WidkaJS

What I didn't know about JavaScript

Kevin Putrajaya

Why JavaScript?

- Simple and forgiving for beginners
- Powerful, though hard to tame
- Good performance for a scripting language
- Cross-platform (web, mobile, desktop)
- One of the most popular programming languages
- I wasn't properly introduced to it (jQuery, anyone?)

Brief history

- Created in 1995 by Brendan Eich (Netscape), released with Netscape 2 in 1996
- Planned to be called LiveScript, but was renamed to JavaScript in a marketing decision because Java was popular
- Months later, Microsoft followed up with JScript in IE3
- Netscape submitted JavaScript to Ecma International (European standards organization), resulted in ECMAScript 3
- Currently we're in between ECMAScript 5 (December 2009) and ECMAScript 6 (June 2015)

Language features

> C-3PO: "[...] I don't know what all this trouble is about, but I'm sure it must be your fault."
> R2D2: **beeps an angry response**
> C-3PO: You watch your language!

— *Star Wars: Episode IV - A New Hope*

Types

- Number
- String
- Boolean
- Symbol (new in ES6, used as object keys)
- Object
    - Function (functions are first-class objects)
    - Array
    - Date
    - RegExp
- null (deliberate non-value)
- undefined (uninitialized value)

Functions as first-class objects

- A function is an instance of the Object type

```
function eat() {
    console.log('yum');
}
console.log(eat instanceof Object); // true
```

- A function can have properties and has a link back to its constructor method

```
eat.isImportant = true;
console.log(eat.isImportant); // true
console.log(eat.constructor); // function Function()
```

- You can store function in a variable

```
var work = eat;
work(); // 'yum'
```

Functions as first-class objects (cont'd)

- You can pass function as a parameter to another function

```
function doImportantTask(task) {
    if (task.isImportant) {
        task();
    }
}
doImportantTask(eat); // 'yum'
```

- You can return function from a function

```
function prepareFood(){
   console.log('preparing');
   return eat;
}
var food = prepareFood(); // 'preparing'
food(); // 'yum'
```

Constructors

JavaScript doesn't have 'classes', but it has constructors.

```
var john = {
    firstName: 'John',
    lastName: 'Doe'
};
var jane = {
    firstName: 'Jane',
    lastName: 'Doe'
};

// to get fullnames, create a function -- good
var getFullName = function (person) {
    return person.firstName + ' ' + person.lastName;
};

console.log(getFullName(john)); // 'John Doe'
console.log(getFullName(jane)); // 'Jane Doe'
```

Constructors (cont'd)

Constructors make your code more structured and future-proof.

```
// use constructor instead to keep scope clean -- better
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function () {
        return this.firstName + ' ' + this.lastName;
    };
}

var john = new Person('John', 'Doe');
var jane = new Person('Jane', 'Doe');

console.log(john.getFullName()); // 'John Doe'
console.log(jane.getFullName()); // 'Jane Doe'
```

Prototypal inheritance

Proto-to-type-pal what?

```
var animal = { eats: true };

// inheriting animal
var rabbit = { jumps: true };
rabbit.__proto__ = animal; // inherits
console.log(rabbit.eats); // true

// inheritance is by reference
animal.moves = true;
console.log(rabbit.moves); // true

// if own property found, use it
var fedUpRabbit = { eats: false };
fedUpRabbit.__proto__ = rabbit; // inherits
console.log(fedUpRabbit.eats); // false
```

Prototypal inheritance (cont'd)

Let's throw in constructors in the equation.

```
function Animal() {
    this.eats = true;
}
function Rabbit(name) {
    this.name = name;
    this.jumps = true;
}
Rabbit.prototype = new Animal();

var rabbit = new Rabbit('Bunny');
console.log(rabbit.name, rabbit.eats, rabbit.jumps);
// 'Bunny' true true
```

Asynchronous execution

JavaScript supports single-threaded asynchronous execution.

