OOP w/ Javascript

OOP w/ Javascript

Part I - Classes

The past

"You have to know the past to understand the present."

- Carl Sagan

There were always very limited approaches to create class-like objects in Javascript.

Javascript Objects

```
1 var name = "Cthulhu";
2
3 var GreatOld = {
4     getName: function() {
5         return name;
6     }
7 };
8
9 console.log(GreatOld.getName()); // Cthulhu
```

Prototypes

```
1  function GreatOld(name) {
2      this._name = name;
3  };
4
5  GreatOld.prototype.getName = function() {
6      return this._name;
7  };
8
9  var cthulhu = new GreatOld("Cthulhu");
10
11 console.log(cthulhu.getName()); // Output: Cthulhu
```

Prototypes

```
1  function GreatOld(name) {
2      this._name = name;
3  };
4
5  GreatOld.prototype.getName = function() {
6      return this._name;
7  };
8
9  var cthulhu = new GreatOld("Cthulhu");
10
11 console.log(cthulhu.getName()); // Output: Cthulhu
```

Prototypes

```
1  function GreatOld(name) {
2      this._name = name;
3  };
4
5  GreatOld.prototype.getName = function() {
6      return this._name;
7  };
8
9  var cthulhu = new GreatOld("Cthulhu");
10
11 console.log(cthulhu.getName()); // Output: Cthulhu
```

Prototypes and inheritance

```
1  function OuterGod(name) {
2      this._name = name;
3  };
4
5  OuterGod.prototype.getName = function() {
6      return this._name;
7  };
8
9  function GreatOld(name, housing) {
10     OuterGod.call(this, name);
11     this.housing = housing;
12 }
13
14 GreatOld.prototype = Object.create(OuterGod.prototype);
15 GreatOld.prototype.constructor = GreatOld;
16 GreatOld.prototype.getName = function() {
```

Prototypes and inheritance

```
7  };
8
9  function GreatOld(name, housing) {
10     OuterGod.call(this, name);
11     this.housing = housing;
12 }
13
14 GreatOld.prototype = Object.create(OuterGod.prototype);
15 GreatOld.prototype.constructor = GreatOld;
16 GreatOld.prototype.getName = function() {
17     let name = OuterGod.prototype.getName.call(this);
18
19     return `Our lord and savor ${name}`;
20 }
21 GreatOld.prototype.getHousing = function() {
22     return this.housing;
23 }
24
25 var nyarlathotep = new OuterGod("Nyarlathotep");
```

Prototypes and inheritance

```
12 }
13
14 GreatOld.prototype = Object.create(OuterGod.prototype);
15 GreatOld.prototype.constructor = GreatOld;
16 GreatOld.prototype.getName = function() {
17     let name = OuterGod.prototype.getName.call(this);
18
19     return `Our lord and savor ${name}`;
20 }
21 GreatOld.prototype.getHousing = function() {
22     return this.housing;
23 }
24
25 var nyarlathotep = new OuterGod("Nyarlathotep");
26 var cthulhu = new GreatOld("Cthulhu", "R'lyeh");
27
28 console.log(nyarlathotep.getName()); // Nyarlathotep
29 console.log(cthulhu.getName()); // Our lord and savor Cthulhu
30 console.log(cthulhu.getHousing()); // R'lyeh
```

The present

The prototype syntax got wrapped by the new class syntax. That means all prototyping-features are available for the classes.

So OOP in Javascript becomes readable and fun again.

The present

The prototype syntax got wrapped by the new class syntax. That means all prototyping-features are available for the classes.

So OOP in Javascript becomes readable and fun again.

But it's still limited compared to other languages.

