Javascript

André Restivo

Index

Introduction Variables Control Structures Functions Objects

Arrays Exceptions DOM

Ajax Advanced Functions Advanced Arrays Timers jQuery

Introduction

Javascript

- *Javascript* is a prototype-based, dynamic, object-oriented, imperative and functional language.
- In *Javascript*, functions are considered first-class citizens.
- Most commonly used as part of web browsers as a client-side scripting language.

History

- Originally developed by **Brendan Eich** at **Netscape**.
- Developed under the name **Mocha** but later named **LiveScript**.
- Changed name from LiveScript to **JavaScript**, in **1995**, at the time Netscape added support for Java.
- Microsoft introduced JavaScript support in Internet Explorer in August **1996** (called JScript).
- Submitted to **Ecma** International for consideration as an industry standard in 1996 (**ECMAScript**).
- Ecma International released the first version of the specification in **1997**.
- Nowadays JavaScript is a trademark of the **Oracle** Corporation.
- But JavaScript is officially managed by the **Mozilla** Foundation.
- *ECMAScript 6* or *ECMAScript 2015* introduced lots of new features.

Console

- Modern browsers all have a *Javascript* console that can be used to log messages from within web pages.
- It can also be used to inspect variables, evaluate expressions and just plain experimentation.
- The specifics of how it works vary from browser to browser, but there is a *defacto* set of features that are typically provided.
- The console.log(msg) function outputs a message to the console.
- Other debug level are possible like console.info(msg), console.warn(msg) and console.error(msg).
- Browsers allow filtering messages depending on their level.

Alert

The alert function opens a popup window with some text.

```
alert("Hello world!");
```

Strict Mode

ECMAScript 5 brought some big changes. To opt-in for those changes, scripts (or functions) must start with:

```
'use strict';
```

Some changes:

- No more global undeclared variables.
- No more declaring variables with **var**.
- Some warnings are now errors.

Resources

- Reference:
    - MDN Javascript Reference
    - EcmaScript Reference
    - MDN DOM Reference
- Resources:
    - MDN Javascript Resources
    - JS Fiddle
- Tutorials:
    - The Modern Javascript Tutorial
    - jQuery: Javascript 101
    - Javascript Style Guide

Variables

Variables

- JavaScript is a loosely typed or a dynamic language. That means you don't have to declare the type of a variable ahead of time.
- The type will get determined automatically while the program is being processed.
- Variables are declared using the **let** command.
- Variable names must contain only letters, digits, $ and _ (and not start with a digit).

```
let bar = 10;
bar = 'John Doe';
bar = true;
```

```
let foo = 10, bar;
bar = 'John Doe';
```

Constants

- Constants behave exactly the same way as variables except they can't be changed.
- Constants are declared using the const command.

```
const bar = 10;
bar = 20;       // TypeError: invalid assignment to const `bar'
```

Var

In older scripts you might find variables declared using **var** instead of **let**.

They have a different behavior than variables declared with **let**:

- They have no block scope (only function scope).
- Are processed when a function starts

```
if (true) {
  var bar = "1234";
  console.log(bar);     // 1234
}

console.log(bar);       // 1234
```

```
function foo() {
  bar = "1234";
  console.log(bar);     //1234
  var bar;
}
```

Not declaring variables

- It might seem that declaring variables in *Javascript* is *optional* but that is not the case.
- When you use a variable without declaring it, that variable will bubble up until if finds a variable declared with the same name.
- If it doesn't it attaches itself to the *window* or *global* object.
- This might have unforeseen and hard to debug consequences.

```
function foo() {
  bar = 1234;
}

let bar = 10;
foo();
console.log(bar);
```

Primitive Data Types

The standard defines the following data types:

- Number (double-precision 64-bit)
- String (textual data - single or double quoted)
- Boolean (true or false)
- Null (only one possible value: case sensitive null)
- Undefined (has not been assigned a value)

Strings

Strings can be defined equally using single or double quotes:

```
let firstname = 'John';
let lastname = "Doe";
```

We can also use *backticks*. With *backticks*, expressions inside *${…}* are evaluated and the result becomes a part of the string.

```
alert( `Hello, ${firstname} ${lastname}!` ); // Hello, John Doe!
alert( `The result is ${1 + 2}` );           // The result is 3
```

The + Operator

The plus (+) operator sums numbers, but if one of the operands is a string, it converts the other one into a string and concatenates the two:

```
console.log(11 + 31);   // 42
console.log("11" + 31); // "1131"
console.log(11 + "31"); // "1131"
```

Most of the time, operators and functions automatically convert a value to the right type (type conversion). You can still use the *String*, *Number* and *Boolean* functions to manually convert a value:

```
  let a = 0;
  let b = Boolean(a); // false
  let c = String(a);  // "0"
  let d = String(b);  // "false"
```

Comparison

When comparing values belonging to different types, they are converted to numbers:

Examples:

```
1 == "1";    // 1 == 1 -> true
0 == false;  // 0 == 0 -> true
"0" == true; // 0 == 1 -> true
"" == false; // 0 == 0 -> true
Boolean("0") == false; // 1 == 0 -> false
Boolean("0") == true;  // 1 == 1 -> true
```

Boolean Evaluation

The following values all evaluate to false:

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

All other values, including objects evaluate to true.

