INTERNET SYSTEMS

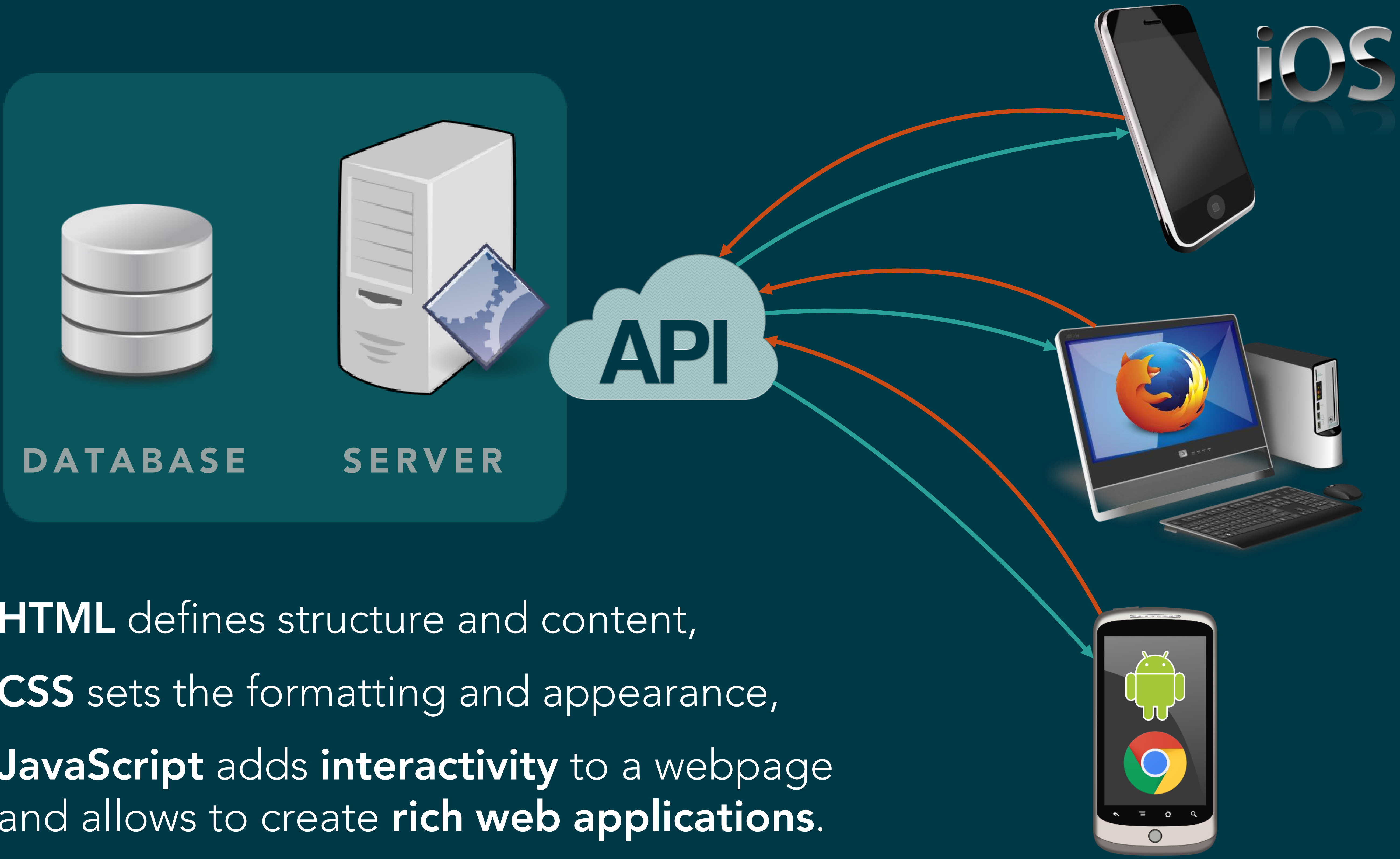
JAVASCRIPT

TOMASZ PAWLAK, PHD
MARCIN SZUBERT, PHD
INSTITUTE OF COMPUTING SCIENCE, POZNAN UNIVERSITY OF TECHNOLOGY

PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- Basic JavaScript
- *JavaScript: The Good Parts*
- *JavaScript: The Bad Parts*
- Languages that compile to JavaScript
- ECMAScript 6

MODERN WEB APPLICATION



HTML defines structure and content,

CSS sets the formatting and appearance,

JavaScript adds **interactivity** to a webpage and allows to create **rich web applications**.

WHY JAVASCRIPT?

- JavaScript is the language of the web browser — it is **the most widely deployed programming language** in history
- At the same time, it is one of **the most despised** and **misunderstood** programming languages in the world
- *The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with **enormous expressive power**. It is **even better when you know what you're doing***

— *JAVASCRIPT: THE GOOD PARTS*, DOUGLAS CROCKFORD

WHY JAVASCRIPT?

- **Q**: *If you had to start over, what are the technologies, languages, paradigms and platforms I need to be up-to-date and mastering in my new world of 2014?*
- **A**: *Learn one language you can build large systems with AND also* ***learn JavaScript****.*
- *JavaScript is the language of the web. The web will persist and the web will win. That's why I suggest you learn JavaScript*

— SCOTT HANSELMAN, 2014

WHAT IS JAVASCRIPT?

- JavaScript is a cross-platform, object-oriented, functional, lightweight, **small** scripting language.
- JavaScript contains a **standard library of built-in objects**, such as `Array` and `Math`, and a core set of language elements such as operators, control structures, and statements.
- **Core JavaScript** can be extended for a variety of purposes by supplementing it with **additional objects**:
    - **Client-side JavaScript** extends the core language by supplying objects to control a browser and its **Document Object Model** (**DOM**).
    - **Server-side JavaScript** extends the core language by supplying objects relevant to running JavaScript on a server, e.g., file and database access.

DOCUMENT OBJECT MODEL

- The Document Object Model (DOM) is a **programming interface** for HTML, XML, and SVG documents.
- DOM provides a **structured representation** of the document (a **tree**) and it defines a way that the structure can be accessed from programs and scripts so that they can **change** the document **structure**, **style** and **content**.
- All of the **properties**, **methods**, and **events** available for manipulating and creating web pages are organized into **objects** (e.g., the **`document`** **object**).
- Although DOM was designed to be **language-independent**, it is usually accessed using JavaScript.

PRESENTATION OUTLINE

- What is JavaScript?
- **Historical Perspective**
- Basic JavaScript
- *JavaScript: The Good Parts*
- *JavaScript: The Bad Parts*
- Languages that compile to JavaScript
- ECMAScript 6

HISTORICAL PERSPECTIVE

- *1995*: **JavaScript** was created by **Brendan Eich** in **10 days** during his work for **Netscape** (before getting the license from Sun the language was called **Mocha** and **LiveScript**).
- *1996*: Microsoft implemented the same language, under the name **JScript**, in Internet Explorer 3.0.
- *1996*: Netscape submitted JavaScript to **ECMA International** for consideration as an **industry standard** — the specification **ECMA-262** standardizes the **ECMAScript** programming language.
    - Q: Why the standard is not named JavaScript?
- *1997*: **The first edition** of ECMA-262 specification; the current version of the ECMAScript standard is **10** (released in 2019).

HISTORICAL PERSPECTIVE

- *1995*: **JavaScript** was created by **Brendan Eich** in **10 days** during his work for **Netscape** (before getting the license from Sun the language was called **Mocha** and **LiveScript**).
- *1996*: Microsoft implemented the same language, under the name **JScript**, in Internet Explorer 3.0.
- *1996*: Netscape submitted JavaScript to **ECMA International** for consideration as an **industry standard** — the specification **ECMA-262** standardizes the **ECMAScript** programming language.
  - Q: Why the standard is not named JavaScript?
  - A: JavaScript is a registered trademark of Oracle Corporation
- *1997*: **The first edition** of ECMA-262 specification; the current version of the ECMAScript standard is **10** (released in 2019).

HISTORICAL PERSPECTIVE

- *1997*: **Dynamic HTML** appeared first in Internet Explorer 4 and in Netscape Navigator 4; it allows to dynamically change the content and appearance of a web page by manipulating the **DOM**.

- *1999*: **XMLHttpRequest** introduced in IE 5; this API lets a client-side script send an HTTP request to a server and get back data, usually in a text format (XML, JSON).

- *2001*: **Douglas Crockford** named and documented **JSON**, whose main idea is to use JavaScript syntax to store structured data in text format.

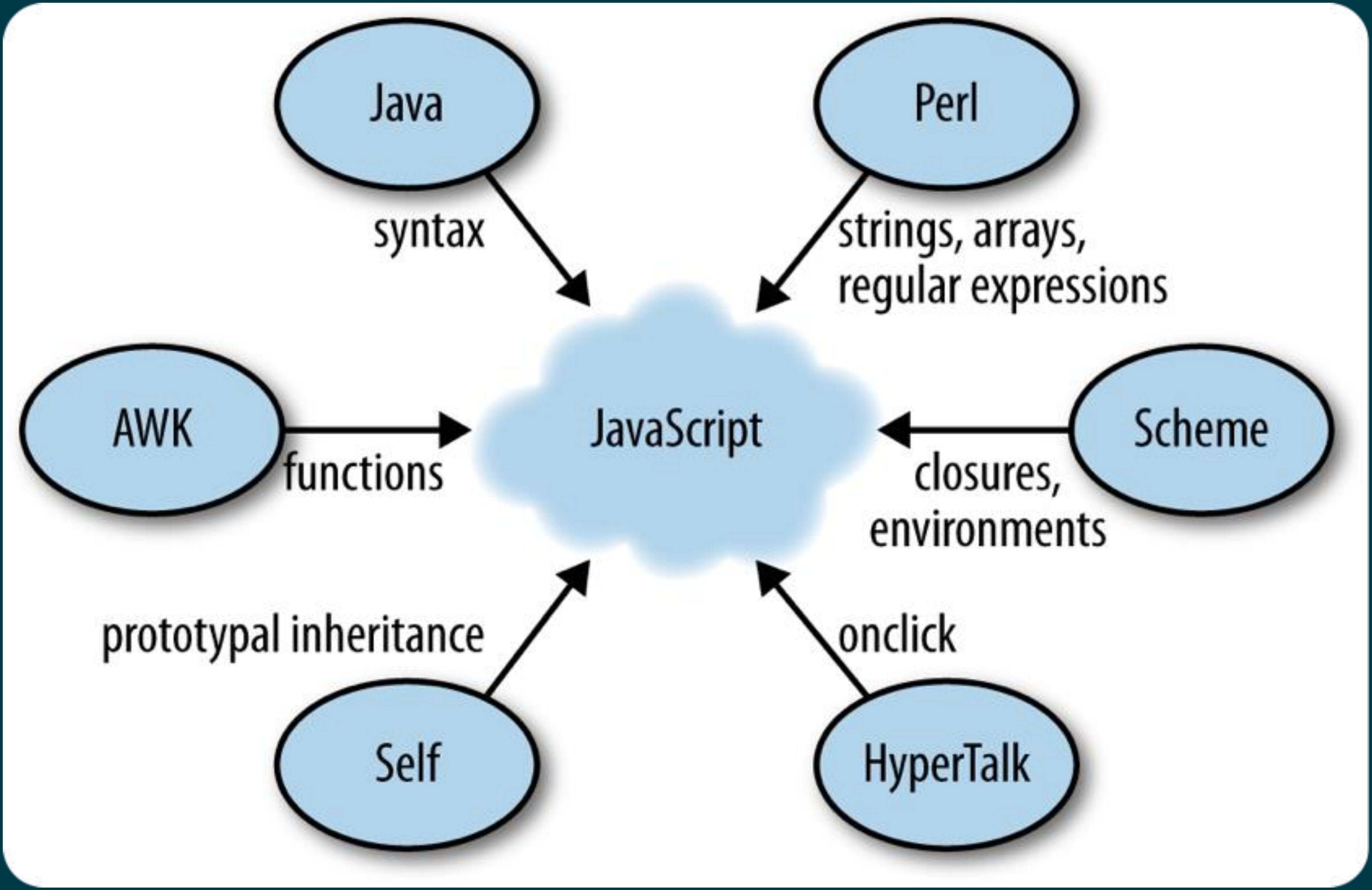
- **2005: Asynchronous Javascript and XML (AJAX)** — a collection of technologies (DOM, XML, JavaScript, XMLHttpRequest) that brings a level of interactivity to web pages that rivals that of desktop applications.

