector(“#main”)
3
4   mainDiv.removeChild(firstParagraph)
5
```

Modifying DOM tree programmatically

HTML

```
1  <html>
2    ...
3    <body>
4      <div id=”main”>
5        <p class=”text”>2</p>
6        <p>another paragraph</p>
7      </div>
8    </body>
9  </html>
```

Javascript

```
1  const firstParagraph =
     document.querySelector(“p”)
2  const mainDiv =
     document.querySelector(“#main”)
3
4  mainDiv.removeChild(firstParagraph)
5
```

Modifying DOM tree programmatically

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8       </div>
9     </body>
10  <html>
```

Javascript

```
1   const mainDiv =
      document.querySelector(“#main”)
2   const newParagraph =
      document.createElement(“p”)
3   p.innerText = “4th paragraph”
4
5   mainDiv.appendChild(newParagraph)
6
```

Modifying DOM tree programmatically

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p class=”text”>1</p>
6         <p class=”text”>2</p>
7         <p>another paragraph</p>
8         <p>4th paragraph</p>
9       </div>
10    </body>
11  </html>
```

Javascript

```
1   const mainDiv =
      document.querySelector(“#main”)
2   const newParagraph =
      document.createElement(“p”)
3   p.innerText = “4th paragraph”
4
5   mainDiv.appendChild(newParagraph)
6
```

Event listeners

As a result for user's interaction (event) with various HTML elements, we may want to execute some particular instructions. Registering these reactions happens via event listeners.

Event listeners

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p id=”greeting”>
6           Click me!
7         </p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   function greet() {
2     const name = prompt(“What’s your name?”)
3     alert(`Hello ${name}!`)
4   }
5
6   const greetP =
      document.querySelector(“#greeting”)
7   greetP.addEventListener(“click”, greet)
```

Event listeners

HTML

```
1   <html>
2     ...
3     <body>
4       <div id=”main”>
5         <p id=”adder”>
6           Focus on me!
7         </p>
8       </div>
9     </body>
10  </html>
```

Javascript

```
1   let psCounter = 0
2
3   function add() {
4     const mainDiv =
        document.querySelector(“#main”)
5     const newP = document.createElement(“p”)
6     newP.innerText = `Paragraph ${++psCounter}`
7     mainDiv.appendChild(newP)
8   }
9
10  const addP =
      document.querySelector(“#adder”)
11  addP.addEventListener(“mouseover”, add)
```

Event types

https://developer.mozilla.org/pl/docs/Web/Events

setTimeout()

Sometimes we want to defer some action over time. Such an action (a reference to a function) should be then passed as a parameter to setTimeout() function call, alongside with the desired delay (in ms).

setTimeout() example

```
1  function delayedGreeting() {
2    console.log(“Hello!”)
3  }
4
5  setTimeout(delayedGreeting, 5000)
```

clearTimeout()

Under some conditions we may want to cancel the execution of the deferred function. In these cases we should use clearTimeout() function.

Since we may have a few timers running in our application, we should explicitly pass the correct timer ID as a parameter of clearTimeout(). The IDs are returned from each setTimeout() calls.

clearTimeout() example

```
1  function delayedGreeting() {
2    console.log(“Hello!”)
3  }
4
5  const greetTimerId = setTimeout(delayedGreeting, 5000)
6
7  console.log(“Goodbye!”)
8  clearTimeout(greetTimerId)
```

setInterval() / clearInterval()

Some operations may need to be executed periodically in equal time intervals (e.g. counting). For that purpose, setInterval() function comes in handy.

Similarly to setTimeout(), setInterval() accepts two arguments - reference to the action to be executed periodically and a time interval (in ms). Clearing the interval is also done analogically.

setInterval() / clearInterval() example

```
1   let counter = 0
2   let intervalId
3
4   function count() {
5     console.log(++counter)
6     if (counter === 10) {
7       clearInterval(intervalId)
8     }
9   }
10
11  intervalId = setInterval(count, 1000)
```

Embedding JS inside HTML

Just like CSS styles definitions, JS scripts can be directly embedded into HTML markup.

In order to inject Javascript code (or reference to the separate files with our scripts), we should utilize <script> tag.

<script> tag should be placed either in <head> or just before </body> closing tag.