Defining classes

```
1  class OuterGod {
2      constructor(name) {
3          this._name = name;
4      }
5
6      getName() {
7          return this._name;
8      }
9  }
10
11 class GreatOld extends OuterGod {
12     constructor(name, housing) {
13         // OuterGod.call(this, name);
14         super(name);
15
16         this.housing = housing;
```

Defining classes

```
11 class GreatOld extends OuterGod {
12     constructor(name, housing) {
13         // OuterGod.call(this, name);
14         super(name);
15 
16         this.housing = housing;
17     }
18 
19     getName() {
20         // let name = OuterGod.prototype.getName.call(this);
21         let name = super.getName();
22 
23         return `Our lord and savor ${name}`;
24     }
25 
26     getHousing() {
27         return this.housing;
28     }
```

Defining classes

```
 4      }
 5
 6      getName() {
 7          return this._name;
 8      }
 9  }
10
11  class GreatOld extends OuterGod {
12      constructor(name, housing) {
13          // OuterGod.call(this, name);
14          super(name);
15
16          this.housing = housing;
17      }
18
19      getName() {
20          // let name = OuterGod.prototype.getName.call(this);
21          let name = super.getName();
22
```

Defining classes

```
class Creator extends OuterGod {
    constructor(name, housing) {
        // OuterGod.call(this, name);
        super(name);

        this.housing = housing;
    }

    getName() {
        // let name = OuterGod.prototype.getName.call(this);
        let name = super.getName();


        return `Our lord and savor ${name}`;
    }

    getHousing() {
        return this.housing;
    }
}
```

Defining classes

```
18
19      getName() {
20          // let name = OuterGod.prototype.getName.call(this);
21          let name = super.getName();
22
23          return `Our lord and savor ${name}`;
24      }
25
26      getHousing() {
27          return this.housing;
28      }
29  }
30
31  let nyarlathotep = new OuterGod("Nyarlathotep");
32  let cthulhu = new GreatOld("Cthulhu", "R'lyeh");
33
34  console.log(nyarlathotep.getName()); // Nyarlathotep
35  console.log(cthulhu.getName()); // Our lord and savor Cthulhu
36  console.log(cthulhu.getHousing()); // R'lyeh
```

More syntax and features

super();

Comparable to PHPs *parent()*

```
1  class OuterGod {
2      constructor(name) {
3          this._name = name;
4      }
5
6      getName() {
7          return this._name;
8      }
9  }
10
11 class GreatOld extends OuterGod {
12     constructor(name, housing) {
13         super(name);
14         this._housing = housing;
15     }
16
17     getHousing() {
18         return this._housing;
19     }
20 }
21
22 let cthulhu = new GreatOld("Cthulhu", "R'lyeh");
23
24 console.log(cthulhu.getName(), " dreams in ", cthulhu.getHousing()); // Cthulhu dreams in R'lyeh
```

Super is therefor like a reference to the extended ***prototype***.

Hoisting*

While functions are hoisted, classes aren't!

```
1  let designer = new Designer(); // Instance of Designer
2
3  function Designer() {
4
5  }
6
7  // -----
8
9  let developer = new GreatOld(); // ReferenceError
10
11 class GreatOld {
12
13 }
```

**Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.*

Getter and Setter methods

Allows e.g. validations, logging, async requests, ... when setting/getting data.
Comparable to PHPs *__get()* or *__set()*

```
1  class GreatOld {
2      constructor(name) {
3          this._name = name;
4      }
5
6      set name(name) {
7          if (name.trim().length === 0) {
8              throw new Error("Invalid value provided");
9          }
10
11         this._name = name;
12     }
13
14     get name() {
15         return this._name;
16     }
17 }
```

Getter and Setter methods

Allows e.g. validations, logging, async requests, ... when setting/getting data.
Comparable to PHPs *__get()* or *__set()*

```
2       constructor(name) {
3           this._name = name;
4       }
5
6       set name(name) {
7           if (name.trim().length === 0) {
8               throw new Error("Invalid value provided");
9           }
10
11          this._name = name;
12      }
13
14      get name() {
15          return this._name;
16      }
17  }
18
19  let cthulhu = new GreatOld("Cthulhu");  // Output: Cthulhu
20  cthulhu.name = "Peter"; // Output: Peter
21  cthulhu.name = ""; // Output: Error: Invalid value provided
```