- ***2006*: jQuery** made DOM manipulation easier by abstracting over browser differences and by providing a powerful fluent-style API for querying and modifying the DOM.

HISTORICAL PERSPECTIVE

- *2007*: **V8 JavaScript Engine** developed by Google for Chrome:
  - V8 compiles JavaScript to **native machine code** before execution
  - V8 changed the perception of JavaScript as being slow and led to a speed race with other browser vendors, from which we are still profiting.
  - V8 can be used in a **browser** or as a **standalone high-performance engine**.
- *2009*: **Node.js** — an open source, cross-platform runtime environment for **server-side** applications written in JavaScript; Node.js uses the Google V8 JavaScript engine to execute code.
- *2009*: **CommonJS** — a project aimed at standardizing API for use of JavaScript outside the browser was founded.
- *2015*: **ECMAScript 6** — the new version of the standard revolutionized programming model and brought features from other modern programming languages.

JAVASCRIPT WAS INFLUENCED BY SEVERAL PROGRAMMING LANGUAGES



NATURE OF JAVASCRIPT

- **Dynamic** — many things can be changed at runtime: you can freely add and remove properties of objects after they have been created.
- **Dynamically typed** — variables can always hold values of any type.
- **Functional** and **object-oriented** — JavaScript supports two programming language paradigms.
- **Deployed as source code** — JavaScript is always deployed as source code and compiled by JavaScript engines.
- **Part of the Web Platform** — JavaScript is such an essential part of the web platform (HTML5 APIs, DOM, etc.) that it is easy to forget that the former can also be used without the latter.

PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- **Basic JavaScript**
- *JavaScript: The Good Parts*
- *JavaScript: The Bad Parts*
- Languages that compile to JavaScript
- ECMAScript 6

SYNTAX

The JavaScript syntax should be **immediately familiar** to anybody who has experience with **C-family languages**, such as C++, Java, C#, and PHP.

```
var x; // declaring a variable (dynamic typing)

x = 3 + y; // assigning a value to the variable `x`

foo(x, y); // calling function `foo` with parameters `x` and `y`
obj.bar(3); // calling function `bar` of object `obj`

// A conditional statement
if (x == 0) { // Is `x` equal to zero?
    x = 123;
}

// Defining function `baz` with parameters `a` and `b`
function baz(a, b) {
    return a + b; // semicolons are optional in JavaScript.
}
```

VARIABLES

- The names of variables, called **identifiers**, conform to certain rules — an identifier must start with a **letter**, **underscore**, or **dollar** sign; subsequent characters can also be **digits**.
- Variables can be declared:
  - With the keyword `var`
    - Functional scope: no matter where a variable is declared, it is available in whole function

VARIABLES

- The names of variables, called **identifiers**, conform to certain rules — an identifier must start with a **letter**, **underscore**, or **dollar** sign; subsequent characters can also be **digits**.
- Variables can be declared:
  - With the keyword `var`
    - Functional scope: no matter where a variable is declared, it is available in whole function
  - With the keyword `let`
    - Block scope: variable is available in a current block delimited by curly brackets `{}`

VARIABLES

- The names of variables, called **identifiers**, conform to certain rules — an identifier must start with a **letter**, **underscore**, or **dollar** sign; subsequent characters can also be **digits**.
- Variables can be declared:
  - With the keyword `var`
    - Functional scope: no matter where a variable is declared, it is available in whole function
  - With the keyword `let`
    - Block scope: variable is available in a current block delimited by curly brackets `{}`
  - By simply assigning it with a value, e.g., `x=42` declares a **global variable**
    - Both `var` and `let` can be used in global scope as well

VARIABLES

- The names of variables, called **identifiers**, conform to certain rules — an identifier must start with a **letter**, **underscore**, or **dollar** sign; subsequent characters can also be **digits**.
- Variables can be declared:
  - With the keyword `var`
    - Functional scope: no matter where a variable is declared, it is available in whole function
  - With the keyword `let`
    - Block scope: variable is available in a current block delimited by curly brackets `{}`
  - By simply assigning it with a value, e.g., `x=42` declares a **global variable**
    - Both `var` and `let` can be used in global scope as well
- Global variables are in fact properties of the **global object**. In web pages the global object is `window`, so you can set and access global variables using the `window.`*`variable`* syntax.

HOISTING

- Hoisting means **moving to the beginning of a scope**.
- **Function declarations** are hoisted **completely** — it allows to call a function before it has been declared.
- **Variable declarations** are hoisted **partially**, without assignments made with them.

```
function foo() {
    console.log(tmp);
    if (false) {
        var tmp = 3;
    }
}
```

```
function foo() {
    var tmp; // hoisted declaration
    console.log(tmp);
    if (false) {
        tmp = 3; // assignment stays
    }
}
```

VALUES

- JavaScript makes distinction between values:
  - **primitive values** — immutable, compared by value
    - booleans: `true, false`
    - numbers: `1736, 1.351`
    - strings: `'abc', "abc"`
    - special values: `undefined`, `null`
  - **objects** — mutable, compared by reference
    - arrays: `[ 'apple', 'banana', 'cherry' ]`
    - functions: `function(x, y) { return x * y }`
    - regular expressions: `/^a+b+$/`
    - plain objects: `{ firstName: 'Jane', lastName: 'Doe' }`
- The three primitive types have corresponding constructors: `Boolean`, `Number`, `String` for creating **wrapper objects**.

BOOLEANS

- Wherever JavaScript expects a boolean you can provide any value and it is **automatically converted to boolean**.
- **Falsy values** are automatically converted to false:
  - `undefined, null`
  - Boolean: `false`
  - Number: `0, NaN`
  - String: `''`
- All other values — including all **objects** (even empty ones), **empty arrays**, and `new Boolean(false)` — are **truthy**.

NUMBERS

- JavaScript has a **single type** for all numbers: it treats all of them as 64-bit **floating-point numbers** (like `double` in Java).
- Internally, most **JavaScript engines optimize** and distinguish between floating-point numbers and **integers** (ECMAScript specification features **bitwise operators** for integers).
- A large class of **numeric type errors is avoided** — for instance, problems of overflow in short integers.
- Special numbers include:
  - **NaN** — the result of an operation that cannot produce a normal result
  - **Infinity** — represents all values greater than `1.79769313486231570e+308`.

STRINGS

- Strings can be created directly via **string literals** delimited by **single** or **double quotes**.

- Single characters are accessed via **square brackets**:

```
> var str = 'abc';
> str[1]
'b'
```

- The property `length` counts the number of characters

```
> 'abc'.length
3
```

- Strings are **concatenated** via the plus (+) operator, which converts the other operand to a string if one of the operands is a string.

```
> var messageCount = 3;
> 'You have ' + messageCount + ' messages'
'You have 3 messages'
```

OBJECTS

- An object is a **container of properties**, where a property has a **name** and a **value**:
  - a property **name** can be **any string**, including the empty string
  - a property **value** can be **any value**
- A property's value can be a **function**, in which case the property is known as a **method**; methods use `this` to refer to the object that was used to call them (usually* — `this` can be redefined).
- Objects in JavaScript are **class-free** — objects do not have classes, however from ES6 classes can be used to create objects with the same properties and a constructor used to create an object can be identified at runtime.
- JavaScript has a number of **predefined objects**. In addition, you can create your own custom objects.

OBJECT LITERALS

- Object literals provide a convenient notation for **creating new objects**; **an object literal** is a pair of curly brackets surrounding zero or more properties (**name/value** pairs) separated by commas;
- The quotes around a **name** are optional if the name is a **legal identifier**.

```
var flight = {
    airline_number: "Oceanic",
    "flight-number": 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

ACCESSING OBJECT PROPERTIES

- The **dot operator** provides a compact syntax for accessing properties whose **keys** are **legal identifiers**.

```
var jane = {
        name: 'Jane',
        describe: function() {
                return 'Person named ' + this.name;
        }
};

jane.name // get property `name` -> 'Jane'
jane.describe // get property `describe` -> [Function]
jane.unknownProperty // undefined
jane.describe()  // call method `describe` -> 'Person named Jane'
jane.name = 'John'; // set property `name`
jane.describe() // 'Person named John'
jane.surname = "Watson" // augmenting the object
delete jane.name // delete property
jane.name // undefined
```

ACCESSING OBJECT PROPERTIES

- If you want to read or write **properties** with **arbitrary names**, you need to use the **bracket operator**.
- While the dot operator works with **fixed property keys**, the bracket operator allows you to **refer to a property** via an **expression**.

```
var obj = {
        someProperty: 'abc',
        'not an identifier': 123,
        '6': 'bar',
        myMethod: function() {
                return true
        }
};

obj['some' + 'Property'] // 'abc'
var propKey = 'someProperty';
obj[propKey] // 'abc'
obj['not an identifier'] //123
obj[3 + 3] // key: the string '6' -> 'bar'
obj['myMethod']() //true
```

ARRAYS

- Arrays are **sequences of elements** that can be accessed via **integer indices** starting at **zero**.
- Arrays can be very fast data structures. Unfortunately, JavaScript does not have anything like this kind of array.
- Instead, JavaScript provides an **object** that acts as a **map** (dictionary) from indices to values — arrays may **not** be **contiguous**.
- Arrays are still objects and can have **object properties** — those are not considered array elements.
- Indices are numbers `i` in the range $0 \leq i < 2^{32}-1$. Indices that are out of range are treated as **normal property keys** (strings!)

ARRAY LITERALS

- **Array literals** provide a very convenient notation for **creating new array values**; an array literal is a pair of square brackets surrounding zero or more values separated by commas.

