MENTORMATE™
DevCamp

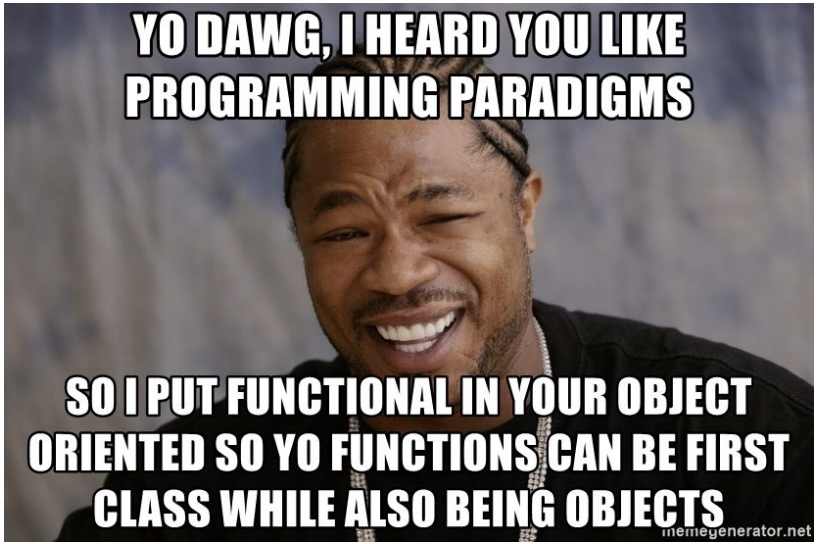
JavaScript Paradigms

Yulia Tenincheva

Senior Cloud Engineer, MentorMate

Agenda for today

- JavaScript Arrays, Text, Objects, Functions - **Syntax** & **Methods**
- OOP in JavaScript - The *classic* way vs **Classes**



Non-Primitive Data Types

- Object
- Array
- Function
- RegExp
- Date
- ... others

```
// Objects
typeof {name: "John", age: 18};  // Returns: "object"
// Dates
typeof new Date();               // Returns: "object"
// RegExp
typeof new RegExp();             // Returns: "object"
// Arrays
typeof [1, 2, 4];                // Returns: "object"
// Functions
typeof function(){};             // Returns: "function"
```

What are objects?

Objects in JavaScript are **collections of key/value pairs**.

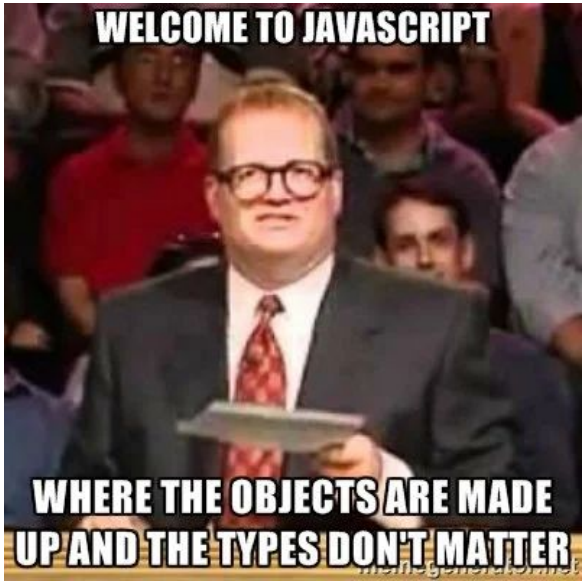
Values are often referred to as **properties**. All objects in JavaScript descend from the parent Object constructor.

```
const obj = { name: 'Pesho', age: 20 };

let prop = 'foo';
let o = {
    [prop]: 'hey',
    ['b' + 'ar']: 'there'
};
```

Characteristics of objects

- They are referential types
- Their properties (keys) are also objects.
- Each property can hold any value, including references to other objects
- They may hold circular references



The object spread operator & Destructuring

There's a special operator that allows us to "spread" the properties of one object into the properties of another object. It can be applied only to iterable objects.

```
const obj1 = { foo: 'bar' };
const obj2 = { baz: 'qax' };
const obj3 = { ...obj1, ...obj2 }; // merged
const clonedObj = { ...obj1}; // shallow-cloning
delete clonedObj.foo;
```

```
let a, b;
({a, b} = {a: 1, b: 2});
```

Object methods

```
const obj1 = { foo: 'bar' };
const obj2 = { baz: 'qax' };
const obj3 = Object.assign({}, obj1, obj2); // merged
```

Note that `Object.assign()` triggers `setters`, whereas spread syntax doesn't!

```
Object.keys({ foo: 'bar' }); // ['foo']
Object.values({ foo: 'bar' }); // ['bar']
Object.create(obj3); // {}
```

What is JSON?