Getter and Setter methods

Allows e.g. validations, logging, async requests, ... when setting/getting data.
Comparable to PHPs *__get()* or *__set()*

```
2       constructor(name) {
3           this._name = name;
4       }
5
6       set name(name) {
7           if (name.trim().length === 0) {
8               throw new Error("Invalid value provided");
9           }
10
11          this._name = name;
12      }
13
14      get name() {
15          return this._name;
16      }
17  }
18
19  let cthulhu = new GreatOld("Cthulhu");  // Output: Cthulhu
20  cthulhu.name = "Peter"; // Output: Peter
21  cthulhu.name = ""; // Output: Error: Invalid value provided
```

Static methods

These are statically callable by using the classname, like in PHP.

```
1  class OuterGod {
2    constructor(name) {
3      this._name = name;
4    }
5
6    static destroyWorld() {
7      return "Been there, done that.";
8    }
9  }
10
11 class GreatOld extends OuterGod {
12
13 }
14
15 console.log(OuterGod.destroyWorld()); // "Been there, done that."
16 console.log(GreatOld.destroyWorld()); // "Been there, done that."
```

Computed methods

Usage might be comparable to unbelievable, cosmic horror.

```
// default
class GreatOld extends OuterGod {
    destroyWorld() { }
}

// partial dynamic
const place = "World";

class GreatOld extends OuterGod {
    ["destroy" + place]() { }
}

// full dynamic
const destroyWorldMethod = "destroyWorld";

class GreatOld extends OuterGod {
    [destroyWorldMethod]() { }
}
```

Part II - Module syntax

What and why?

What and why?

- Modules are reuseable code-fragments.

What and why?

- Modules are reuseable code-fragments.
- They can be organized in a well-arranged folder structure.

What and why?

- Modules are reuseable code-fragments.
- They can be organized in a well-arranged folder structure.
- Similar to languages like C or JAVA, modules can only be imported when needed to lower the overhead.

What and why?

- Modules are reuseable code-fragments.
- They can be organized in a well-arranged folder structure.
- Similar to languages like C or JAVA, modules can only be imported when needed to lower the overhead.
- When bundling: The size of the final bundle/application will change automatically.

What and why?

- Modules are reuseable code-fragments.
- They can be organized in a well-arranged folder structure.
- Similar to languages like C or JAVA, modules can only be imported when needed to lower the overhead.
- When bundling: The size of the final bundle/application will change automatically.
- Only specific logic can be exposed, similar to public and private methods.

How?

How?

Import / Export

export

You can export functions, const, let, var and classes but if have to happen on the top level. Unexposed methods will not be available for import.

```
// GreatOld.js
export const GreatOldEnum = {
    CTHULHU: "Cthulhu"
};

export class GreatOld {
    // ...
}

export function destroyWorld() {
    // ...
}

// or

const GreatOldEnum = {
    CTHULHU: "Cthulhu"
```

export

You can export functions, const, let, var and classes but if have to happen on the top level. Unexposed methods will not be available for import.

```
 9
10 export function destroyWorld() {
11     // ...
12 }
13
14 // or
15
16 const GreatOldEnum = {
17     CTHULHU: "Cthulhu"
18 };
19
20 class GreatOld {
21     // ...
22 }
23
24 function destroyWorld() {
25     // ...
26 }
27
```

export

You can export functions, const, let, var and classes but if have to happen on the top level. Unexposed methods will not be available for import.

```
 9
10 export function destroyWorld() {
11     // ...
12 }
13
14 // or
15
16 const GreatOldEnum = {
17     CTHULHU: "Cthulhu"
18 };
19
20 class GreatOld {
21     // ...
22 }
23
24 function destroyWorld() {
25     // ...
26 }
27
```

import

All variables/classes/functions which are exported are non-anonymous and can be imported by directly naming them in the import-statement.

```
1 // GreatOld.js
2 export const GreatOldEnum = {
3     CTHULHU: "Cthulhu"
4 };
5
6 export class GreatOld {
7     // ...
8 }
```

```
1 // app.js
2 import { GreatOld, GreatOldEnum }  from './GreatOld.js';
```

export default

Defining an *export default* in a module allows to expose an anonymous variable/function/class.
When importing a module, the *export default* will be referenced with a given name.

```
1  // GreatOld.js
2  export const GreatOldEnum = {
3      CTHULHU: "Cthulhu"
4  };
5
6  export default class {
7      // ...
8  }
9
10 // app.js
11 import GreatOld, {GreatOldEnum} from './GreatOld.js';
```