```
var empty = [];
var numbers = [
        'zero', 'one', 'two',
        'three', 'four', 'five',
        'six', 'seven', 'eight'
];

empty[1] // undefined
numbers[1] // 'one'
empty.length // 0
numbers.length // 9
```

```
var numbers_object = {
        '0': 'zero',
        '1': 'one',
        '2': 'two',
        '3': 'three',
        '4': 'four',
        '5': 'five',
        '6': 'six',
        '7': 'seven',
        '8': 'eight'
};
```

ARRAY METHODS

```
> var arr = [ 'a', 'b', 'c' ];
> arr.slice(1, 2)  // copy elements
[ 'b' ]

> arr.push('x')  // append an element
4
> arr
[ 'a', 'b', 'c', 'x' ]

> arr.pop()  // remove last element
'x'
> arr
[ 'a', 'b', 'c' ]

> arr.indexOf('b')  // find the first index of an element
1                   // use lastIndexOf to find the last index
> arr.indexOf('y')
-1

> arr.join('-')  // all elements in a single string
'a-b-c'
> arr.join()
'a,b,c'
```

ARRAY ITERATION

- In order to achieve the best performance when iterating over arrays, it is best to use the classic for loop:

```
var list = [1, 2, 3, 4, 5, ......100000000];
for (var i = 0, l = list.length; i < l; i++) {
        console.log(list[i]);
}
```

- Alternative **iteration methods**:

```
> ['a', 'b', 'c'].forEach(
        function(elem, index) {
                console.log(index + '. ' + elem);
        });
1. a
2. b
3. c
> [1, 2, 3].map(function(x) { return x * x })
[1, 4, 9]
```

REGULAR EXPRESSIONS

- **Regular expressions** are **patterns** used to match character combinations in strings. In JavaScript, regular expressions are **objects**.
- General syntax to create `RegExp` object:
  - `/pattern/flags`
- To check whether a pattern is found in a string, use the `test` method; for more information (but slower execution) use the `exec` method.
- For instance

```
/^a+b+$/.test('aaab') // true
/^a+b+$/.test('aaa') // false
/a(b+)a/i.exec('_abBba_aba_') // [ 'abBba', 'bBb' ]
```

REGULAR EXPRESSIONS

- Flags
    - `g` – global match; find all matches in a string (useful with `String.match()`)
    - `i` – ignore case
    - `m` – multiline; treat each line as separate input by boundary characters
    - `u` – unicode; treat pattern as a sequence of unicode code points
    - `y` – sticky; matches only from the index indicated by the `lastIndex` property of this regular expression in the target string

REGULAR EXPRESSION PATTERNS CHARACTER SETS & GROUPS

Set	Meaning
[xyz] [a-c]	Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character.
[^xyz] [^a-c]	A negated or complemented character set. It matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character.

Group	Meaning
(*x*)	Matches *x* and remembers the match. These are called capturing groups. The capturing groups are numbered according to the order of left parentheses of capturing groups, starting from 1. The matched substring can be recalled from the resulting array's elements [1], ..., [n].
\n	Where *n* is a positive integer. A back reference to the last substring matching the n parenthetical in the regular expression (counting left parentheses).
(?:*x*)	Matches *x* but does not remember the match. These are called non-capturing groups. The matched substring can not be recalled from the resulting array's elements [1], ..., [n].

REGULAR EXPRESSION PATTERNS
CHARACTER CLASSES

Class	Meaning
.	Matches any single character except line terminators: \n, \r, \u2028 or \u2029.
\d	Matches a digit character in the basic Latin alphabet. Equivalent to [0-9].
\D	Matches any character that is not a digit in the basic Latin alphabet. Equivalent to [^0-9].
\w	Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to [A-Za-z0-9_].
\W	Matches any character that is not a word character from the basic Latin alphabet. Equivalent to [^A-Za-z0-9_].
\s	Matches a single white space character, including space, tab, form feed, line feed and other Unicode spaces. Equivalent to [\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff].
\S	Matches a single character other than white space. Equivalent to [^ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff].
\t	Matches a horizontal tab.
\r	Matches a carriage return.
\n	Matches a linefeed.
\v	Matches a vertical tab.
\f	Matches a form-feed.
[\b]	Matches a backspace. (Not to be confused with \b)
\0	Matches a NUL character. Do not follow this with another digit.
\xhh	Matches the character with the code *hh* (two hexadecimal digits).
\uhhhh	Matches a UTF-16 code-unit with the value *hhhh* (four hexadecimal digits).
\	Escape character, indicates that the next character is not special and should be interpreted literally.

REGULAR EXPRESSION PATTERNS QUANTIFIERS

Quantifier	Meaning
*x**	Matches the preceding item *x* 0 or more times.
x+	Matches the preceding item *x* 1 or more times. Equivalent to {1,}.
x?	Matches the preceding item *x* 0 or 1 time.
x{*n*}	Where *n* is a positive integer. Matches exactly *n* occurrences of the preceding item *x*.
x{*n*,}	Where *n* is a positive integer. Matches at least *n* occurrences of the preceding item *x*.
x{*n*,*m*}	Where *n* and *m* are positive integers. Matches at least *n* and at most *m* occurrences of the preceding item *x*.
*x**? *x*+? *x*?? *x*{n}? *x*{n,}? *x*{n,m}?	Matches the preceding item *x* like *, +, ?, and {...} from above, however the match is the smallest possible match. **Quantifiers without ? are said to be greedy, since they select the largest range of matching characters. Those with ? are called "non-greedy".**

REGULAR EXPRESSION PATTERNS BOUNDARIES, ALTERNATION & ASSERTIONS

Boundary	Meaning
^	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character.
$	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character.
\b	Matches a word boundary. This is the position where a word character is not followed or preceded by another word-character, such as between a letter and a space. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero.
\B	Matches a non-word boundary. This is a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. Such as between two letters or between two spaces. The beginning and end of a string are considered non-words. Same as the matched word boundary, the matched non-word boundary is also not included in the match.

Alternation	Meaning
x|*y*	Matches either *x* or *y*.

Assertion	Meaning
x(?=*y*)	Matches *x* only if *x* is followed by *y*.
x(?!*y*)	Matches *x* only if *x* is not followed by *y*.

FUNCTIONS

- **Functions** are one of the **fundamental building blocks** in JavaScript — a function is a sequence of statements that performs a task or calculates a value.
- JavaScript has **first-class functions** — functions are **objects** that can be used like **any other value**:
  - Functions can be stored in variables, objects, and arrays.
  - Functions can be passed as arguments to functions, and functions can be returned from functions.
  - Functions can have properties and methods.
- To use a function, you must **define** it somewhere in the **scope** from which you wish to call it.

DEFINING FUNCTIONS

- A **function declaration** consists of the `function` keyword, followed by:
  - The optional name of the function,
  - The optional list of comma-separated arguments to the function, enclosed in parentheses,
  - The JavaScript statements that define the function, enclosed in curly brackets.

```
function square(number) {
        return number * number;
}
```

- Functions can also be created by a **function expression**. Such a function can be **anonymous**.

```
var square = function(number) { return number * number };
var x = square(4) // x gets the value 16
```

FUNCTION EXPRESSIONS

- **Function expressions** are convenient when passing a function as an argument to another function.
- The following example shows a `map` function being defined and then called with an anonymous function as its first parameter:

```
function map(f, a) {
        var result = [], i;
        for (i = 0; i != a.length; i++) {
                result[i] = f(a[i]);
        }
        return result;
}

map(function(x) {return x * x * x}, [0, 1, 2, 5, 10]);
```

FUNCTION PARAMETERS

- **Primitive parameters** are passed to functions **by value**; if the function changes the value of such a parameter, this change is not reflected outside.
- **Objects parameters** are passed by reference, they are never copied; if the function changes the object's properties, that change is visible outside.

```
function myFunc(theObject) {
        theObject.make = "Toyota";
}

var mycar = {
        make: "Honda",
        model: "Accord",
        year: 1998
};

var x = mycar.make; // x gets the value "Honda"

myFunc(mycar);
var y = mycar.make; // y gets the value "Toyota"
```

A WAY TO OVERRIDE THIS

- `apply()` and `call()` are two special functions that can be called on every function
  - Both of them accept `this` object as the first argument
  - The remaining arguments are the arguments for the function to be called
    - `apply()` accepts the arguments as an array
    - `call()` accepts the arguments as comma-separated list

```
var A = {
    name: 'a',
    getName: function () {
        return this.name;
    }
};

var B = {
    name: 'b'
};

A.getName.call(B); // b
```

PROMISES

- **Promise** is a design pattern which is increasingly used in JavaScript APIs.
- A promise object represents a future result of computation.
- A promise object has four states:
    - **Resolved** – the computation already succeeded
    - **Rejected** – the computation stopped with errors
    - **Pending** – none of the above
    - **Settled** – Resolved or Rejected
- It raises events when the state changes.
- A promise object supports **chaining**, i.e., resolving one promise may create another. It also supports a single point of error handling for a chain.

PROMISES
A BASIC EXAMPLE

- The Promise constructor takes a function that does computation, possibly asynchronously.
- That function takes two callbacks as arguments:
  - resolve – to be called when the function succeeds,
  - reject – to be called when the function fails.

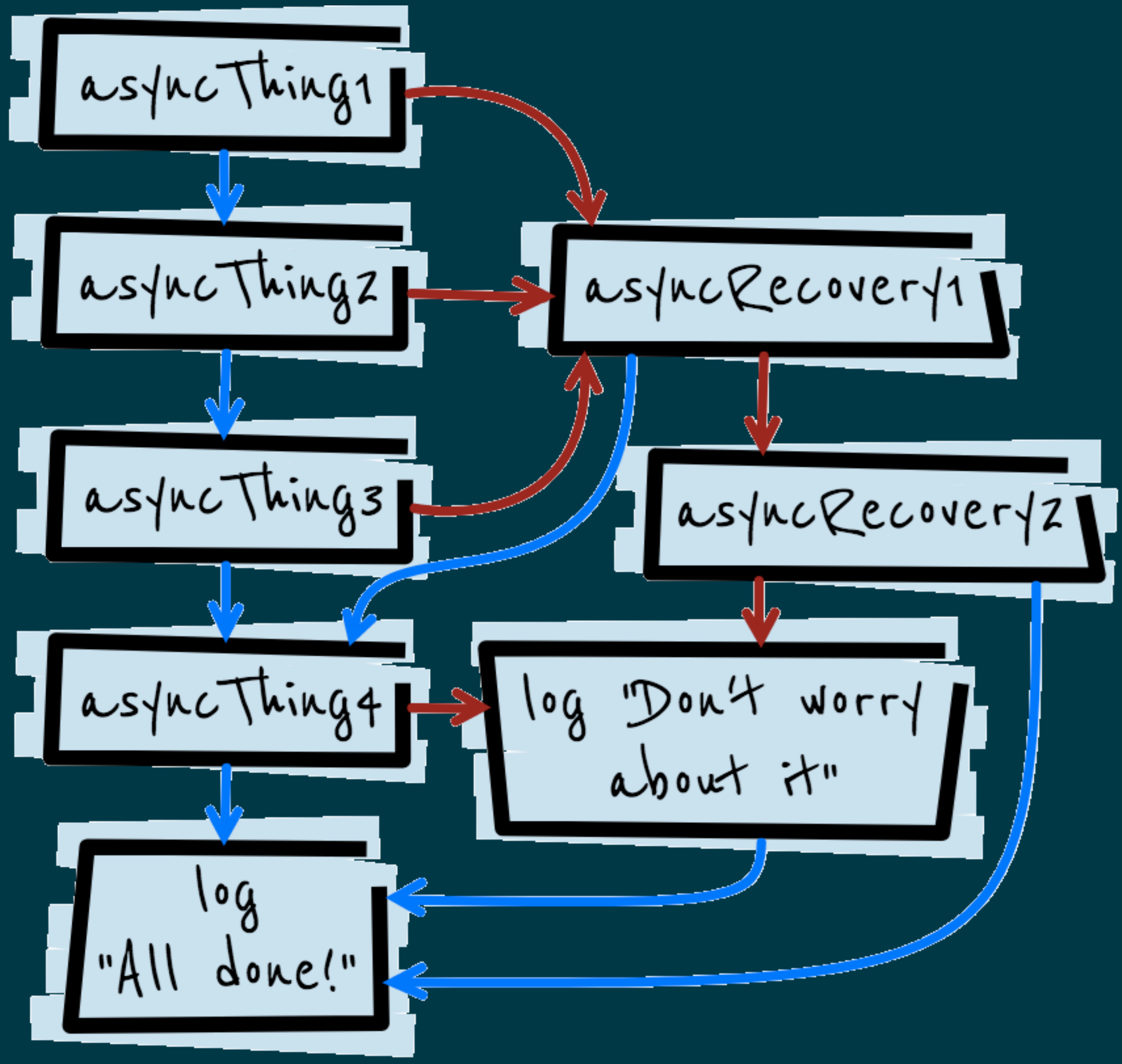
```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  } else {
    reject(Error("It broke"));
  }
});

promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}).catch(function(err) {
  console.log(err); // Error: "It broke"
});
```

PROMISES
A CHAINING EXAMPLE

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
})
```



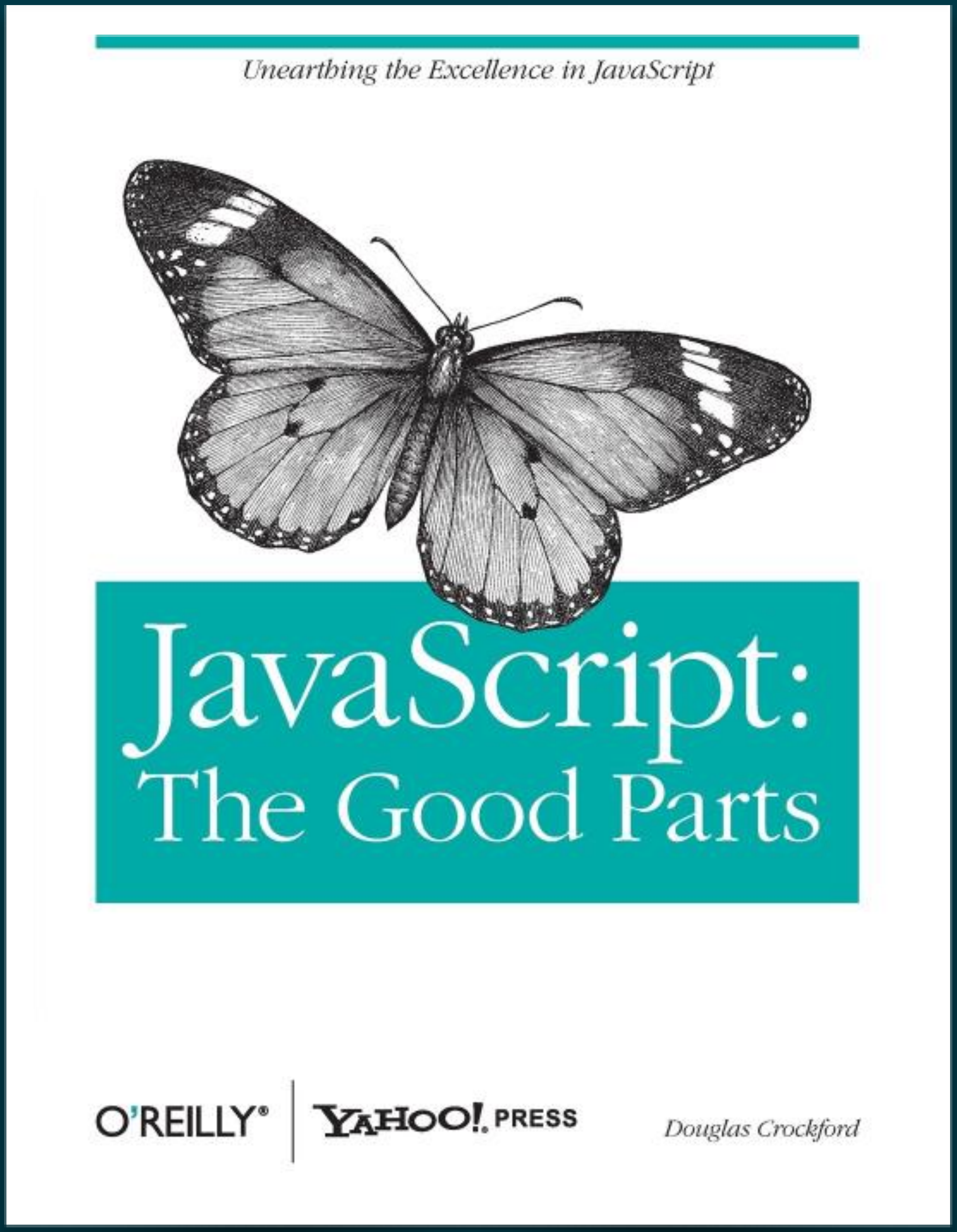
PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- Basic JavaScript
- ***JavaScript: The Good Parts***
- *JavaScript: The Bad Parts*
- Languages that compile to JavaScript
- ECMAScript 6

JAVASCRIPT: THE GOOD PARTS

- *Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts.*

- *JavaScript is a language with more than its share of bad parts. It went from non-existence to global adoption in an alarmingly short period of time.* ***It never had an interval in the lab when it could be tried out and polished.***


Unearthing the Excellence in JavaScript

JavaScript: The Good Parts

O'REILLY | YAHOO! PRESS

Douglas Crockford

BEAUTIFUL FEATURES

- *Fortunately, JavaScript has some extraordinarily good parts.*
- **Functions as first-class objects with lexical scoping** — Javascript is the first mainstream lambda language.
- **Dynamic objects with prototypal inheritance**
  - Objects are **class-free**.
  - Objects can be **dynamically extended** with new members.
  - An object can **inherit** members from another object.
- **Object literals and array literals — a convenient notation for creating new objects and arrays. JavaScript literals were the inspiration for the JSON data interchange format.**

FUNCTIONS

- *The best thing about JavaScript is its implementation of functions. It got almost everything right. But, as you should expect with JavaScript, it didn't get everything right.*
  —DOUGLAS CROCKFORD

- Functions in JavaScript are **objects** created with two additional **hidden properties**: the function's **context** and the code that implements the **behavior** of the function.

- Functions in JavaScript are so powerful, in fact, that they can completely **replace the need for objects**.

FUNCTIONS

- Functions in JavaScript are **first class objects** — they can be passed around like any other value; if a function is used as an argument or return value from another function, it's a **lambda**.
- **Higher-order functions** are functions that consume or return functions as data (lambdas).
- In JavaScript, lambdas are commonly used to:
  - pass an **anonymous function** as a **callback** to another, possibly an **asynchronous** function, to be executed when the function is complete.
  - attach **event handlers** for DOM interactions.
  - perform functional **enumeration** or **transformation**:

```
[5, 5, 5].forEach(function addTo(number) { result += number; });
[1, 0, 3, 0].filter(function (x) { return x !== 0 })
```

FUNCTION SCOPE

- **Function Scope** — variables defined inside a function cannot be accessed from anywhere outside the function; however, a function can access all variables and functions defined inside the scope in which it is defined.
- **Lexical scoping (static scoping)** — the scope of a variable is defined by its location within the source code.

```
function init() {
        var name = "Mozilla"; // name is a local variable

        function displayName() { // a nested function
                alert(name);
        }
        displayName();
}
init();
```

CLOSURES

- **Closures** are one of the **most powerful features** of JavaScript.
- A closure is a special **kind of object** that **combines two things**: a **function**, and the **environment** (**context**) in which that function was created; the environment consists of any local variables that were **in-scope** at the time that the closure was created.

```
function init() {
        var name = "Mozilla";

        function displayName() {
                alert(name);
        }
        return displayName;
}

var myFunc = init();
myFunc();
```

PRACTICAL CLOSURES

- A closure allows associate some **data** (the environment) with a **function** that operates on that data; this is similar to **object oriented programming**, where objects allow us to associate data (the object's properties) with one or more **methods**.

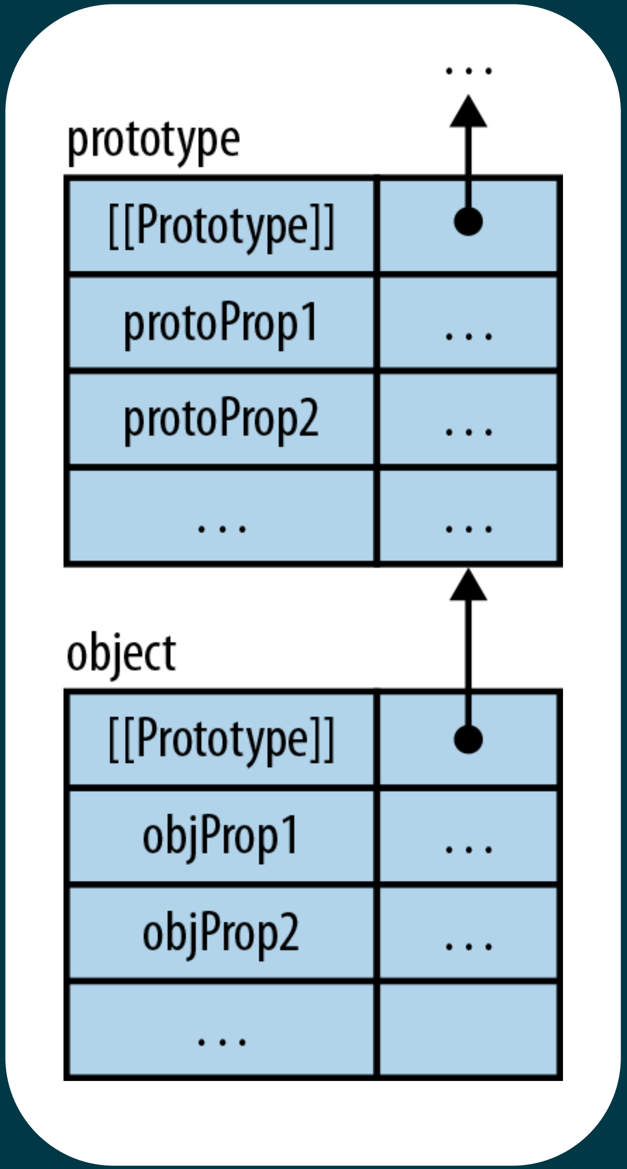
- Consequently, you can use a closure anywhere that you might normally use an **object with only a single method**.