- JSON stands for **JavaScript Object Notation**. It's made of key/value pairs
- It's is a lightweight format for storing and transporting data
- It's often used when data is sent from a server to a web page
- It's is "self-describing" and easy to understand

```
{
  "name": "Gosho",
  "age": 18,
  "hasDriversLicense": true,
  "cars": ["BMW", "Mercedes"],
  "contacts": { "personal": "+359111222333", "work": "+359333222111" }
}
```

Stringifying Objects & Parsing JSON

There's a special built-in method called JSON.stringify, which accepts an object

```
const obj = { foo: 'bar' };
const json = JSON.stringify(obj);
// '{ "foo": "bar" }'
```

There's a special built in method called JSON.parse(), which accepts a string.

```
const obj = JSON.parse(json);
// { foo: 'bar' }
```

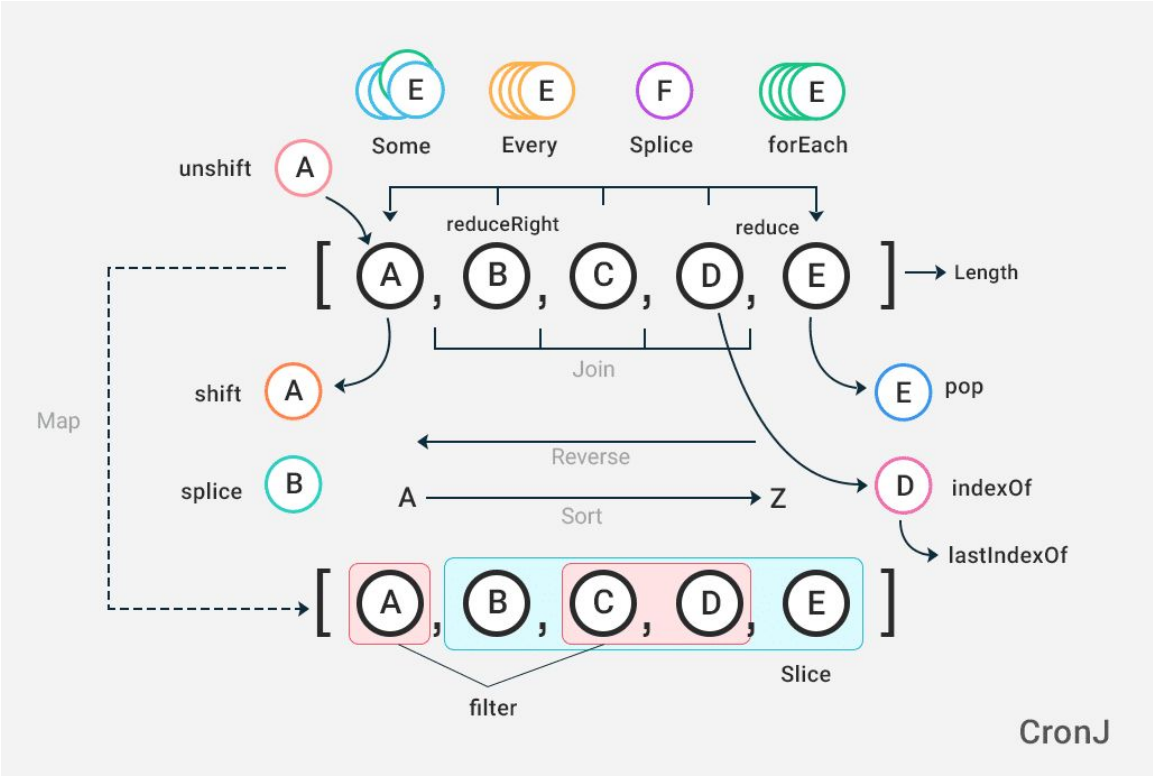
What are Arrays?

- Arrays in JS are **list-like objects**.
- They’re defined by using square brackets:

  `['array', 'of', 'strings']`
- Arrays can hold multiple values of any type. They’re indexed and can be traversed.
- Arrays have lots of useful methods.

Array methods

- length, indexOf()
- `slice(),splice()`
- `push(), pop()`
- shift(), unshift()
- some(), every()
- map(), filter(), reduce()
- reverse(), sort()



Array.splice() vs Array.slice()

- **slice()** method returns the selected element(s) in an array, as a new array object.

```
// Example with slice()
let email = 'john@example.com'
let localPart =
email.slice(0,email.indexOf('@'));

console.log(localPart); // "john"
```

- The **splice()** method returns the removed item(s) in an array. It can take n number of arguments.