Be careful with the Boolean object:

```
let foo = new Boolean(false);
let bar = Boolean(false);
if (foo) // evaluates to true
if (bar) // evaluates to false
```

Strict Equality

- Strict equality compares two values for equality.
- Neither value is implicitly converted to some other value before being compared.
- If the values have different types, the values are considered unequal.

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

Comparing anything with **null** and **undefined** returns false. Comparisons between them have the following results:

```
  null === undefined; // false
  null == undefined;  // true
```

Control Structures

If … else

- Use the **if** statement to execute a statement if a logical condition is true.
- Use the optional **else** clause to execute a statement if the condition is false.

```
if (condition) {
  //do domething
} else {
  //something else
}
```

Switch

- A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label.
- If a match is found, the program executes the associated statement.

```
switch (expression) {
   case label_1:
      statements_1
      break;
   case label_2:
      statements_2
      break;
   //...
   default:
      statements_def
      break;
}
```

Loops

JavaScript supports the **for**, **do while**, and **while** loop statements:

```
for (let i = 0; i <= 10; i++) {
  console.log(i);
}// 0 1 2 3 4 5 6 7 8 9 10
```

```
let i = 0;
do {
   console.log(i);
   i++;
} while (i <= 10); // 0 1 2 3 4 5 6 7 8 9 10
```

```
let i = 0;
while (i <= 10) {
   console.log(i);
   i++;
} // 0 1 2 3 4 5 6 7 8 9 10
```

Break and continue

- The break statement finishes the current loop prematurely.
- The continue statement finishes the current iteration and continues with the next.

```
for (let i = 0; i < 10; i++) {
  if (i == 8) break;
  if (i % 2 == 0) continue;
  console.log(i);
} // 1 3 5 7
```

Functions

Defining functions

A function is defined using the **function** keyword.

```
function add(num1, num2) {
  console.log(num1 + num2);
}

add(1, 2);
```

- Primitive parameters are passed to functions by value.
- Non-primitive parameters (objects) are passed by reference.

Return

Functions can also return values.

```
function add(num1, num2) {
  return num1 + num2;
}

console.log(add(1, 2));
```

A function with an empty *return* or no *return* at all, returns **undefined**.

Default values

- If a parameter expected by a function is not passed, it becomes **undefined**.
- Unless we declare a default value for that parameter.
- Default values can be complex expressions and are only calculated when needed.

```
let count = 1;

function bar() {
  return count++;
}

function foo(var1, var2 = 1234, var3 = bar()) {
  console.log(var1);
  console.log(var2);
  console.log(var3);
}

foo(10, 20);    // 10 20 1
foo(10);        // 10 1234 2
foo();          // undefined 1234 3
```

Function Expressions

Another way to declare a function is the following:

```
let foo = function() {
  console.log('bar');
};  // don't forget the semicolon
```

This has the same effect as:

```
function foo() {
  console.log('bar');
}
```

Functions are just another datatype stored in variables. We can even copy them or display them in the console:

```
let bar = foo;
bar();
console.log(foo);
```

Functions as Parameters

Functions can be passed as parameters to other functions.

```
function foo(i) {
  console.log('bar = ' + i);
}

function executeNTimes(f, n) {
  for (let i = 0; i < n; i++)
    f(i);
}

executeNTimes(foo, 3);   // bar = 1 bar = 2 bar = 3
executeNTimes(foo(), 3); // this is a common mistake
```

Arrow Functions

A more compact way of declaring functions:

```
let foo = function(var1, var2) {
  return var1 + var2;
}
```

Is the same as:

```
let foo = (var1, var2) => var1 + var2;
```

Using the function from the previous slide:

```
executeNTimes((i) => console.log(i * i), 3);  // 0 1 4
```

Multi-line arrow functions are possible using a code-block **{...}**.