- Such situations are common on the web where much of the code is **event-based** — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress).

- Closure is generally attached as a **callback**: a single function which is executed in response to the event.

PROTOTYPAL INHERITANCE

- JavaScript (until ES6) does not use a **classical inheritance model**; it is the only widely used language that features **prototypal inheritance**.
- Every object is linked to a **prototype object** from which it can **inherit properties**.
- The prototype relationship is a **dynamic relationship** — adding a new property to a prototype, makes it visible in all of the objects that are based on that prototype.



HTTP://SPEAKINGJS.COM

[[PROTOTYPE]] PROPERTY

- Following the ECMAScript standard, the notation `someObject.[[Prototype]]` is used to designate the prototype of `someObject`
- Since ECMAScript 2015, the `[[Prototype]]` is accessed using the accessors `Object.getPrototypeOf()` and `Object.setPrototypeOf()`
- This is equivalent to the JavaScript property `__proto__` which is non-standard but de-facto implemented by many browsers.

INHERITING PROPERTIES

```
// Let's assume we have object o, with its own properties a and b:
// {a: 1, b: 2}
// o.[[Prototype]] has properties b and c:
// {b: 3, c: 4}
// Finally, o.[[Prototype]].[[Prototype]] is null.
// Thus, the full prototype chain looks like:
// {a:1, b:2} ---> {b:3, c:4} ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called "property shadowing"

console.log(o.c); // 4
// Is there a 'c' own property on o? No, check its prototype.
// Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.

console.log(o.d); // undefined
// Is there a 'd' own property on o? No, check its prototype.
// Is there a 'd' own property on o.[[Prototype]]? No, check prototype.
// o.[[Prototype]].[[Prototype]] is null, stop searching,
// no property found, return undefined
```

CREATING NEW OBJECTS
OBJECT INITIALIZERS

```
var o = {a: 1};
// The created object o has Object.prototype as its prototype
// Object.prototype has null as its prototype.
// o ---> Object.[[Prototype]] ---> null

var a = ["yo", "whadup", "?"];
// Arrays inherit from Array.prototype
// (which has methods like indexOf, forEach, etc.)
// The prototype chain looks like:
// a ---> Array.[[Prototype]] ---> Object.[[Prototype]] ---> null

function f(){
  return 2;
}
// Functions inherit Function.prototype
// (which has methods like call, bind, etc.)
// f ---> Function.[[Prototype]] ---> Object.[[Prototype]] ---> null
```

CREATING NEW OBJECTS
OBJECT.CREATE()

- Objects can also be created using the `Object.create()` method which allows to **choose the prototype object**.

```
var a = {a: 1};
// a ---> Object.[[Prototype]] ---> null

var b = Object.create(a);
// b ---> a ---> Object.[[Prototype]] ---> null
console.log(b.a); // 1 (inherited)

var c = Object.create(b);
// c ---> b ---> a ---> Object.[[Prototype]] ---> null

var d = Object.create(null);
// d ---> null
```

CREATING NEW OBJECTS OBJECT.CREATE()

- Objects can also be created using the `Object.create()` method which allows to **choose the prototype object**.

```
var Animal = {
  type: "Invertebrates", // Default value of properties
  displayType : function(){  // Method which display the type
    console.log(this.type);
  }
}

var animal1 = Object.create(Animal);
animal1.displayType(); // Output:Invertebrates

var fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Output:Fishes
```

CREATING NEW OBJECTS
CONSTRUCTORS

- A **constructor function** (short: **constructor**) helps with producing objects that are similar in some way (have a particular initial set of properties and values).

- A constructor is a **function** that is invoked via the **new operator** — constructors are thus a rough **analog to classes** in other languages.

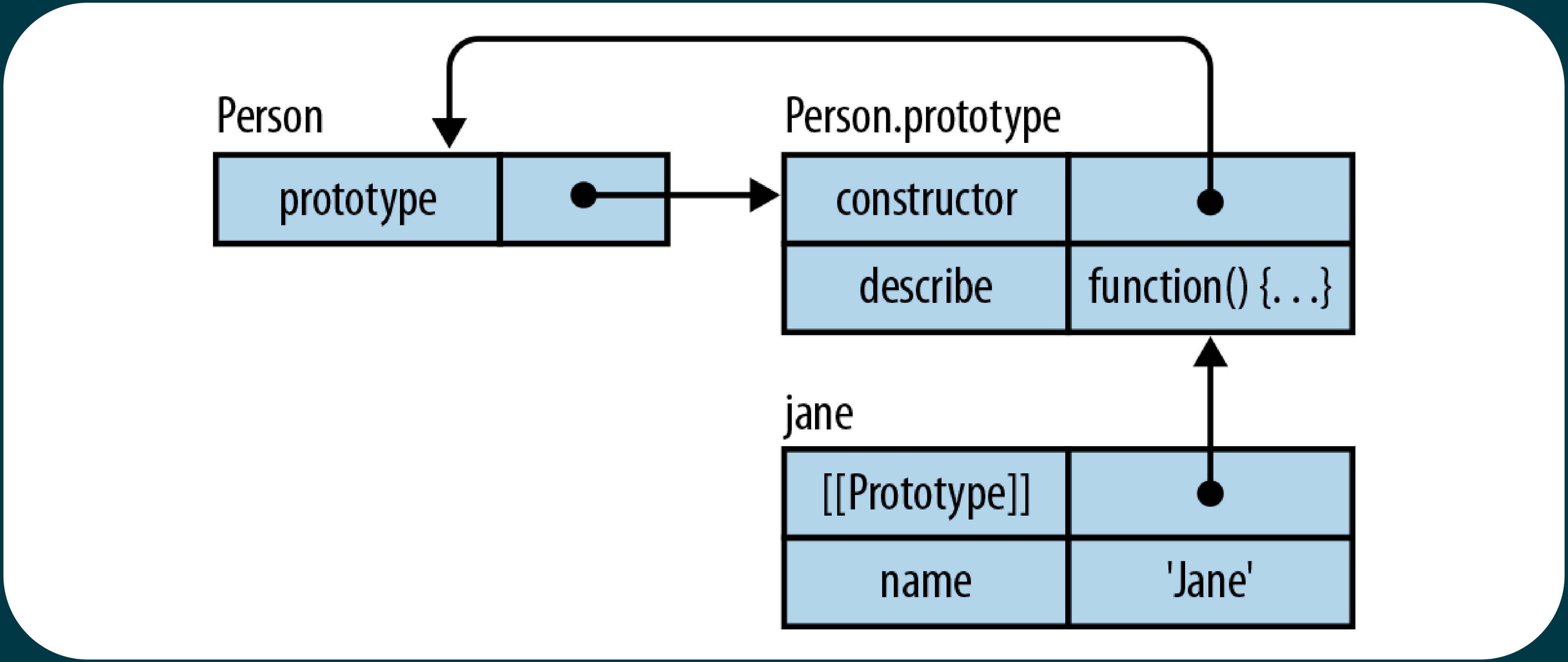
- By **convention**, the names of constructors start with **uppercase** letters, while the names of methods start with lowercase letters.

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

var person1 = new Person("Alice");
var person2 = new Person("Bob");
```

CREATING NEW OBJECTS CONSTRUCTORS

- The **prototype of the new object** is set to the `prototype` of the **function object** that was invoked as the constructor
- `Person.[[Prototype]]` should not be confused with `Person.prototype`: The former specifies the prototype of the **function** Person itself and the latter the prototype of all **instances** created using function Person



CREATING NEW OBJECTS
CONSTRUCTORS

- Methods are defined by assigning functions to a named property of the constructor function's `prototype` property.

```
var Person = function (firstName) {
  this.firstName = firstName;
};

Person.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");

// call the Person sayHello method.
person1.sayHello(); // logs "Hello, I'm Alice"
person2.sayHello(); // logs "Hello, I'm Bob"
```

CREATING NEW OBJECTS
CONSTRUCTORS

```
function Student(firstName, subject) {
  Person.call(this, firstName);
  this.subject = subject;
};

// Create a Student.prototype object that inherits from Person.prototype.
// Note: A common error here is to use "new Person()" to create the
// Student.prototype. That's incorrect for several reasons, not least
// that we don't have anything to give Person for the "firstName" argument.
// The correct place to call Person is above, where we call it from Student.
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName + ". I'm studying "
              + this.subject + ".");
};

Student.prototype.sayGoodBye = function(){
  console.log("Goodbye!");
};

// Example usage:
var student1 = new Student("Janet", "Applied Physics");
student1.sayHello();   // "Hello, I'm Janet. I'm studying Applied Physics."
student1.sayGoodBye(); // "Goodbye!"
```

CREATING NEW OBJECTS
CONSTRUCTORS

```
function Student(firstName, subject) {
  Person.call(this, firstName);
  this.subject = subject;
};

// Create a Student.prototype object that inherits from Person.prototype.
// Note: A common error here is to use "new Person()" to create the
// Student.prototype. That's incorrect for several reasons, not least
// that we don't have anything to give Person for the "firstName" argument.
// The correct place to call Person is above, where we call it from Student.
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName + ". I'm studying "
              + this.subject + ".");
};

Student.prototype.sayGoodBye = function(){
  console.log("Goodbye!");
};

// Example usage:
var student1 = new Student("Janet", "Applied Physics");
student1.sayHello();   // "Hello, I'm Janet. I'm studying Applied Physics."
student1.sayGoodBye(); // "Goodbye!"
```

Why setting constructor explicitly?

http://stackoverflow.com/questions/8453887/why-is-it-necessary-to-set-the-prototype-constructor

PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- Basic JavaScript
- *JavaScript: The Good Parts*
- ***JavaScript: The Bad Parts***
- Languages that compile to JavaScript
- ECMAScript 6

BAD PARTS
GLOBAL VARIABLES

- A global variable is a variable that is visible in **every scope**.
- JavaScript does not have a linker. All compilation units are loaded into a common global object.
- Because global variables **can be changed by any part of the program at any time**, they can significantly complicate the behavior of the program.
- Use of global variables degrades the reliability of the programs that use them.

BAD PARTS
GLOBAL VARIABLES

- There are three ways to define global variables:
  - place a **`var` statement outside** of any function,
  - add a property directly to the **global object**; the global object is the container of all global variables, e.g. `window` in browsers,
  - **implied global** — use a variable without declaring it.
- Implied globals were intended as a **convenience** to beginners by making it **unnecessary to declare variables**.
- Unfortunately, forgetting to declare a variable is a very common mistake. JavaScript's policy of **making forgotten variables global** creates bugs that can be very difficult to find.

BAD PARTS
SCOPE

- Most mainstream languages are **block-scoped** — variables declared in a **block** are not visible outside of the block, e.g. in Java:

```
public static void main(String[] args) {
    { // block starts
        int foo = 4;
    } // block ends
    System.out.println(foo); // Error: cannot find symbol
}
```

- JavaScript uses the **block syntax**, but variables are **function-scoped**: only functions introduce new scopes; blocks are ignored when it comes to scoping.