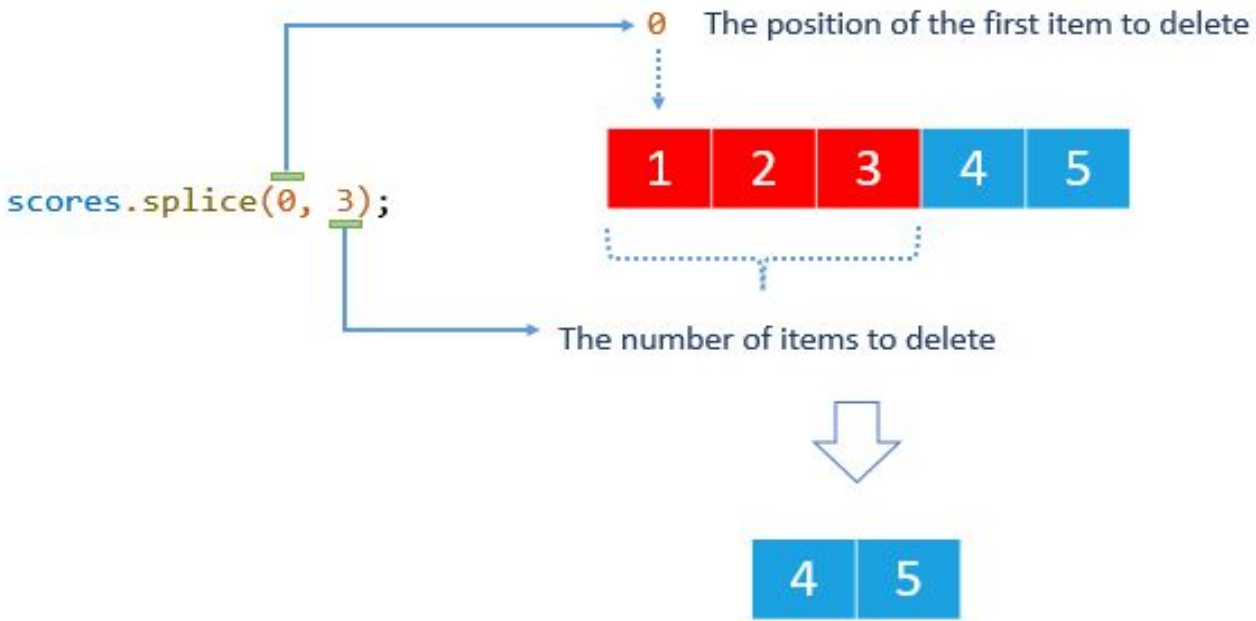
```
// Syntax
Array.splice(position, 0,new_element_1,new_element_2,...);


// Example
let colors = ['red', 'green', 'blue'];
colors.splice(2, 0, 'purple');

console.log(colors); // ["red", "green", "purple",
"blue"]
```

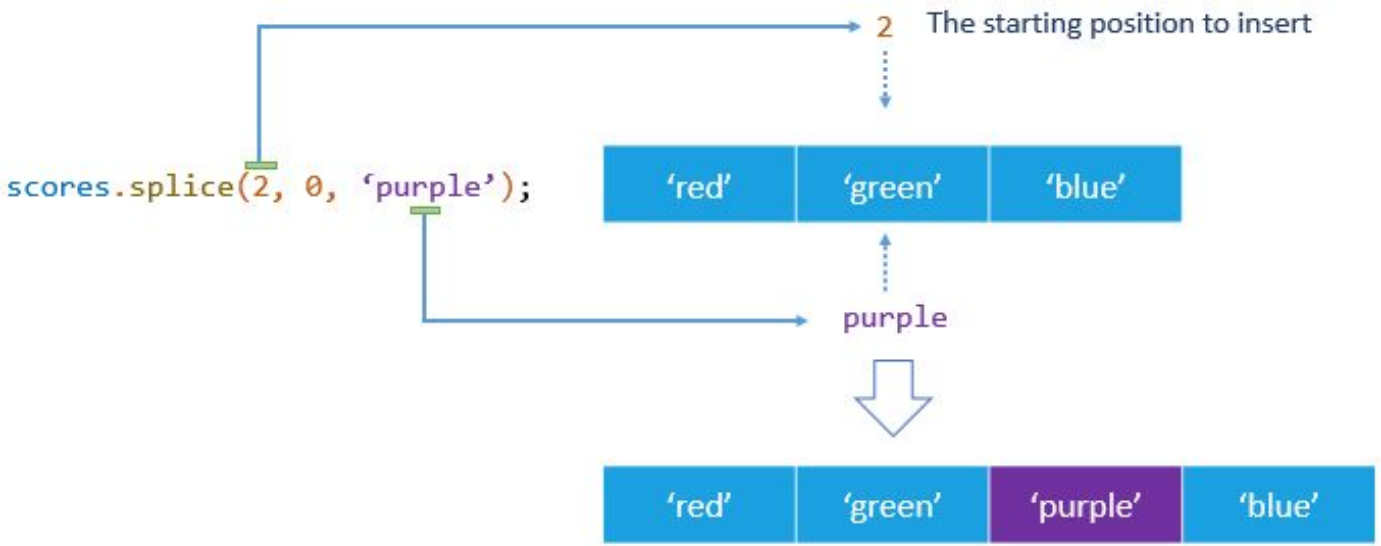
Array.splice() vs Array.slice()