Objects

Objects

- JavaScript is designed on a simple **object-based** paradigm.
- An object is a collection of **properties**, and a property is an association between a name and a value.
- A property's value can be a function, in which case the property is known as a **method**.
- JavaScript is a **prototype-based** language and **does not** have a class statement (or does it?).

```
let person = { name: 'John Doe', age: 45 };
person.job = 'Driver';
console.log(person); // Object { name: "John Doe", age: 45, job: "Driver" }
```

Methods

- Methods are properties of an object that happen to be functions.
- Methods are defined the way normal functions are defined, except that they are assigned as the property of an object.
- You can use the **this** keyword within a method to refer to the current object.

```
let person = { name: "John Doe",
               age: 45,
               car: {make: "Honda", model: "Civic"},
               print: function() {
                 console.log(this.name + " is " + this.age + " years old!");
               }
             };
person.print(); // John Doe is 45 years old!
```

Assigning Methods

We can also assign a method to an object:

```
let person = { name: "John Doe",
               age: 45,
               car: {make: "Honda", model: "Civic"},
             };

person.print = function() {
  console.log(this.name + " is " + this.age + " years old!");
}

person.print(); // John Doe is 45 years old!
```

This

In *Javascript*, the this keyword (current context) behaves unlike in almost any other language.

- In the global execution context, this refers to the *global object* or *window*.
- Inside a function it depends on how the function was called.
    - Simple function call (undefined in strict mode).
    - Using *apply* or *call* (*this* is the first argument).
    - Object method (the object method was called from)
    - Arrow functions (retains the enclosing context)
    - Browser Events (the object that fired the event)

This in functions

Using **this** in simple functions:

```
function bar(var1, var2) {
  console.log(var1);
  console.log(var2);
  console.log(this);
}

bar(10, 20);                 // 10 20 undefined
bar.call('foo', 10, 20);     // 10 20 foo
bar.apply('foo', [10, 20]);  // 10 20 foo
```

- **Call** and **apply** are an alternative ways to call functions.
- Both receive the **context** as the **first** argument.
- The remaining parameters are sent as regular parameters in call and as an array in apply.

This in methods

Using **this** inside objects:

```
let foo = {
  bar() {
    console.log(this);
  }
}
foo.bar();          // Object { bar: bar() }
let bar = foo.bar;
bar();              // Undefined
bar.apply('foo');   // foo
```

This in arrow functions

Using **this** inside arrow functions:

```
let foo = {
  bar1: function() {
    return () => console.log(this);
  },

  bar2: function() {
    return function(){return console.log(this);}
  }
}

foo.bar1()();  // Object { bar1: bar1(), bar2: bar2() }
foo.bar2()();  // Undefined
```

Objects as arrays

- Properties of JavaScript objects can also be accessed or set using a bracket notation.
- Objects can be seen as associative arrays, since each property is associated with a string value that can be used to access it.

```
let person = new Object();  // Another way to define an empty object would be {}

person['name'] = "John Doe";
person['age'] = 45;

console.log(person.age);    // 45
console.log(person['age']); // 45
```

For ... in

- The **for...in** statement iterates a specified variable over all its properties.
- For each distinct property, JavaScript executes the specified statements.

```
for (let foo in person)
  console.log(foo + " = " + person[foo]);
```

Almost Everything is an Object

- In JavaScript, almost everything is an object.
- All primitive types except null and undefined are treated as objects.

```
let name = "John Doe";
console.log(name.substring(0,4));
```

- In this example, the primitive type is *cast* temporarily into a String object that is discarded afterwards.

Getter and Setters

- A **getter** is a method that gets the value of a specific property.
- A **setter** is a method that sets the value of a specific property.

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    get fullName() {
        return this.firstName + ' ' + this.lastName;
    },
    set fullName (name) {
        let words = name.split(' ');
        this.firstName = words[0];
        this.lastName = words[1];
    }
}

person.fullName = 'John Doe';
console.log(person.firstName); // John
console.log(person.lastName)   // Doe
console.log(person.fullName)   // John Doe
```

Functions are objects

When a function is created using the **function** keyword we are really defining an object.

```
function sayHello() {
  console.log("Hello");
}

sayHello();                                   //Hello
sayHello.info = "This function says hello!";

console.log(sayHello.info);                   //This functions says hello!

sayHello.goodBye = function() {
  console.log("Goodbye");
}

sayHello();                                   //Hello
sayHello.goodBye();                           //Goodbye
```

Constructor functions

Functions can be used to create new objects using the **new** keyword.

```
function Person (name, age, car) {
  this.name = name;
  this.age = age;
  this.car = car;
  this.print = function() {
    console.log(this.name + " is " + this.age + " years old!");
  }
}

let john = new Person("John Doe", 45, {make: "Honda", model: "Civic"});
person.print(); // John Doe is 45 years old!
```

Prototype

- Each *Javascript* function has an internal **prototype** property that is initialized as a nearly empty object.
- When the **new** operator is used on a constructor function, a new object derived from its prototype is created. The function is then executed having the new object as its context.
- We can change the prototype of a function by changing the **prototype property** directly.

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

let john = new Person("John Doe");
Person.age = 45;                    // Only changes the Person function/object
                                    // not its prototype.
let jane = new Person("Jane Doe");
console.log(jane.age);              // undefined

Person.prototype.age = 45;          // Changes the prototype.
let mary = new Person("Mary Doe");  // All objects constructed using the
console.log(mary.age); //45         // person constructor now have an age.
console.log(jane.age); //45         // Even if created before the change.
```

Prototype

You can inspect the prototype of a function easily in the console.