```
var foo = "1";
(function () {
  if (!foo) { var foo = "2"; }
  console.log(foo); // ???
}());
```

INTRODUCING A NEW SCOPE THE IIFE PATTERN

- Sometimes you want to introduce a **new variable scope** — for example, to prevent a variable from becoming global.

- In JavaScript (ES5), you can’t use a block to do so; you must use a function. But there is a **pattern** for using a function in a block-like manner — it is called **IIFE** (**immediately invoked function expression**):

```
(function () { // open IIFE
    var tmp = ...; // not a global variable
}()); // close IIFE
```

- An IIFE is a **function expression** that is called **immediately** after you define it. Inside the function, **a new scope** exists, preventing `tmp` from becoming global.

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"        // false
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"          // false
0           == ""           // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"      // false
0           == ""       // true
0           == "0"      // true
false       == "false"  // false
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
false       == "false"   // false
false       == "0"       // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"        // false
0           == ""         // true
0           == "0"        // true
false       == "false"    // false
false       == "0"        // true
false       == 0          // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
false       == "false"   // false
false       == "0"       // true
false       == 0         // true
false       == undefined // false
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (**!**=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
false       == "false"   // false
false       == "0"       // true
false       == 0         // true
false       == undefined // false
false       == null      // false
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""         == "0"       // false
0          == ""        // true
0          == "0"       // true
false      == "false"   // false
false      == "0"       // true
false      == 0         // true
false      == undefined // false
false      == null      // false
null       == undefined // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""         == "0"       // false
0          == ""        // true
0          == "0"       // true
false      == "false"   // false
false      == "0"       // true
false      == 0         // true
false      == undefined // false
false      == null      // false
null       == undefined // true
" \t\r\n"  == 0         // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
false       == "false"   // false
false       == "0"       // true
false       == 0         // true
false       == undefined // false
false       == null      // false
null        == undefined // true
" \t\r\n"   == 0         // true
" \t\r\n"   == false     // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""          == "0"       // false
0           == ""        // true
0           == "0"       // true
false       == "false"   // false
false       == "0"       // true
false       == 0         // true
false       == undefined // false
false       == null      // false
null        == undefined // true
" \t\r\n"   == 0         // true
" \t\r\n"   == false     // true
```

but...

```
" \t\r\n"          ? true : false // true
" \t\r\n" == true  ? true : false // false
" \t\r\n" == false ? true : false // true

""          ? true : false // false
"" == true  ? true : false // false
"" == false ? true : false // true
```

BAD PARTS
EQUALITY AND COMPARISONS

- JavaScript has two ways of checking if two values are equal.
- **Normal** (or **lenient**) equality (==) and inequality (!=) tries to convert values of different types before comparing them.

```
""        == "0"       // false
0         == ""        // true
0         == "0"       // true
false     == "false"   // false
false     == "0"       // true
false     == 0         // true
false     == undefined // false
false     == null      // false
null      == undefined // true
" \t\r\n" == 0         // true
" \t\r\n" == false     // true
```

but…

```
" \t\r\n"          ? true : false // true
" \t\r\n" == true  ? true : false // false
" \t\r\n" == false ? true : false // true

""          ? true : false // false
"" == true  ? true : false // false
"" == false ? true : false // true
```

- The above table shows the results of the **type coercion** which can **introduce bugs** due to its **complicated conversion rules**.

BAD PARTS
EQUALITY AND COMPARISONS

- **Strict** equality (===) and strict inequality (!==) consider only values that have the same type to be equal.

```
""             ===   "0"            // false
0              ===   ""             // false
0              ===   "0"            // false
false          ===   "false"        // false
false          ===   "0"            // false
false          ===   undefined      // false
false          ===   null           // false
null           ===   undefined      // false
" \t\r\n"      ===   0              // false
```

- The above results are a lot clearer — **always use strict equality and avoid lenient equality.**

BAD PARTS
RESERVED KEYWORDS

- The following words are **reserved** in JavaScript:
  `abstract boolean break byte case catch char class const continue debugger default delete do double else enum export extends false final finally float for function goto if implements import in instanceof int interface long native new null package private protected public return short static super switch synchronized this throw throws transient true try typeof var volatile void while with`
- Most of these words are **not used** in the language.
- They cannot be used in variable and function names. The reserved words can be used as keys in object literals:

```
object = {case: value}; // ok
object = {'case': value}; // ok
object.case = value; // ok
object['case'] = value; // ok
var case = value; // illegal
function case() {}; // illegal
```

BAD PARTS
+ OPERATOR

- The **+** operator can **add** or **concatenate**:
  - If either operand is an **empty** string, it produces the other operand **converted** to a string.
  - If both operands are **numbers** or **booleans**, it produces the **sum**.
  - Otherwise, it **converts** both operands to **strings** and **concatenates** them.
- This complicated behavior is a common source of bugs.
- If you intend **+** to add, make sure that both operands are numbers.

BAD PARTS
FALSY VALUES

- JavaScript has a surprisingly large set of falsy values:
  - `undefined, null`
  - Boolean: `false`
  - Number: `0, NaN`
  - String: `''`
- These values are all falsy, but they are **not interchangeable**:

  ```
  value = myObject[name];
  if (value === null) {
      alert(name + ' not found.');
  }
  ```

  `undefined` is the value of missing members, but the snippet is testing for null.
- Also, `undefined` and NaN are **not constants**. They are **global variables**, and **in legacy browsers** you can change their values, even that ES5 specifies that they are non-writable.

BAD PARTS
WRAPPER OBJECTS

- The three primitive types `boolean`, `number`, and `string` have corresponding constructors: `Boolean`, `Number`, `String`.
- Their instances (so-called **wrapper objects**) contain (wrap) primitive values; for example: `new Boolean(false)`
- This turns out to be completely unnecessary and confusing.
- Don't use `new Boolean` or `new Number` or `new String`.
- Also avoid `new Object` and `new Array`. Use `{}` and `[]` instead — **prefer literals over constructors**.

BAD PARTS
AUTOMATIC SEMICOLON INSERTION

- ES5 defines rules governing automatic semicolon insertion
- Missing semicolon is automatically placed before line terminator, closing brace, end of stream
- Thus, white space can change meaning:

```
function plus1(a, b) {
  return a + b
}

function plus2(a, b) {
  return
  a + b
}

console.log(plus1(1, 2)); // 3
console.log(plus2(1, 2)); // undefined
```

STRICT MODE – A WAY TO HANDLE BAD PARTS

- **Strict mode** is a special mode of script execution, where semantics of *some* JavaScript constructs and functions changes
- To invoke strict mode for current scope (function or entire script) put `"use strict";` before any other code in this scope

MOTIVATION FOR STRICT MODE

- JavaScript was designed to be easy for novice developers, and it gives operations which should be errors non-error semantics. Sometimes this fixes the immediate problem, but sometimes this creates worse problems in the future. Strict mode treats these mistakes as errors so that they're discovered and promptly fixed.

CHANGES IN STRICT MODE

- Differences between modes

Functionality	Sloppy mode	Strict mode
Use of an undeclared variable	The variable becomes global	ReferenceError: assignment to undeclared variable x
Assignment to a read-only property	Silently ignored	TypeError: "undefined" is read-only
Deletion of undeletable properties	Silently ignored	TypeError: property "prototype" is non-configurable and can't be deleted
Duplicated names of function arguments	Last argument hides previous duplicates	SyntaxError: duplicate formal argument x
Octal syntax (e.g., 015)	Leading 0 changes number interpretation to octal	SyntaxError: octal literals and octal escape sequences are deprecated
Setting properties on primitives	Silently ignored	TypeError: can't assign to properties of (new Boolean(false)): not an object
New variables introduced by `eval`	Included in scope, where `eval` is executed	Limited to scope of eval arguments
Deleting of variables	Silently ignored	SyntaxError: applying the 'delete' operator to an unqualified name is deprecated
Names of variables: `eval` and `arguments`	Allowed, hides language-variables	SyntaxError: can't assign to eval in strict mode

- And many more:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

CHANGES IN STRICT MODE SIMPLIFIES USE OF VARIABLE

- Many compiler optimizations rely on the ability to say that variable *X* is stored in *that* location: this is critical to fully optimize JavaScript code.

- JavaScript sometimes makes this basic mapping of name to variable definition in the code impossible to perform until runtime. Strict mode removes most cases where this happens, so the compiler can better optimize strict mode code.

PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- Basic JavaScript
- *JavaScript: The Good Parts*
- *JavaScript: The Bad Parts*
- **Languages that compile to JavaScript**
- ECMAScript 6

COFFEESCRIPT

- CoffeeScript is a little language that **transcompiles** to JavaScript (http://coffeescript.org):
  - the code compiles one-to-one into the equivalent JavaScript.
  - the code can be integrated seamlessly with any JavaScript library.
  - the compiled output will work in every JavaScript runtime.
- CoffeeScript adds **syntactic sugar** inspired by **Ruby**, **Python** and **Haskell** to enhance JavaScript's brevity and readability
- CoffeeScript is one of the first attempts to **expose the good parts of JavaScript** in a simple way.

TYPESCRIPT

TypeScript

- TypeScript is a **free** and **open source** programming language (developed by **Microsoft**) which **transcompiles** to JavaScript.
- It is a **strict superset** of **JavaScript** — any existing JavaScript programs are also valid TypeScript programs.
- TypeScript adds features not normally available in JavaScript:
  - optional **static typing**
  - **class-based object-oriented programming**
  - **arrow syntax** for anonymous functions
- TypeScript may be used to develop JavaScript applications for **client-side** or **server-side** execution.

TYPESCRIPT TYPE ANNOTATIONS

- TypeScript provides **static typing** through **type annotations** to enable **type checking at compile time**. This is optional and can be ignored to use the regular **dynamic typing** of JavaScript.

```
function add(left: number, right: number): number {
   return left + right;
}
```

- The TypeScript compiler makes use of **type inference** to infer types when types are not given.

- For example, the `add` method in the code above would be inferred as returning a `number` even if no return type annotation had been provided. If no type can be inferred because of lack of declarations then it will default to the **dynamic `any` type**.

TYPESCRIPT CLASS-BASED OOP

TypeScript

- Traditional JavaScript focuses on **functions** and **prototype-based inheritance** as the basic means of building up reusable components
- Programmers are often more comfortable with a **class-based** object-oriented approach, where **classes inherit functionality** and objects are built from these classes.
- Starting with **ECMAScript 6**, JavaScript programmers are able to build their applications using classes.
- TypeScript allows to use these techniques too, and compile them down to ES5 that **works across all major browsers** and platforms.