- Removing elements using **splice().**

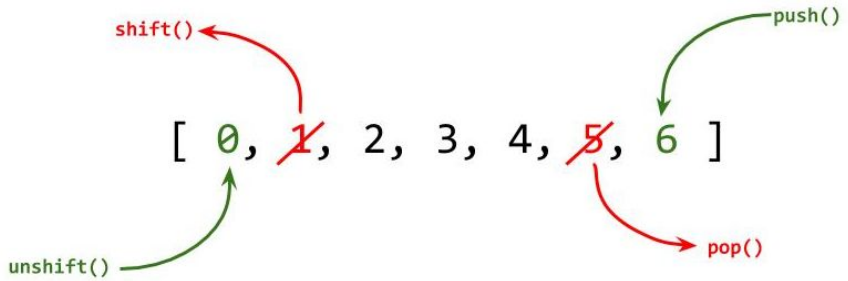


- Inserting elements using **splice().**

  Note that the **splice()** method actually changes the original array.



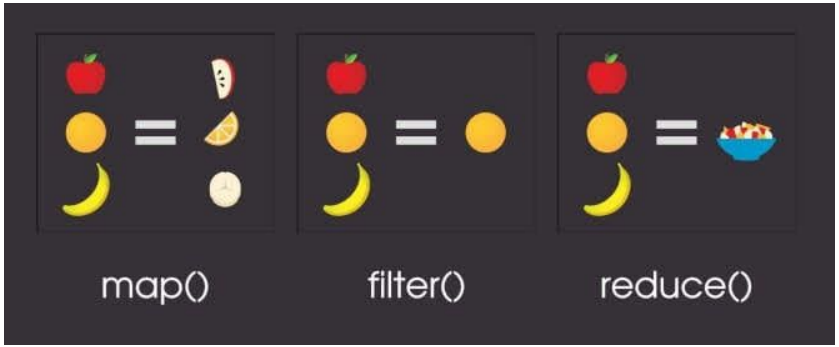
Array.pop(), push(), shift(), unshift()



Map, filter and reduce

```
const array = [1, 2, 3];

array.map(x => x * x); // [1, 4, 9]
array.filter(x => x > 1); // [2, 3]
array.reduce((a, b) => a + b, 0); //
6
array.find(x => x > 1); // 2
array.findIndex(x => x > 1); // 1
array.indexOf(2); // 1
```




```
array.filter(x => x > 1).map(x => x * x).find(x => x % 2 === 0) + "2"; // 42
```

String Methods

Method	Description
charAt(position)	Returns the character at the specified position (in Number).
charCodeAt(position)	Returns a number indicating the Unicode value of the character at the given position (in Number).
concat([string,,])	Joins specified string literal values (specify multiple strings separated by comma) and returns a new string.
indexOf(SearchString, Position)	Returns the index of first occurrence of specified String starting from specified number index. Returns -1 if not found.
lastIndexOf(SearchString, Position)	Returns the last occurrence index of specified SearchString, starting from specified position. Returns -1 if not found.
localeCompare(string,position)	Compares two strings in the current locale.
match(RegExp)	Search a string for a match using specified regular expression. Returns a matching array.
replace(searchValue, replaceValue)	Search specified string value and replace with specified replace Value string and return new string. Regular expression can also be used as searchValue.
search(RegExp)	Search for a match based on specified regular expression.

Method	Description
slice(startNumber, endNumber)	Extracts a section of a string based on specified starting and ending index and returns a new string.
split(separatorString, limitNumber)	Splits a String into an array of strings by separating the string into substrings based on specified separator. Regular expression can also be used as separator.
substr(start, length)	Returns the characters in a string from specified starting position through the specified number of characters (length).
substring(start, end)	Returns the characters in a string between start and end indexes.
toLocaleLowerCase()	Converts a string to lower case according to current locale.
toLocaleUpperCase()	Converts a sting to upper case according to current locale.
toLowerCase()	Returns lower case string value.
toString()	Returns the value of String object.
toUpperCase()	Returns upper case string value.
valueOf()	Returns the primitive value of the specified string object.

Let’s use all of them!

Task:

Write a JavaScript function reverseWordsInString(str) to reverse the characters of every word in the string but leaves the words in the same order.
Words are considered to be sequences of characters separated by spaces. Write a JavaScript program reverseWords.js that prints on the console the output of the examples below:

1. “Java is to JavaScript what car is to carpet”
2. “The strength of JavaScript is that you can do anything. The weakness is that you will.”

Regular Expressions

Practice here: https://regex101.com/

- **The literal notation's** parameters are enclosed between slashes and do not use quotation marks.
- **The constructor function's** parameters are not enclosed between slashes but do use quotation marks.