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

Person.prototype; // Object {...}
Person.prototype.saySomething = function (){console.log("Something")};
Person.prototype; // Object { saySomething: Person.prototype.saySomething(), ... }

let john = new Person();
john.saySomething()        // Something
john.constructor;          // function Person(name) { this.name = name; }
john.constructor.prototype // Object { saySomething: Person.prototype.saySomething(), ... }
```

Object __proto__

When a object is created using **new**, a **__proto__** property is initialized with the prototype of the function that created it.

```
function Person(name) {
  this.name = name;
}
let john = new Person("John");

Person.prototype.saySomething = function (){console.log("Something")};
john.prototype;     // undefined
john.__proto__;     // Object { saySomething: Person.prototype.saySomething(), ... }
john.saySomething() // Something
```

When we read a property from an object, and it’s missing, JavaScript will automatically take it from the prototype using **__proto__**.

Inheritance

Inheritance can be emulated in *Javascript* by changing the prototype chain.

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

Person.prototype.print = function() {console.log(this.name);}

function Worker(name, job) {
  this.job = job;
  Person.call(this, name);
}

Worker.prototype = new Person;
Worker.prototype.print =
  function() {console.log(this.name + " is a " + this.job);}

let mary = new Person("Mary");
mary.print(); // Mary
let john = new Worker("John", "Builder");
john.print(); // John is a Builder
```

Classes

- The *class* keyword is just *syntactic sugar* for prototype-based classes.
- Classes can only have methods and getters/setters.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  print() {console.log(this.name);}
}

class Worker extends Person {
  constructor(name, job) {
    super(name);
    this.job = job;
  }
  print() {console.log(this.name + ' is a ' + this.job);}
}

let john = new Worker("John", "Builder");
john.print();
```

Arrays

Arrays

- Arrays are **list-like objects** whose prototype has methods to perform traversal and mutation operations.
- *JavaScript* arrays are zero-indexed
- Arrays can be initialized using a bracket notation:

```
let years = [1990, 1991, 1992, 1993];
console.log(years[0]); // 1990
years.info = "Nice array";
console.log(years.info); // Nice array
```

Array elements are object properties but they cannot be accessed using the **dot** notation because their name is not valid.

```
let years = [1990, 1991, 1992, 1993];
console.log(years[0]); // 1990
console.log(years.0);  // Syntax error
```

Array prototype

By changing the Array prototype we can add methods and properties to all arrays.

```
let years = [1990, 1991, 1992, 1993];
Array.prototype.print = function() {
  console.log("This array has length " + this.length)
};
years.print(); // This array has length 4
```

Array prototype methods

These are some of the methods defined by the Array prototype:

- Properties: prototype, length
- Mutators: fill, pop, push, reverse, shift, sort, splice, unshift
- Accessor: concat, contains, join, slice, indexOf, lastIndexOf
- Iterator: forEach, entries, every, some, filter

Some examples:

```
let years = [1990, 1991, 1992, 1993];
years.push(1994);
console.log(years.length); // 5

years.reverse();
console.log(years);        // [1994, 1993, 1992, 1991, 1990]

let sum = 0;
years.forEach(function (element, index, array) {sum += element});
console.log(sum);          //9960

years.every(function (element, index, array) {return element >= 1990}); //true
years.some(function (element, index, array) {return element % 2 == 0}); //true
```

Exceptions

Throw

- You can throw exceptions using the **throw** statement.
- You can throw any expression.

```
function UserException (message){
  this.message=message;
  this.name="UserException";
}

UserException.prototype.toString = function (){
  return this.name + ": " + this.message;
}

throw new UserException("Value too high");
```

```
throw "This is an error";
```

Error Object

If you are throwing your own exceptions, in order to take advantage of the name and message properties, you can use the **Error** constructor.

```
throw new Error("This is an Error");
```

Try ... Catch

The **try...catch** statement marks a block of statements to try, and specifies a response, should an exception be thrown.

```
try {
  // code to try
}
catch (e) {
   // statements to handle any exceptions
}
```

DOM

DOM

- The **Document Object Model** (DOM) is a **programming interface** for HTML and XML documents.
- It provides a structured representation of the document and it defines a way that the structure can be accessed from programs so that they can change the document **structure**, **style** and **content**.
- The DOM is a fully object-oriented representation of the web page, and it can be modified with a scripting language such as **JavaScript**.

Javascript on HTML Documents

Javascript can be embedded directly into an HTML document:

```
<script>
  // javascript code goes here
</script>
```

Or as an external resource:

```
<script src="script.js"></script>
```

The closing *tag* is mandatory.

Script tag position

As *Javascript* is capable of changing the HTML structure of a document, whenever the browser finds a **script** tag, it first fetches and runs that script and only then resumes loading the page.

Most *Javascript* scripts don't change the document until it is fully loaded but the browser does not know this. For that reason, it was recommended that **script** tags were placed at the bottom of the **body**.

Modern browsers support the async and defer attributes, so scripts can safely be placed in the **head** of the document:

```html
<head>
  <script src="script.js" async></script>
  <script src="script.js" defer></script>
</head>
```

- A asynchronous (**async**) script is run as soon as it is downloaded but without blocking the browser.
- Deferred (**defer**) scripts are executed only when the page is loaded and in order.

Document

The Document object represents an HTML document.

You can access the current document in *Javascript* using the **global** variable **document**.

Some Document **properties**:

- **URL** - read-only location of the document
- **title** - contains the document title
- **location** - a *location* object that can be assigned in order to change to another document

```
document.location = 'http://www.google.com/';
```

There is also another **global** variable that represents the browser called **window**.

Accessing Elements

The following *document* **methods** can be used to access specific HTML elements:

Element **getElementById**(id)	returns the element with the specified id
NodeList **getElementsByClassName**(class)	returns all elements with the specified class
NodeList **getElementsByTagName**(name)	returns all elements with the specified tag name
Element **querySelector**(selector)	returns the first element selected by the specified CSS selector
NodeList **querySelectorAll**(selector)	returns all elements selected by the specified CSS selector

```
let menu = document.getElementById('menu');
let paragraphs = document.getElementsByTagName('p');
let intros = document.querySelectorAll('article p:first-child');
```

Element

An Element object represents an HTML element.

Some common Element **properties**:

id	The id attribute
innerHTML	The HTML code inside the element
outerHTML	The HTML code including this element
style	The CSS style of the element

Element

Some common Element **methods**:

String **getAttribute**(name)	get the attribute with the given name (or null).
setAttribute(name, value)	modifies the attribute with the given name to value.
remove()	removes the element from its parent.

We can also use the same methods we used with the *document* object to access element children:

```
let article = document.getElementById('top-article');
let intro = article.getElementsByTagName('p')[0];
```

Other **methods**: **removeAttribute**, **hasAttribute**

Creating Elements

The **createElement** method of the *document* object can be used to create new elements:

```
let title = 'Some Title';
let intro = 'This is a long introduction';

let article = document.createElement('article');
article.setAttribute('class', 'post');
article.innerHTML = '<h1>' + title + '</h1><p>' + intro + '</p>';

console.log(article.outerHTML);
```

```
<article class="post">
  <h1>Some Title</h1>
  <p>This is a long introduction</p>
</article>
```

This **does not insert** the element anywhere in the *document*.

HTML Element

The HTMLElement inherits from the Element object. There are different HTMLElement objects for each HTML element.

HTMLElement	style, title, blur(), click(), focus()
HTMLInputElement	name, type, value, checked, autocomplete, autofocus, defaultChecked, defaultValue, disabled, min, max, readOnly, required
HTMLSelectElement	name, multiple, required, size, length
HTMLOptionElement	disabled, selected, defaultSelected, text, value
HTMLAnchorElement	href, host, hostname, port, hash, pathname, protocol, text, username, password
HTMLImageElement	alt, src, width, height

Node

The Node object represents a node in the document tree. The *Element* object inherits from the *Node* object.

Some common Node **methods**:

appendChild(node)	appends a node to this node.
replaceChild(new, old)	replaces a child of this node.
removeChild(child)	removes a child from this node.
insertBefore(new, reference);	inserts a new child before the reference child.

Element and Node

Some examples:

```
let element = document.getElementById("menu"); // gets the element with id menu

element.style.color = "blue";                  // changes the text color to blue
element.style.padding = "2em";                 // and the padding to 2em

let paragraph = document.createElement("p");   // creates a new paragraph
paragraph.innerHTML = "Some text";             // inserts text in the paragraph

element.appendChild(paragraph);                // adds the paragraph to the menu
element.remove();                              // removes the menu
```

Traversing the DOM tree

The *Node* object has the following properties that can be used to traverse the DOM tree:

firstChild and **lastChild**	first and last node children of this node.
childNodes	all children nodes as a NodeList.
previousSibling and **nextSibling**	previous and next siblings to this node.
parentNode	parent of this node.
nodeType	the type of the node.

We have to be careful as not all nodes are elements (see node type list)

Traversing the DOM tree

Consider the following HTML:

```html
<article id="article">
  <h1>Title</h1>
  <p>Some text</p>
</article>
```

And the following *Javascript*:

```javascript
let article = document.getElementById('article');
console.log(article.firstChild);                         // #text
console.log(article.firstChild.textContent);             // '\n '
console.log(article.firstChild.nextSibling);             // <h1>
console.log(article.firstChild.nextSibling.textContent); // 'Title'
```

Traversing the DOM tree

To solve this problem, the following properties have been added since *EcmaScript 6*:

firstElementChild and lastElementChild
first and last element children of this node.

children
all children elements as a NodeList.

previousElementSibling and nextElementSibling
previous and next element siblings to this node.