There is only one *default* allowed per file.

import as alias

Sometimes classes have different pathes but the same name. Therefor you can alias the imports.

```
1 // app.js
2 import {NameEnum as GreatOldEnum} from './GreatOld.js';
3 import {NameEnum as OuterGodEnum} from './OuterGods.js';
```

Module => Object

With a little help of our friends *alias* and *wildcard* we can also move all exported logic to a Javascript object.

```
1  // GreatOld.js
2  export const types = {
3      CTHULHU: "Cthulhu"
4  };
5
6  export function destroyWorld {
7      // ...
8  }
9
10 // app.js
11 import * as GreatOld from './GreatOld.js';
12
13 console.log(GreatOld.types); // { CTHULHU: "Cthulhu" }
```

Better use classes to create such an object. Alternatively *export default* a class instance of you want to use singletonpatterns.

Aggregation

Sometimes you want to combine multiple modules in one aggregated module.

```
// Folder structure
modules/
    canvas.js
    shapes.js
    shapes/
        circle.js
        square.js
        triangle.js

// shapes.js
export { Square } from './shapes/square.js';
export { Triangle } from './shapes/triangle.js';
export { Circle } from './shapes/circle.js';

// canvas.js
// before
import { Square } from './modules/square.js';
```

Aggregation

Sometimes you want to combine multiple modules in one aggregated module.

```
// Folder structure
modules/
    canvas.js
    shapes.js
    shapes/
        circle.js
        square.js
        triangle.js

// shapes.js
export { Square } from './shapes/square.js';
export { Triangle } from './shapes/triangle.js';
export { Circle } from './shapes/circle.js';

// canvas.js
// before
import { Square } from './modules/square.js';
import { Circle } from './modules/circle.js';
import { Triangle } from './modules/triangle.js';

```

Aggregation

Sometimes you want to combine multiple modules in one aggregated module.

```
    canvas.js
    shapes.js
    shapes/
        circle.js
        square.js
        triangle.js

// shapes.js
export { Square } from './shapes/square.js';
export { Triangle } from './shapes/triangle.js';
export { Circle } from './shapes/circle.js';

// canvas.js
// before
import { Square } from './modules/square.js';
import { Circle } from './modules/circle.js';
import { Triangle } from './modules/triangle.js';

// after
import { Square, Circle, Triangle } from './modules/shapes.js';
```

Dynamic module loading

One of the newest features in browsers is to import modules dynamically witch is similar to the wildcard import.

```
// GreatOld.js
export class GreatOld {
    destroyWorld {
        // ...
    }
}

// app.js
someButton.addEventListener('click', () => {
    import('./GreatOld.js').then((Module) => {
        let cthulhu = new Module.GreatOld("Cthulhu");
        cthulhu.destroyWorld();
    })
});
```

Additional

- ***.mjs** is a possible file extension for module files.

Additional

- ***.mjs** is a possible file extension for module files. But caution, this is not fully supported by browsers and manly only for Node.js servers. But Babel can compile these files to JS-files.

Summary

Summary

- Use the class-syntax to create reuseable logic

Summary

- Use the class-syntax to create reuseable logic
- Organize your classes in modules

Summary

- Use the class-syntax to create reuseable logic
- Organize your classes in modules
- Think about a reasonable exposure of logic

Summary

- Use the class-syntax to create reuseable logic
- Organize your classes in modules
- Think about a reasonable exposure of logic
- Use processors like Babel to have the full support of new features as the Javascript-Specs might change fast.

Sources

- An overview
- ES6 Definitions
- Class expression
- Hoisting explained
- Modules