The following three expressions create the same regular expression:

```
/ab+c/i
new RegExp(/ab+c/, 'i') // literal notation new
RegExp('ab+c', 'i') // constructor
```

```
var re = /\w+\s/g;
var str = 'fee fi fo fum';
var myArray = str.match(re);

console.log(myArray);
// ["fee ", "fi ", "fo "]
```

Writing Regular Expression patterns

- Using simple patterns - `/abc/` and special characters

Characters / constructs	Corresponding article
`\`, `.`, `\cX`, `\d`, `\D`, `\f`, `\n`, `\r`, `\s`, `\S`, `\t`, `\v`, `\w`, `\W`, `\0`, `\xhh`, `\uhhhh`, `\uhhhhh`, `[\b]`	Character classes
`^`, `$`, `x(?=y)`, `x(?!y)`, `(?<=y)x`, `(?<!y)x`, `\b`, `\B`	Assertions
`(x)`, `(?:x)`, `(?<Name>x)`, `x|y`, `[xyz]`, `[^xyz]`, `\Number`	Groups and ranges
`*`, `+`, `?`, `x{n}`, `x{n,}`, `x{n,m}`	Quantifiers
`\p{UnicodeProperty}`, `\P{UnicodeProperty}`	Unicode property escapes

Using Regular Expression in JavaScript

Method	Description
`exec()`	Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.
`test()`	Tests for a match in a string. It returns `true` or `false`.
`match()`	Returns an array containing all of the matches, including capturing groups, or `null` if no match is found.
`matchAll()`	Returns an iterator containing all of the matches, including capturing groups.
`search()`	Tests for a match in a string. It returns the index of the match, or `-1` if the search fails.
`replace()`	Executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`replaceAll()`	Executes a search for all matches in a string, and replaces the matched substrings with a replacement substring.
`split()`	Uses a regular expression or a fixed string to break a string into an array of substrings.

Functions

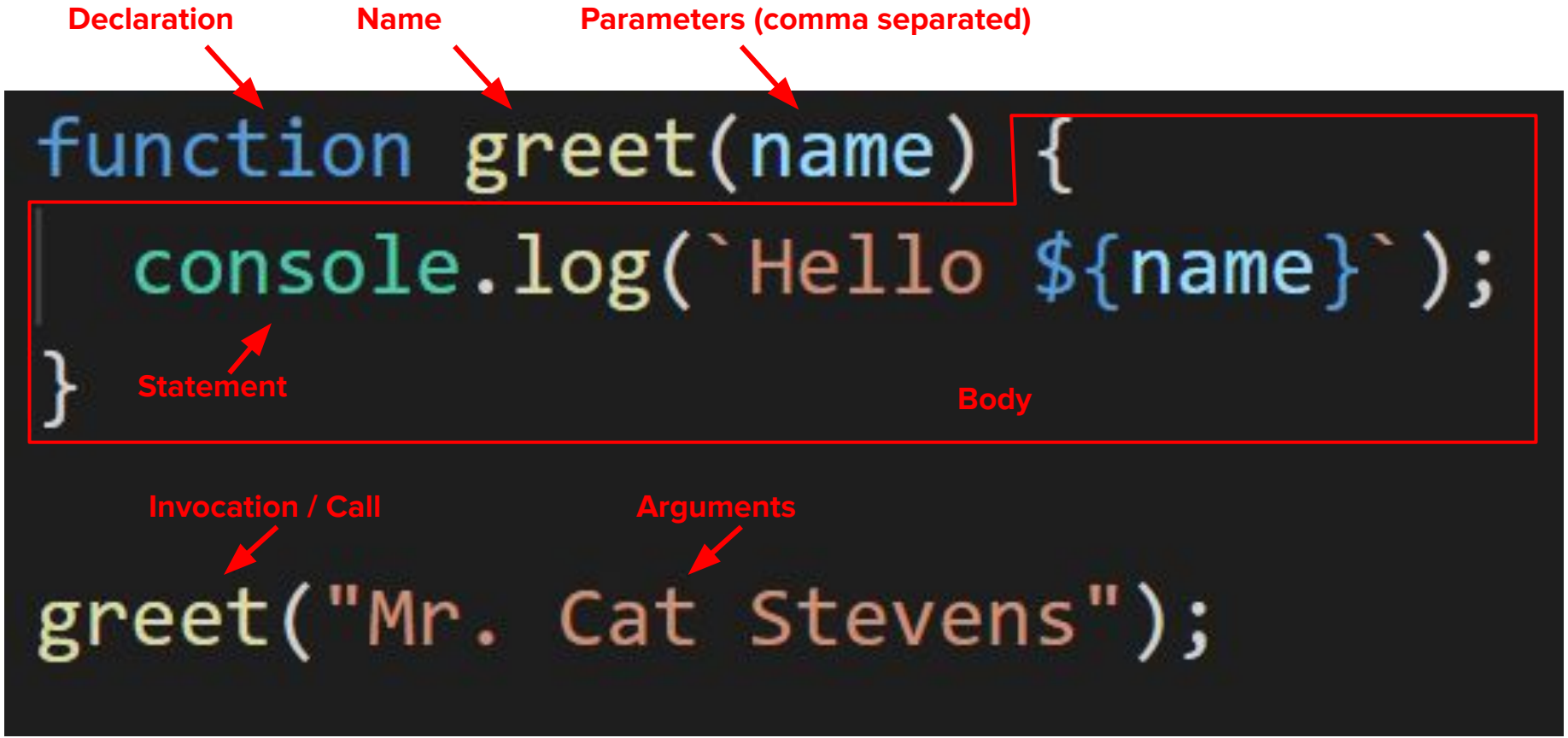
What are functions?