```html
<article id="article">
  <h1>Title</h1>
  <p>Some text</p>
</article>
```

```javascript
let article = document.getElementById('article');
console.log(article.firstElementChild);                        // <h1>
console.log(article.firstElementChild.textContent);            // 'Title'
```

NodeList

- A *NodeList* is an object that behaves like an array of elements.
- Functions like **document.getElementsByTagName()** return a *NodeList*.
- Items in a Node List can be accessed by index like in an array:

```
let elements = document.getElementsByTagName("p");
for (let i = 0; i < elements.length; i++) {
  let element = elements[i];
  // do something with the element
}
```

Events

- Events are sent to notify code of interesting things that have taken place.
- Each event is represented by an object which is based on the Event interface, and may have additional custom fields and/or functions used to get additional information about what happened.

Some possible events:

Mouse	click, dblclick, mousedown, mouseup, mouseenter, mouseleave, mouseover, mousewheel
Keys	keypress, keydown, keyup
Text	cut, copy, paste, select
Form	reset, submit
Input	focus, blur, change

Events in HTML

A possible way to get notified of Events of a particular type (such as click) for a given object is to specify an event handler using:

An HTML attribute named on{eventtype} on an element, for example:

```
<button onclick="return handleClick(event);">
```

or by setting the corresponding property from JavaScript, for example:

```
document.getElementById("mybutton").onclick = function(event) { ... };
```

Add Event Handler

On modern browsers, the *Javascript* function **addEventListener** should be used to handle events.

```
element.addEventListener(type, listener[, useCapture = false])
```

Example:

```
function handleEvent() {
  ...
}

let menu = document.getElementById("menu");
menu.addEventListener("click", handleEvent);
menu.addEventListener("click", function(){...});
```

Event Handler Functions

A function that handles an event can receive a parameter representing the event that caused the function to be called.

```
function handleEvent(event) {
  event.preventDefault();
}

let menu = document.getElementById("menu");
menu.addEventListener("click", handleEvent, false);
```

Depending on its type, the event can have different properties and methods: Reference

To make sure that the original behavior is prevented, we can use the event.preventDefault method.

Bubbling

- When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.
- In each step, the handler can know the current target (*event.currentTarget* or *this*) and also the initial target (*event.target*).

Example where we add some events on all elements and print **this** and **event.target** tag names:

```
<section> <article> <p>Text</p> </article> </section>
```

```
document.querySelector('section').addEventListener('click', function(event){
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName)});
document.querySelector('article').addEventListener('click', function(event){
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName)});
document.querySelector('p').addEventListener('click', function(event){
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName)});
```

Clicking on the paragraph:

```
P - P
ARTICLE - P
SECTION - P
```

To stop bubbling we use the event.stopPropagation method.

Capturing

Event processing has two phases:

- Capturing: goes down to the element.
- Bubbling: the event bubbles up from the element.

Although rarely used, the **useCapture** parameter of the *addEventListener* method, allows us to set the event handler on the capturing phase.

The previous example with some more capture events:

```
document.querySelector('section').addEventListener('click', function(event){
  console.log('Capture: ' + this.tagName + " - " + event.target.tagName)}, true); // notice the true in the end
document.querySelector('article').addEventListener('click', function(event){
  console.log('Capture: ' + this.tagName + " - " + event.target.tagName)}, true);
document.querySelector('p').addEventListener('click', function(event){
  console.log('Capture: ' + this.tagName + " - " + event.target.tagName)}, true);
```

```
Capture: SECTION - P
Capture: ARTICLE - P
Capture: P - P
Bubble: P - P
Bubble: ARTICLE - P
Bubble: SECTION - P
```

On Load Event

As we want to be sure the DOM is completely loaded before adding events to any elements, we normally add any initialization code to the *load* event of the *window* element.

```
window.addEventListener('load', function() {
  // initialization code goes here.
});
```

With *EcmaScript 6* and the *defer* attribute, this is no longer necessary.

Ajax

Ajax

- Asynchronous JavaScript + XML,
- Not a technology in itself, but a term coined in 2005 by **Jesse James Garrett**, that describes an approach to using a number of existing technologies: namely the **XMLHttpRequest** object.

XMLHttpRequest

XMLHttpRequest makes sending HTTP requests very easy.

- Method: get or post.
- Url: The URL to fetch.
- Async: if false, execution will stop while waiting for response.