To utilize it, we generally use "callbacks" or "resolvers" (functions).

```
setTimeout(function () {
    console.log('Hello');
}, 1000);
console.log('World');
```

```
World
Hello
```

A glimpse of the future

> Eleven: **sobs** "Mike..."
> Mike: "Is everything okay?"
> Eleven: **nods**
> Mike: "Are you sure?"
> Eleven: "Promise."

— *Stranger Things*

Promise

Say we have function `doThings(data, success)`.

We need to chain this method to get the result we want.

```
doThings(data, function (res1) {
    doThings(res1, function (res2) {
        doThings(res2, function (res3) {
            console.log(res3); // success
        });
    });
});
```

Promise (cont'd)

If we want to handle errors like `doThings(data, success, error)`, things would get more awkward (and ugly).

```
doThings(data, function (res1) {
    doThings(res1, function (res2) {
        doThings(res2, function (res3) {
            console.log(res3); // success
        }, function (err3) {
            console.error(err3); // error
        });
    }, function (err2) {
        console.error(err2); // error
    });
}, function (err1) {
    console.error(err1); // error
});
```

Promise (cont'd)

Promise is a simpler alternative to manage async operations.

It's been around even before ES6 (Q, Bluebird, AngularJS, jQuery).

```
var promise = new Promise(function (resolve, reject) {
    // do things, probably asynchronously
    if (successful) {
        resolve('Stuff worked!');
    } else {
        reject(Error('It broke'));
    }
});
```

Let's promisify `doThings` then.

```
var doPromise = function (data) {
    return new Promise(function (resolve, reject) {
        doThings(data, resolve, reject);
    });
};
```

Promise (cont'd)

We now can rewrite our previous code like this.

```
doPromise(data)
.then(function (res1) {
    return doPromise(res1);
})
.then(function (res2) {
    return doPromise(res2);
})
.then(function (res3) {
    console.log(res3); // success
})
.catch(function (err) {
    console.error(err); // error
});
```

Promise (cont'd)

Then simplify things a bit, syntactically.

```
doPromise(data)
.then(doPromise)
.then(doPromise)
.then(console.log) // success
.catch(console.error); // error
```

Promise (cont'd)

To do multiple async calls in parallel, `Promise.all` is convenient.

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("zero");
    }, 1000);
});
var p2 = 1234;
var p3 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("test");
    }, 3000);
});

Promise.all([p1, p2, p3])
.then(console.log); // after 3s: ["zero", 1234, "test"]
```

Promise (cont'd)

It also has a fail-fast behaviour so we don't waste any time.

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject("zero"); // now reject this
    }, 1000);
});
var p2 = 1234;
var p3 = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("test");
    }, 3000);
});

Promise.all([p1, p2, p3])
.then(console.log)
.catch(console.error); // after 1s: "zero"
```

Arrow functions

Arrow functions allow us to be more concise when building lambda expressions — something you do often in JavaScript.

```
['John', 'Jane'].map(function (name) {
    return 'Hello, ' + name;
});
```

```
['John', 'Jane'].map(name => 'Hello, ' + name);
```

Default parameters

Better late than never? JavaScript finally got this.

```
var adjust = function (height, color) {
    height = height || 50;
    color = color || 'red';
    // do things
};
```

```
var adjust = function (height = 50, color = 'red') {
    // do things
};
```

Template literals

And this, as well.

```
var name = 'Your name is ' + first + ' ' + last + '.';
var url = 'http://localhost:3000/api/messages/' + id;
```

```
var name = `Your name is ${first} ${last}.`;
var url = `http://localhost:3000/api/messages/${id}`;
```

Conclusions

- JavaScript is more complex than it looks
- JavaScript has a great potential at its core
- JavaScript is catching up with more powerful features
- Asynchronous programming is fun!

References and further reads

- Marp (Markdown Presentation Writer), Yuki Hattori
- A Re-introduction to JavaScript, Mozilla Dev Net
- Functions Are First-Class Objects in JavaScript, Helen Emerson
- Inheritance and the prototype chain, Mozilla Dev Net
- Promises for asynchronous programming, Axel Rauschmayer
- Learning JavaScript Design Patterns, Addy Osmani
- Top 10 ES6 Features, Azat Mardan