CLASSES IN ECMASCRIPT 6

```
class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

class Square extends Polygon {
  constructor(sideLength) {
    super(sideLength, sideLength);
  }
  get area() {
    return this.height * this.width;
  }
  set sideLength(newLength) {
    this.height = newLength;
    this.width = newLength;
  }
}

var square = new Square(2);
square.sideLength = 3
var a = square.area // 9
```

CLASSES IN TYPESCRIPT

```
class Animal {
    private name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        alert("Slithering...");
        super.move(meters);
    }
}
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 45) {
        alert("Galloping...");
        super.move(meters);
    }
}

var sam = new Snake("Sammy the Python");
var tom: Animal = new Horse("Tommy the Palomino");
sam.move();
tom.move(34);
```

DART

- Dart is an **open-source**, **class-based**, **object-oriented**, **statically typed** programming language developed by **Google** for building structured HTML5 web apps (https://www.dartlang.org)

- Dart provides not only a new language, but **libraries**, an **editor**, a **virtual machine (VM)**, a **browser** that can run Dart apps natively, and a **compiler to JavaScript.**

- Dart aims to help in two main ways:
  - **Better performance** — a more structured language is easier to optimize, and a fresh VM enables improvements such as faster startup.
  - **Better productivity** — **libraries** and **packages** helps you work with other developers and easily reuse code from other projects; **types** can make APIs clearer and easier to use; **tools** help you refactor, navigate, and debug code.

DART EXAMPLE

Dart

```
// Create a class for Point.
class Point {

  // Final variables cannot be changed once they are assigned.
  // Create two instance variables.
  final num x, y;

  // A constructor, with syntactic sugar for setting instance variables.
  Point(this.x, this.y);

  // A named constructor with an initializer list.
  Point.origin()
      : x = 0,
        y = 0;

  // A method.
  num distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return math.sqrt(dx * dx + dy * dy);
  }

  // Example of Operator Overloading
  Point operator +(Point other) => new Point(x + other.x, y + other.y);
}
```

DART USAGE

- **Compiled as JavaScript** — the Dart code pre-compiled to JavaScript using the **dart2js compiler** is compatible with **all major browsers**. Through optimization of the compiled output, code written in Dart can **run faster** than equivalent code hand-written using JavaScript idioms.

- **Dartium Browser** — the Dart SDK ships with a version of the **Chromium web browser** modified to include a Dart virtual machine. This browser can run Dart code directly **without compilation** to JavaScript. It is intended as a **development tool** for Dart applications, rather than as a general purpose web browser.

- **Stand-alone** — the Dart SDK ships with a **stand-alone Dart VM**, allowing dart code to run in a **command-line environment**. Dart ships with a complete **standard library** allowing users to write fully functional system apps, such as custom web servers.

WHY DART?

- **Dart is easy to learn — *Dart is new, yet familiar***. It's an object-oriented language with classes, single inheritance, lexical scope, top-level functions, and a familiar syntax (https://www.dartlang.org).

- **Dart compiles to JavaScript**. Dart has been designed from the start to compile to JavaScript, so that Dart apps can run across the entire modern web. Every feature considered for the language must somehow be translated to **performant JavaScript** before it is added.

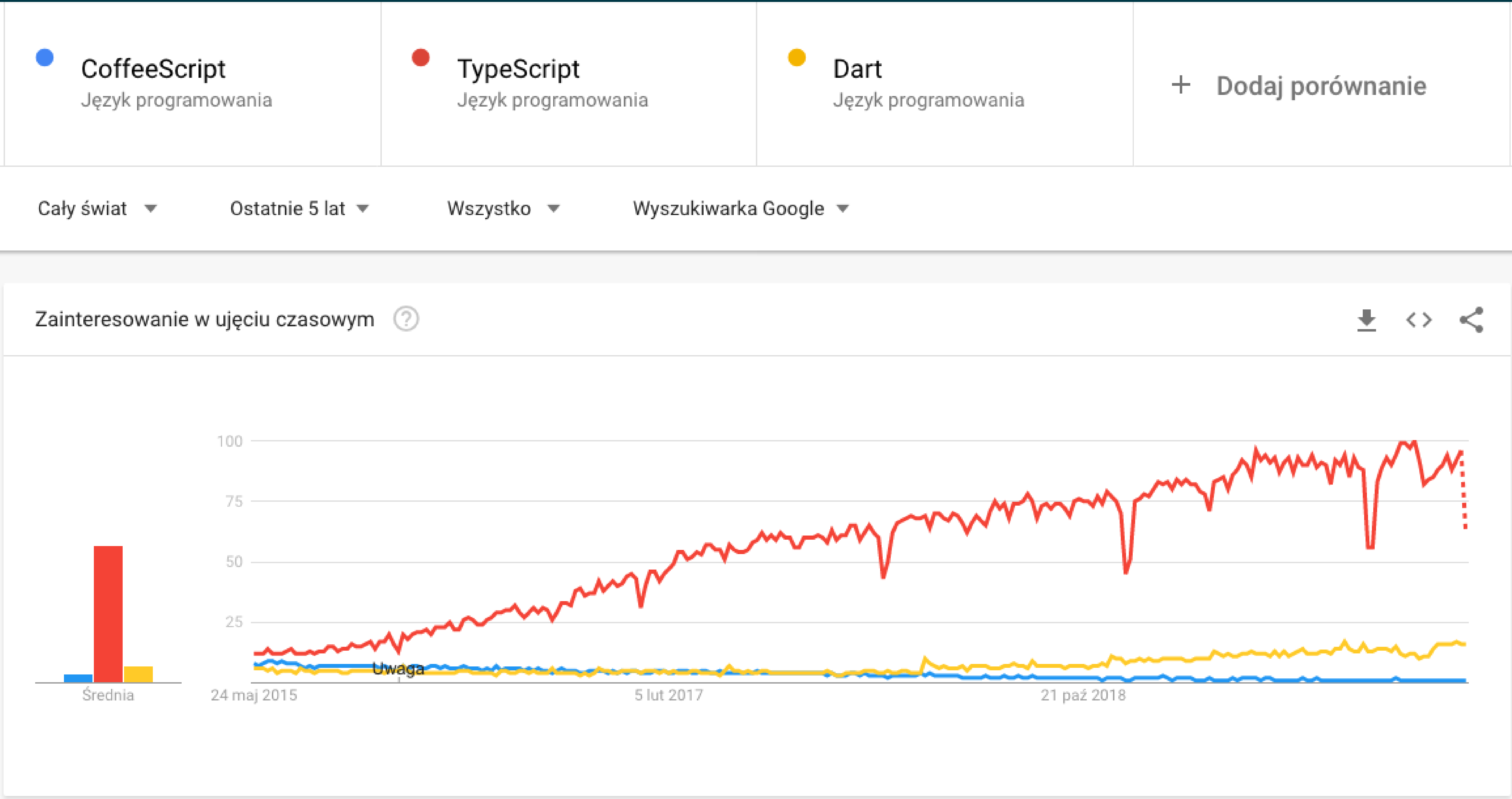
- **Dart runs in the client and on the server**. The Dart VM can be integrated into a browser, but it can also run standalone on the command line.

- **Dart comes with a lightweight editor.** You can use Dart Editor to write, launch, and debug Dart apps. The editor can help you with code completion, detecting potential bugs, debugging both command-line and web apps, and even refactoring.

WHY DART?

- **Dart has a wide array of built-in libraries** (https://www.dartlang.org)
  - The core library supports fundamental features such as collections, dates, and regular expressions.
  - Web apps can use the HTML library — think DOM programming, but optimized for Dart.
  - Command-line apps can use the I/O library to work with files, directories, sockets, and servers.
- **Dart supports safe, simple concurrency with isolates**. Traditional shared-memory threads are difficult to debug and can lead to deadlocks. Dart's **isolates**, inspired by **Erlang**, provide an easier to understand model for running isolated, but concurrent, portions of your code.
- **Dart supports types, without requiring them**. Dart's optional types are static type annotations that act as documentation, clearly expressing your intent. Using types means that fewer comments are required to document the code, and tools can give better warnings and error messages.

LANGUAGE INTEREST OVER TIME


● CoffeeScript
Język programowania

● TypeScript
Język programowania

● Dart
Język programowania

+ Dodaj porównanie

Cały świat ▾ Ostatnie 5 lat ▾ Wszystko ▾ Wyszukiwarka Google ▾

Zainteresowanie w ujęciu czasowym

PRESENTATION OUTLINE

- What is JavaScript?
- Historical Perspective
- Basic JavaScript
- *JavaScript: The Good Parts*
- *JavaScript: The Bad Parts*
- Languages that compile to JavaScript
- **ECMAScript 6**

ECMASCRIPT 6 (ES6/ES2015)

- ECMAScript 6 is the version of the standard released in **June 2015**.
- **ES2015** is a **significant update** to the language, and the first update to the language since ES5 was standardized in **2009**.
- ES2015 introduces some new JavaScipt language features **similar to CoffeeScript and TypeScript**.
- Implementation of these features in major JavaScript engines is almost done: http://kangax.github.io/compat-table/es6/
- **Transpilers** like **Babel.js** or **Traceur** allow to use features **from the future** (ES6 and other experimental ones) **today**.

NEW FEATURES IN ES6
BLOCK SCOPING WITH LET

- The `let` keyword allows to declare variables that are **limited in scope to the block, statement, or expression** on which it is used. This is unlike the `var` keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

```
function varTest() {
  var x = 31;
  if (true) {
    var x = 71;  // same variable!
    console.log(x);  // 71
  }
  console.log(x);  // 71
}
function letTest() {
  let x = 31;
  if (true) {
    let x = 71;  // different variable
    console.log(x);  // 71
  }
  console.log(x);  // 31
}
```

NEW FEATURES IN ES6
CONSTANT DEFINITIONS

- **Constant definitions** are possible with `const`. `let` and `const` behave similarly in the sense that both are **block scoped**.

```
// define my_fav as a constant and give it the value 7
const my_fav = 7;

// trying to redeclare a constant throws an error
const my_fav = 20;

// the name is reserved for constant above, so this will also fail
var my_fav = 20;

// const requires an initializer
const foo; // SyntaxError: missing = in const declaration

// const also works on objects
const myObject = {"key": "value"};

// However, object attributes are not protected,
// so the following statement is executed without problems
myObject.key = "otherValue";
```

NEW FEATURES IN ES6
DEFAULT FUNCTION PARAMETERS

- Default function parameters allow formal parameters to be **initialized** with **default values if no value or undefined** is passed.

- In JavaScript, parameters of functions **default to undefined**.

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;
  // or below; but be aware how type coercion works here
  b = b || 1;

  return a*b;
}
multiply(5); // 5
```

- The syntax for default parameters in ES6 is extremely intuitive. The default parameters are defined when the functions are defined.

```
function multiply(a, b = 1) {
  return a*b;
}
multiply(5); // 5
```

NEW FEATURES IN ES6
EFFICIENT DATA STRUCTURES

- ES6 offers **new data structures** previously not available in JavaScript — **sets** and **maps**.

- Sets allow you to easily create a collection of **unique** values (objects or primitives) without worrying about **type coercion**.