```
void open(method, url, async);
```

Example:

```
function requestListener () {
  console.log(this.responseText);
}

let request = new XMLHttpRequest();
request.onload = requestListener;
request.open("get", "getdata.php", true);
request.send();
```

Monitoring Progress

```
let request = new XMLHttpRequest();

request.addEventListener("progress", updateProgress);
request.addEventListener("load", transferComplete);
request.addEventListener("error", transferFailed);
request.addEventListener("abort", transferCanceled);

request.open("get", "getdata.php", true);
request.send();

function updateProgress (event) {
  if (event.lengthComputable)
    let percentComplete = event.loaded / event.total;
}

function transferComplete(event) {
  alert("The transfer is complete.");
}

function transferFailed(event) {
  alert("An error occurred while transferring the file.");
}

function transferCanceled(event) {
  alert("The transfer has been canceled by the user.");
}
```

Sending data

To send data to the server, we first must encode it properly:

```
function encodeForAjax(data) {
  return Object.keys(data).map(function(k){
    return encodeURIComponent(k) + '=' + encodeURIComponent(data[k])
  }).join('&');
}
```

Sending it using get:

```
request.open("get", "getdata.php?" + encodeForAjax({id: 1, name: 'John'}), true);
request.send();
```

Sending it using post:

```
request.open("post", "getdata.php", true);
request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
request.send(encodeForAjax({id: 1, name: 'John'}));
```

Analyzing a XMLRequest Response

If you use XMLHttpRequest to get the content of a remote **XML** document, the responseXML property will be a DOM Object containing a parsed XML document, which can be hard to manipulate and analyze.

If you use **JSON**, it is very easy to parse the response as JSON is already in *Javascript Object Notation*.

```
JSON.parse('{}');               // {}
JSON.parse('true');             // true
JSON.parse('"foo"');            // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');             // null
JSON.parse('{"1": 1, "2": 2}') // Object {1: 1, 2: 2}
JSON.parse(this.responseText)  // The server response
```

Advanced Functions

Apply and Call

- The **apply()** method calls a function with a given *this* value, and arguments provided as an array.
- The **call()** method calls a function with a given *this* value and arguments provided individually.

```
function foo(bar1, bar2) {
  console.log(this);
  console.log(bar1);
  console.log(bar2);
}

foo.apply('hello', ['john', 123] ); //hello john 123
foo.call('hello', 'john', 123); //hello john 123
```

Bind

The *bind()* method is similar to *call()* but returns a new function where *this* and any of the initial parameters are set to the provided values.

```
function foo(bar1, bar2) {
  console.log(this);
  console.log(bar1);
  console.log(bar2);
}

let foo2 = foo.bind('hello', 'john');
foo2(123); //hello john 123
```

Closures

A closure is the combination of a function and the lexical environment within which that function was declared.

```
function foo() {
  let number = 123;
  return function bar() {
    console.log(number);
  }
}

bar = foo();
bar(); // 123
```

Closures and Events

Closures are the reason code like this works in *Javascript*:

```
let paragraphs = document.querySelectorAll('p');
for (let i = 0; i < paragraphs.length; i++)
  paragraphs[i].addEventListener('click', function() {
      console.log('I am paragraph #' + i);
  });
```

Several functions were created in this code, and for each one of them, the variable **i** has a different value.

Bind and Events

Sometimes we lose our *this*:

```
class Foo {
    setup() {
      document.querySelector('h1').addEventListener('click', this.bar);
    }

    bar(event) {
      console.log(this);         // the h1 element (we wanted the object)
      console.log(event.target); // the h1 element
    }
}

let foo = new Foo();
foo.setup();
```

We can fix it using *bind*:

```
setup() {
  document.querySelector('h1').addEventListener('click', this.bar.bind(this));
}
```

Partial Functions

Sometimes we might want to do this:

```
document.querySelector('p.blue').addEventListener('click', changeColor('blue'));
document.querySelector('p.red').addEventListener('click', changeColor('red'));

function changeColor(color) {
  this.style.color = color;
}
```

But it obviously doesn't work. A solution would be to create anonymous functions to create a closure:

```
document.querySelector('p.blue').addEventListener('click', function(event) {
  changeColor('blue', event)}
);
document.querySelector('p.red').addEventListener('click', function(event) {
  changeColor('red', event)}
);

function changeColor(color, event) {
  event.target.style.color = color;
}
```

Partial Functions

Instead we can create partial functions using bind:

```
let blue = document.querySelector('p.blue')
blue.addEventListener('click', changeColor.bind(blue, 'blue'));

let red = document.querySelector('p.red')
red.addEventListener('click', changeColor.bind(red, 'red'))

function changeColor(color) {
  this.style.color = color;
}
```

Advanced Arrays

forEach

The *forEach()* method executes a provided function once for each array element.

```
let numbers = [4, 8, 15, 16, 23, 42];
numbers.forEach(function(value, index){
    console.log('Element #' + index + ' is ' + value);
});
```

The result would be:

```
Element #0 is 4
Element #1 is 8
Element #2 is 15
Element #3 is 16
Element #4 is 23
Element #5 is 42
```

Filter

The *filter()* method creates a new array with all elements that pass the test implemented by the provided function.