Embedding JS inside HTML

```
1   <html>
2     <head>
3       <script>
4         console.log(“Hello from head!”)
5       </script>
6     </head>
7     <body>
8       <h1>Hi!</h1>
9     </body>
10  </html>
```

Embedding JS inside HTML

```
1   <html>
2     <head>
3       ...
4     </head>
5     <body>
6       <h1>Hi!</h1>
7       <script>
8         console.log(“Hello from body!”)
9       </script>
10    </body>
11  </html>
```

Embedding external JS

A good practice is to separate view layer (HTML) from logic implementation (JS) - in other words: to put them in separate files.

JS code from separate file (usually with .js extension) can be injected into our webpage with src attribute of <script> tag.

Embedding external JS

HTML

```
1   <html>
2     <head>
3       ...
4     </head>
5     <body>
6       <h1>Hi!</h1>
7       <script src=”./main.js”></script>
8     </body>
9   </html>
10
```

Javascript (main.js)

```
1   function changeBackground() {
2     const header =
        document.querySelector(“h1”)
3     header.style.backgroundColor = “red”
4   }
5
6   changeBackground()
```

Mixing concerns (not recommended)

WARNING: Bad practice.

Event listeners can be also attached directly from the HTML markup. That approach is not recommended nowadays - we should avoid mixing view layer with business logic and follow the separation of concerns principle.

Mixing concerns (not recommended)

HTML

```
1   <html>
2     <head>
3       <script src=”./events.js”></script>
4     </head>
5     <body>
6       <h1>Hi!</h1>
7       <button onclick=”changeBackground()”>
8         Click me!
9       </button>
10    </body>
11  </html>
```

Javascript (events.js)

```
1   function changeBackground() {
2     const header =
        document.querySelector(“h1”)
3     header.style.backgroundColor = “red”
4   }
```

Separation of concerns

HTML

```
1   <html>
2     <head>
3       ...
4     </head>
5     <body>
6       <h1>Hi!</h1>
7       <button>
8         Click me!
9       </button>
10      <script src=”./events.js”></script>
11    </body>
12  </html>
```

Javascript (events.js)

```
1   function changeBackground() {
2     const header =
        document.querySelector(“h1”)
3     header.style.backgroundColor = “red”
4   }
5
6   const button =
      document.querySelector(“button”)
7   button.addEventListener(“click”,
      changeBackground)
```

Workshop #1

Stopwatch

Workshop #2

ToDo list

Object Oriented Programming

It’s a programming paradigm based on the concept of objects, which may contain the data and some logic, known as methods.

Objects are instances of classes, which defines the structure of them.

Classes

Class are defined with the usage of keyword class.
Constructor is a special kind of method, which is executed during object initialization.

```
1  class Person {
2    constructor(name, age) {
3      this.name = name
4      this.age = age
5    }
6  }
```

Objects

Objects are instances of classes.
They are created with the keyword new.
Name and age are fields - variables inside objects. This variables can be accessed with <object>.<field> expression (or <object>[<field>]).

```
1  const teacher1 = new Person(“Karol”, 25)
2  const teacher2 = new Person(“Bartek”, 12)
3
4  console.log(“Name:“, teacher1.name, “, Age:“, teacher1.age, “- so old”)
5  console.log(“Name:”, teacher2.name, “, Age:“, teacher2.age, “- baby”)
6  console.log(“Name:”, teacher2[name], “, Age:”, teacher2[age])
```

Methods

Methods are a part of class.

Within class, fields can be accessed through this keyword.

```
1   class person {
2     constructor(name, age) {
3       this.name = name
4       this.age = age
5     }
6
7     makeOlder() {
8       this.age++
9     }
10  }
```

Promise

Promises are mechanism that allows to hold a place for value that could not exist yet, but is expected to exist in the future.

At any time, promise can be in one of three states listed below:

- pending (not completed yet)
- fulfilled (completed, returned a value)
- rejected (completed with an error or failed)

Promise

Promises are extremely useful when working with asynchronous requests. They allow to execute some action, e.g. extract some portion of data, only when they are loaded.

```
1   function writeName(peopleArray) {
2     for(person in peopleArray) {
3       console.log(“Name:“, person.name, “, Age:“, person.age)
4   }
5
6   fetch()
7     .then(writeName)
8     .catch(function() {
9       console.log(“Error occurred.”)
10    })
```

Workshop #3

Egg timer

Workshop #4

F1 race tracking

Workshop #5

Calculator

Questions?