```
var items = new Set();
items.add(5);
items.add("5");
items.add(5);     // oops, duplicate - this is ignored

console.log(items.size);      // 2
console.log(items.has(5));    // true
console.log(items.has(6));    // false

items.delete(5)
console.log(items.has(5));    // false
```

NEW FEATURES IN ES6
EFFICIENT DATA STRUCTURES

- In JavaScript, developers have traditionally used **regular objects** as maps — JSON is based on the premise that objects represent **key-value pairs**.

```
var map = {};

if (!map[key]) {
    map[key] = value;
}
```

- In ES6, the **key can be any JavaScript value** (even an object) and **not just a string**.

```
var map = new Map();
map.set("name", "Nicholas");
map.set(document.getElementById("my-div"), { flagged: false });

var name = map.get("name"),
    meta = map.get(document.getElementById("my-div"));
```

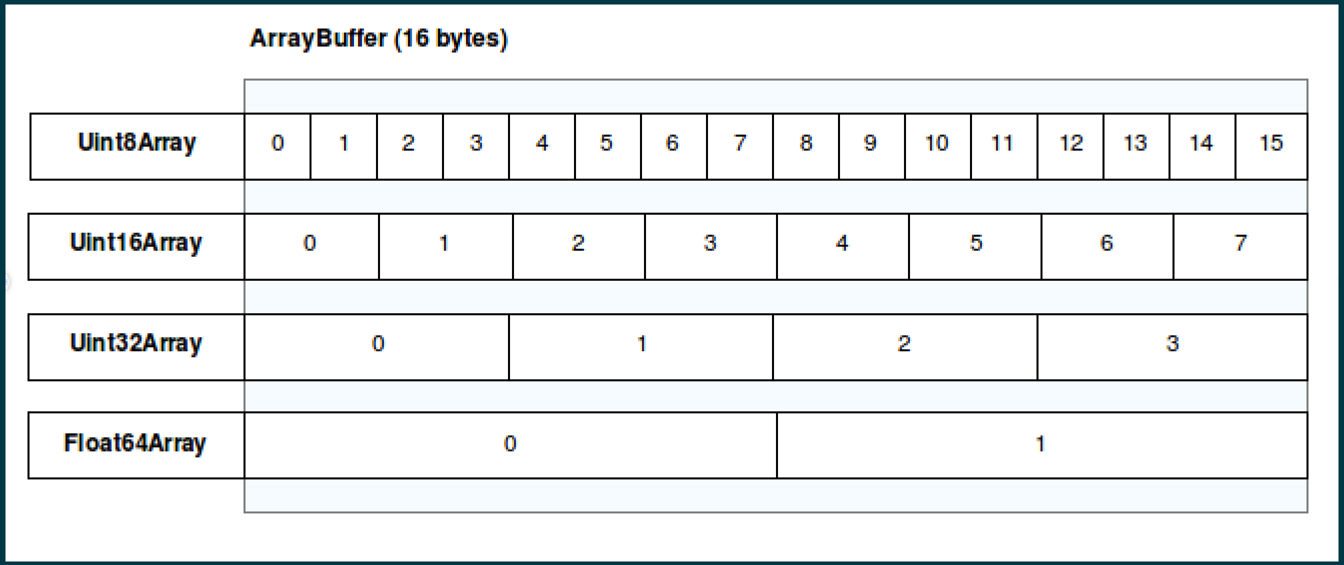
(STRONGLY) TYPED ARRAYS (ES6)

- Typed array is an object intended to store fixed amount of raw binary data
  - This acts more or less as an ordinary array in most programming languages:
  - No dynamic expanding or shrinking
  - Index-based access
- Note that the typed array is not an array in the sense of JavaScript, i.e.: `Array.isArray()` called on the typed array yields `false`

(STRONGLY) TYPED ARRAYS: BUFFERS AND VIEWS

- `ArrayBuffer` is an object that represents a continuous region of memory to store data but it gives no access to the data

- View is a mechanism that puts `ArrayBuffer` into a context that allow reading or writing data to the buffer
  - E.g., multiple views enable us to interpret the same data differently:



(STRONGLY) TYPED ARRAYS EXAMPLE

```
// Declare a buffer of 16 bytes
var buffer = new ArrayBuffer(16);

// Create an int32 view on the buffer
var int32View = new Int32Array(buffer);

// Write data to the buffer as int32
for (var i = 0; i < int32View.length; i++) {
    int32View[i] = i * 2;
}

// Create an int16 view on the buffer
var int16View = new Int16Array(buffer);

// Read data from the buffer as int16
for (var i = 0; i < int16View.length; i++) {
    console.log("Entry " + i + ": " + int16View[i]);
}

/* Output:
Entry 0: 0
Entry 1: 0
Entry 2: 2
Entry 3: 0
Entry 4: 4
Entry 5: 0
Entry 6: 6
Entry 7: 0
*/
```

NEW FEATURES IN ES6
FOR-OF LOOP

- The `for...of` statement creates a **loop** Iterating over **iterable objects** (including **Array**, **Map** and **Set**), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
let arr = [3, 5, 7];
arr.foo = "hello";

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}
for (let i of arr) {
   console.log(i); // logs "3", "5", "7"
}

var es6 = new Map();
es6.set("edition", 6);
es6.set("committee", "TC39");
es6.set("standard", "ECMA-262");
for (var [name, value] of es6) {
  console.log(name + ": " + value);
}
```

NEW FEATURES IN ES6
ARROW FUNCTIONS

- Arrow functions are function shorthand using the **=> syntax**. They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript.

```
(param1, param2, paramN) => { statements }
(param1, param2, paramN) => expression
// equivalent to:  => { return expression; }

// Parentheses around the single argument are not required:
singleParam => { statements }
singleParam => expression

// Function with no arguments requires parentheses:
() => { statements }
```

```
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});
```

NEW FEATURES IN ES6
ARROW FUNCTIONS

- Unlike regular functions, arrow functions share the same **lexical `this`** as their surrounding code.

- The **binding of `this`** inside of functions is one of the most common **source of errors** in JavaScript. Since the **value of `this`** can change inside of a single function **depending on the context** in which it's called, it's possible to mistakenly affect one object when you meant to affect another.

```
var PageHandler = {
    init: function() {
        document.addEventListener("click", function(event) {
            this.doSomething(event.type);
        }, false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

NEW FEATURES IN ES6
ARROW FUNCTIONS

- Unlike regular functions, arrow functions share the same **lexical `this`** as their surrounding code.

- The **binding of `this`** inside of functions is one of the most common **source of errors** in JavaScript. Since the **value of `this`** can change inside of a single function **depending on the context** in which it's called, it's possible to mistakenly affect one object when you meant to affect another.

```
var PageHandler = {
    init: function() {
        document.addEventListener("click", (function(event) {
            this.doSomething(event.type);
        }).bind(this), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

NEW FEATURES IN ES6
ARROW FUNCTIONS

- Unlike regular functions, arrow functions share the same **lexical `this`** as their surrounding code.
- The **binding of `this`** inside of functions is one of the most common **source of errors** in JavaScript. Since the **value of `this`** can change inside of a single function **depending on the context** in which it's called, it's possible to mistakenly affect one object when you meant to affect another.

```
var PageHandler = {
    init: function() {
        document.addEventListener("click",
                event => this.doSomething(event.type), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

NEW FEATURES IN ES6
ARRAY COMPREHENSION

- The array comprehension syntax is a JavaScript expression which allows you to quickly **assemble a new** array based on **an existing one**.

```
[for (x of iterable) x]
[for (x of iterable) if (condition) x]
[for (x of iterable) for (y of iterable) x + y]

var abc = [ "A", "B", "C" ];
[for (letters of abc) letters.toLowerCase()]; // [ "a", "b", "c" ]

var years = [ 1954, 1974, 1990, 2006, 2010, 2014 ];
[for (year of years) if (year > 2000) year]; // [ 2006, 2010, 2014 ]

var numbers = [ 1, 2, 3 ];
var letters = [ "a", "b", "c" ];
var cross = [for (i of numbers) for (j of letters) i+j];
// [ "1a", "1b", "1c", "2a", "2b", "2c", "3a", "3b", "3c" ]
```

CONCLUSIONS

- It would be nice if programming languages were designed to have **only good parts**.
- The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy.
- With the combination of **prototypal inheritance**, **dynamic object extension**, and **closures**, JavaScript has one of the **most flexible** and **expressive object systems** available in any programing language.
- The age of JavaScript is truly upon us — not only has it become completely **ubiquitous on the client side**, but its use as a **server-side** language has finally taken off too, thanks to **Node.js**.

CONCLUSIONS

- *There is an important tradeoff between the computational power of a language and the ability to determine what a program in that language is doing.*

- *Computer Science spent the last forty years making languages which were as powerful as possible. Nowadays we have to appreciate the reasons for picking* ***not the most powerful solution but the least powerful****. Expressing constraints, relationships and processing instructions in less powerful languages increases the flexibility with which information can be reused: the less powerful the language, the more you can do with the data stored in that language.*

— THE RULE OF LEAST POWER, TIM BERNERS-LEE, 2006

CONCLUSIONS

- *There is an important tradeoff between the computational power of a language and the ability to determine what a program in that language is doing.*

- *Computer Science spent the last forty years making languages which were as powerful as possible. Nowadays we have to appreciate the reasons for picking* ***not the most powerful solution but the least powerful****. Expressing constraints, relationships and processing instructions in less powerful languages increases the flexibility with which information can be reused: the less powerful the language, the more you can do with the data stored in that language.*

  — THE RULE OF LEAST POWER, TIM BERNERS-LEE, 2006

- **The Rule of Least Power**: *Use the least powerful language suitable for expressing information, constraints or programs on the World Wide Web. Powerful languages inhibit information reuse.*

CONCLUSIONS

- *There is an important tradeoff between the computational power of a language and the ability to determine what a program in that language is doing.*

- *Computer Science spent the last forty years making languages which were as powerful as possible. Nowadays we have to appreciate the reasons for picking* ***not the most powerful solution but the least powerful****. Expressing constraints, relationships and processing instructions in less powerful languages increases the flexibility with which information can be reused: the less powerful the language, the more you can do with the data stored in that language.*

  — THE RULE OF LEAST POWER, TIM BERNERS-LEE, 2006

- **The Rule of Least Power**: *Use the least powerful language suitable for expressing information, constraints or programs on the World Wide Web. Powerful languages inhibit information reuse.*

- **Atwood's Law**: *any application that can be written in JavaScript, will eventually be written in JavaScript.*

REFERENCES

- *Programming JavaScript Applications* — Eric Elliott, O'Reilly Media, Inc., 2014, available online at: http://chimera.labs.oreilly.com/books/1234000000262
- *Speaking JavaScript* — Axel Rauschmayer, O'Reilly Media, Inc., 2014 http://speakingjs.com
- *JavaScript: The Good Parts* — Douglas Crockford, O'Reilly Media Inc., 2008
- *JavaScript Guide at Mozilla Developer Network* *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide*
- *Expressions at Mozilla Developer Network* *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions*
- http://jsbooks.revolunet.com
- http://javascript.crockford.com
- JavaScript typed arrays, Mozilla Developer Network https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays
- Jake Archibald – JavaScript promises: an Introduction https://developers.google.com/web/fundamentals/primers/promises