```
let numbers = [4, 8, 15, 16, 23, 42];
let even = numbers.filter(function(n) {return n % 2 == 0});
console.log(even); // [ 4, 8, 16, 42 ]
```

Or using arrow functions:

```
let numbers = [4, 8, 15, 16, 23, 42];
let even = numbers.filter(n => n % 2 == 0);
console.log(even); // [ 4, 8, 16, 42 ]
```

The alternative would be:

```
let numbers = [4, 8, 15, 16, 23, 42];
let even = [];
for (let i = 0; i < numbers.length; i++)
  if (numbers[i] % 2 == 0) even.push(numbers[i]);
console.log(even); // [ 4, 8, 16, 42 ]
```

Map

The *map()* method creates a new array with the results of calling a provided function on every element in the calling array.

```
let numbers = [4, 8, 15, 16, 23, 42];
var doubled = numbers.map(function(n) {return n * 2});
console.log(doubled); // 8, 16, 30, 32, 46, 84
```

Or using arrow functions:

```
let numbers = [4, 8, 15, 16, 23, 42];
let doubled = numbers.map(n => n * 2);
console.log(doubled); // 8, 16, 30, 32, 46, 84
```

Generic use of map

The *map()* method can be used on other types of *array like* objects:

```
var ascii = Array.prototype.map.call('John', function(letter) {
  return letter.charCodeAt(0);
});
console.log(ascii); // 74, 111, 104, 110
```

Simpler:

```
var ascii = [].map.call('John', function(letter) {
  return letter.charCodeAt(0);
});
console.log(ascii); // 74, 111, 104, 110
```

A more useful example:

```
var inputs = document.querySelectorAll('input[type=number]');
var values = [].map.call(inputs, function(input) {
  return input.value;
});
console.log(values); // an array with all the number input values
```

Reduce

The *reduce()* method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
let numbers = [4, 8, 15, 16, 23, 42];
let total = numbers.reduce(function(current, number) {
  return current + number;
});
console.log(total); // 108
```

Or with arrow functions:

```
[4, 8, 15, 16, 23, 42].reduce( (c, n) => c + n ); // 108
```

Objects to Arrays

Sometimes we need to convert an *array like* object (like *NodeList*) to a true array so that we can use these awesome new array functions.

```
let paragraphs = document.querySelectorAll('p');
```

There are several ways to achieve this:

```
let array1 = Array.apply(null, paragraphs);
let array2 = Array.prototype.slice.call(paragraphs);
let array3 = [].slice.call(paragraphs);
let array4 = [...paragraphs]; // the ECMAScript 2015 spread operator
```

Timers

Set Timeout

The *window* object has a function (*setTimeout*) that sets a timer which executes a function, or specified piece of code, once it expires:

```
let id = window.setTimeout(function() {alert('Yay!')}, 5000);
```

The return value is an *id* that can be used to cancel the timer:

```
window.clearTimeout(id);
```

Set Interval

Another function (*setInterval*) executes executes a function, or specified piece of code, with a fixed time delay between each call.

```
let counter = 1;
let id = window.setInterval(function() {
  console.log('Yay! ' + counter++)
}, 1000);
```

The return value is an *id* that can be used to cancel the timer:

```
window.clearInterval(id);
```

jQuery

jQuery

jQuery is a *Javascript* library that solves several different problems:

- Inadequacy of the *Javscript* DOM.
- Browser compatibility issues.
- Verbosity of some *Javascript* commands.

Most of these have been mitigated by recent advances in the *Javascript* standard.

How it works

- *jQuery* defines a function/object called *$* (yes, the dollar sign).
- This function is responsible for selecting and filtering elements, traversing and modifying the DOM, …
- Elements selected are returned nested inside a *$* object making it harder to mix *jQuery* with plain *Javascript* code.

Example:

```
$('p').click(function() {
  console.log($(this).text());
});
```

In plain Javascript this would be:

```
let paragraphs = document.querySelectorAll('p');
for (let i = 0; i < paragraphs.length; i++)
  paragraphs[i].addEventListener('click', function(){
    console.log(this.textContent);
  });
```

Drawbacks

- *jQuery* is big (85Kb minified).
- *jQuery* is slow (mainly due to having to maintain compatibility with older browsers).
- You end up being trapped into the *jQuery* ecosystem.

Alternatives

Roll your own:

```javascript
function $(selector) {
  return document.querySelectorAll(selector);
}

NodeList.prototype.css = function(property, value) {
  [].forEach.call(this, function(element) {
    element.style[property] = value;
  });
  return this;
}

$('p').css('color', 'red').css('background-color', 'blue');
```

Smaller and simpler alternatives like: http://zeptojs.com/ (25Kb)

Just use plain Javascript: https://plainjs.com/