Functions are reusable blocks of code.

- In JavaScript, functions are also objects. You can work with functions as if they were ordinary objects.
- You can assign functions to variables, to array elements, and to other objects.
- They can also be passed around as arguments to other functions or be returned from those functions.

Characteristics of functions

- Referential types
- Function declarations are **hoisted**
- Can be assigned to objects - we refer to them as **methods** in this case
- Can be used as expressions (function expressions)
- Can accept other functions as parameters
- Can return anything, even other functions **-** when a function returns another function, the returned function is called a **closure**, which has access to the scope of the outer function
- They return undefined by default if return is omitted
- Can call themselves (recursion)
- There are some special kinds of functions - **arrow**, **async**, **generator** and **constructor**


Declaration
Name
Parameters (comma separated)
function greet(name) {
console.log(`Hello ${name}`);
}
Statement
Body
Invocation / Call
Arguments
greet("Mr. Cat Stevens");

Arrow functions

An **arrow function expression** is a syntactically compact alternative to a regular function expression, although without its own bindings to the `this`, `arguments`, `super`, or `new.target` keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

```
const sum = (a, b) => a + b;
const multiply = (a, b) => a * b;
sum(multiply(2, 2), multiply(3, 3)); // 13
[1, 2, 3].filter(x => x > 1); // [2, 3]
[1, 2, 3].map(x => multiply(x, 2)); // [2, 4, 6]
[1, 2, 3].reduce(sum, 0); // 6
```

Generator functions

A function that returns an iterator. Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.

```
function* idGenerator() {
  let i = 1;
  while (true) {
    yield i++;
  }
}

const iterator = idGenerator();
iterator.next(); // { value: 1; done: false }
iterator.next(); // { value: 2; done: false }
iterator.next(); // { value: 3; done: false }
```

Immediately Invoked Function Expression - IIFE

An IIFE is a JavaScript function that runs as soon as it is defined.

```
(function() {
  console.log(`I'm a very simple iife`);
})();
```

```
(function(foo) {
  console.log(`I'm a very simple iife`);
})(1);
```

Closures

A **closure** is a function bundled together (enclosed) with references to its surrounding state (the **lexical environment, a.k.a scope**).

In JavaScript, closures are created every time a function is created.
Each closure has 3 scopes:

- Local scope (own scope)
- Outer functions scope
- Global scope

```
function createCounter() {
  let privateCounter = 1;
  return function() {
    return privateCounter++;
  };
}

const counter1 = createCounter();
const counter2 = createCounter();
counter1(); // 1
counter1(); // 2
counter2(); // 1
```

This function has access to the outer function's scope

Function Expression

*var x = function (a, b) {return a * b}; // the variable can be used as a function now*

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

Function declarations load before any code is executed while **Function expressions** load only when the interpreter reaches that line of code. Similar to the `var` statement, function declarations are hoisted to the top of other code.

Function Expressions can serve as closuress, IIFEs and arguments to other functions.

Call, bind and apply

- **call** - used to invoke a function and provide an execution context (this) The arguments are individually passed

- **apply** - used to invoke a function and provide an execution context (this) The arguments are passed as an array

- **bind** - creates a new function from the original one, that's bound to an execution context (this). The execution context that the function is bound to cannot be overridden even by consecutive calls to bind!

Call, bind and apply example

```
function greet(name) {
  console.log(`Hi ${name}, I'm ${this.name}`);
}

greet('Pesho'); // Hi Pesho, I'm undefined
greet.call({ name: 'Gosho' }, 'Pesho'); // Hi Pesho, I'm Gosho
greet.apply({ name: 'Gosho' }, ['Pesho']); // Hi Pesho, I'm Gosho
greet.bind({ name: 'Gosho' })('Pesho'); // Hi Pesho, I'm Gosho
```

What is “this”?

The **this** keyword refers to an object, or more specifically - the object that is executing the current bit of js code. In other words, every js function while executing has a reference to its current execution context, called **this**.

```
function greet(name) {
  console.log(`Hi ${name}, I'm ${this.name}`);
}
greet('Anna'); // Hi Anna, I'm undefined
const gosho = { name: 'Gosho', greet };
gosho.greet('Anna'); // Hi Anna, I'm Gosho
```

Recursive functions

```
function factorial(n) {
  if (n === 1) {
    return 1;
  }

  return n * factorial(n - 1);
}

factorial(5); // 120
```

Bottom of recursion

Recursive call

OOP in JavaScript

Constructors and object instances

JavaScript uses special functions called **constructor functions** to define and initialize objects and their features.

```
function createNewPerson(name) {
  const obj = {};
  obj.name = name;
  obj.greet = function() {
    console.log(`Hi! I\'m ${this.name}.`);
  };
  return obj;
}

const salva = createNewPerson('Salva');
salva.name; salva.greet();
```

```
function Person(name) {
  this.name = name;
  this.greet = function() {
    console.log(`Hi! I\'m ${this.name}.`);
  };
}

let person1 = new Person('Bob');
let person2 = new Person('Sarah');

person1.greet();
```

What is the “new” operator?

The **`new` operator** lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function. The **`new`** keyword does the following things:

1. Creates a blank, plain JavaScript object;
2. Links (sets the constructor of) this object to another object;
3. Passes the newly created object from *Step 1* as the `this` context;
4. Returns `this` if the function doesn't return its own object.

```
function Person(name) { this.name = name; }
const [pesho, misho] = [new Person('Pesho'), new Person('Misho')];
typeof pesho; // object
pesho instanceof Person; // true
misho.__proto__ === pesho.__proto__; // true
```

Every JavaScript object has a prototype.

All objects in JavaScript inherit their methods and properties from their prototypes.

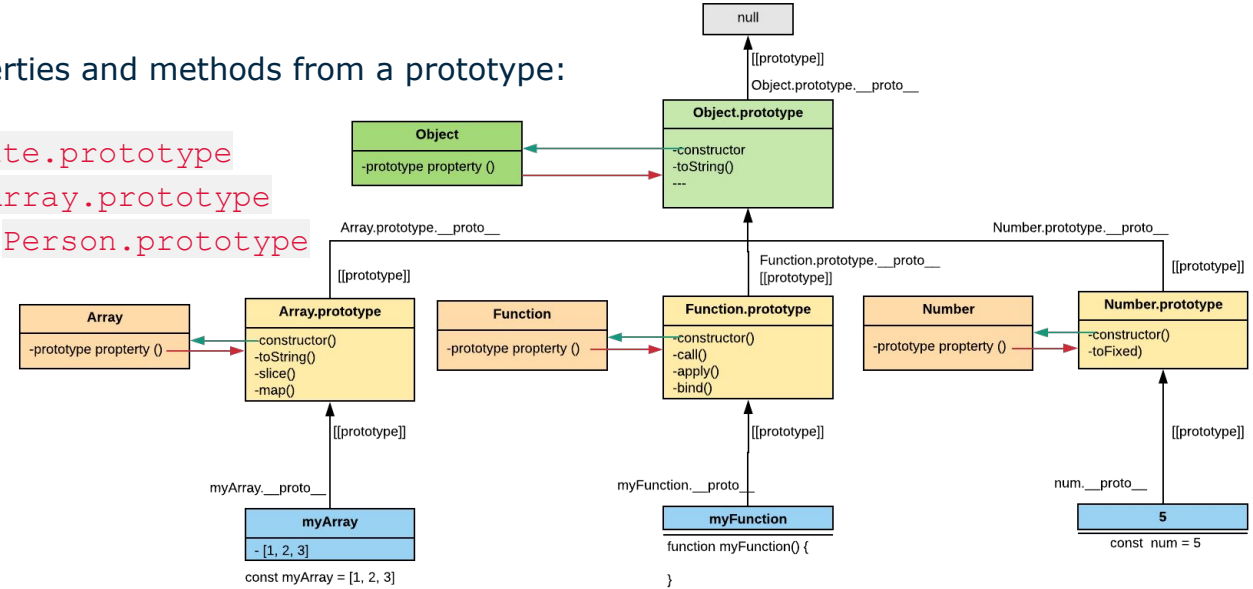
What is the prototype chain?

- When it comes to inheritance, JavaScript only has one construct: **objects**
- Each object has a property which **holds a link to another object** called its **prototype**
- That prototype object (`__proto__`) has a prototype of its own, and so on until an object is reached with **null** as its prototype
- By definition, **null has no prototype**, and acts as the final **link in the prototype chain**.
- Understanding prototypes in JavaScript

Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype:

- `Date` objects inherit from `Date.prototype`
- `Array` objects inherit from `Array.prototype`
- `Person` objects inherit from `Person.prototype`



The `Object.prototype` is on the top of the prototype inheritance chain:

`Date` objects, `Array` objects, and `Person` objects inherit from `Object.prototype`.

Prototype Inheritance

```
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  return `Hi! I\'m ${this.name}.`;
};

let person1 = new Person('Bob');
let person2 = new Person('Sarah');

person1.greet();
```

```
function Employee(name, title) {
  Person.call(this);
  this.name = name;
  this.title = title;
}

Employee.prototype = Object.create(Person.prototype);

let employee = new Employee('Pesho', 'FullStack
Developer');
employee.greet();
```

Extending Built-in Objects

```
Array.prototype.shuffle = function () {
    let input = this;

    for (let i = input.length - 1; i >= 0; i--) {
        let randomIndex = Math.floor(Math.random() * (i + 1));
        let itemAtIndex = input[randomIndex];
        input[randomIndex] = input[i];
        input[i] = itemAtIndex;
    }
    return input;
}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
numbers.shuffle();
```

You don’t control the built-in object’s future! Some functionality just should not be extended or overridden

Classes

- **Syntactic sugar** introduced in ES6 over prototype-based inheritance. The class syntax **does not** introduce a new object-oriented inheritance model to JavaScript. **Classes are just functions underneath**.
- They support subclassing, using the "**extends**" keyword.
- Only a single superclass is supported at a time
- The parent class can be accessed with the "**super**" keyword
- Class properties can have **getters** and **setters**
- Classes can have **static** properties and methods

ES6 Classes

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name;
  }

  set name(name) {
    this._name = name;
  }
}
```

```
class Trainee extends Person {
  constructor(name, track){
    super(name);
    this._track = track;
  }

  practiceJavaScript(){
    return `${this._name} is sweating`;
  }
}

const pesho = new Trainee('Pesho');
pesho.practiceJavaScript();
```

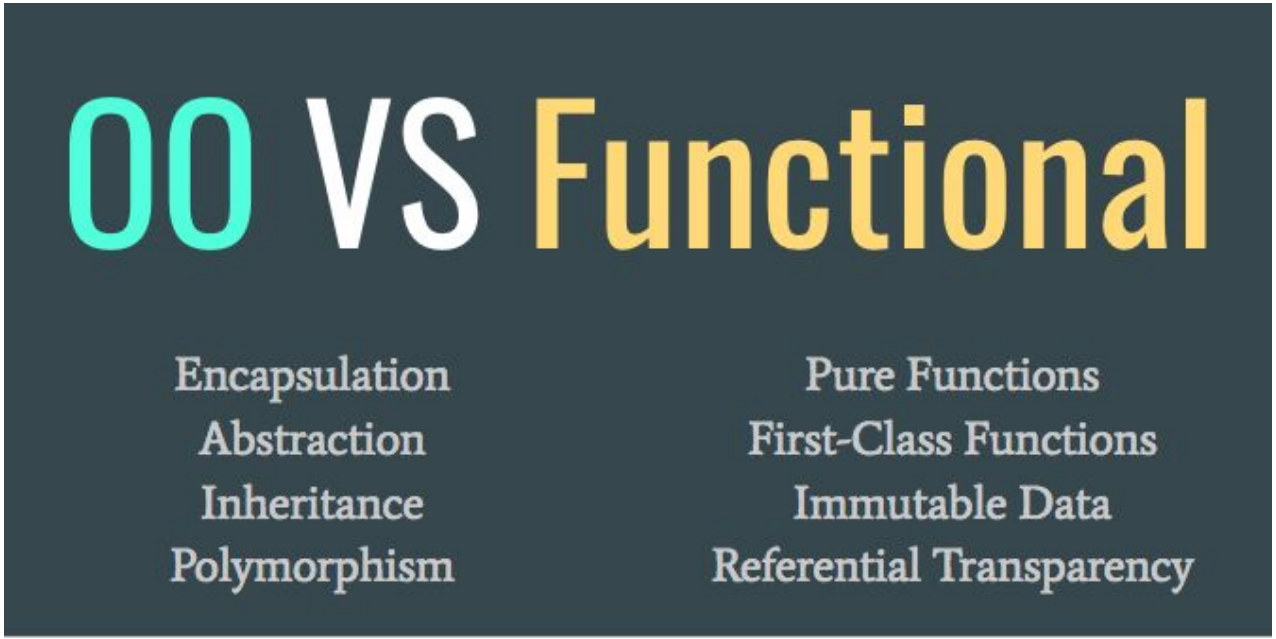
ES2020 Private Class Variables

```
class Message {
  #message = "Howdy"

  greet() { console.log(this.#message) }
}

const greeting = new Message()

greeting.greet() // Howdy
console.log(greeting.#message) // Private name #message is not defined
```


OO VS Functional
Encapsulation
Abstraction
Inheritance
Polymorphism
Pure Functions
First-Class Functions
Immutable Data
Referential Transparency

Homework