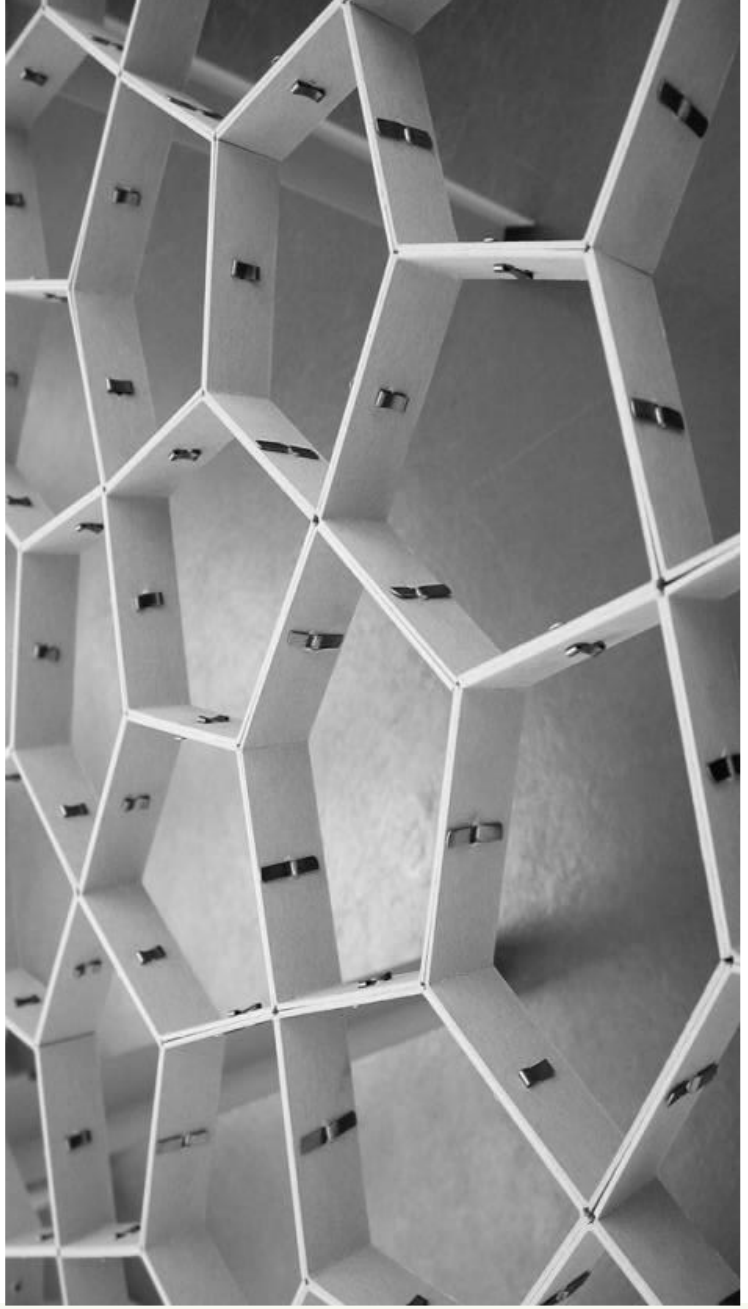
<WA1/>

2020

JavaScript Classes and Modules

“The” language of the Web

Enrico Masala
Fulvio Corno
Luigi De Russis



JavaScript: The Definitive Guide, 7th Edition

- Chapter 9. Classes

Mozilla Developer Network

- Learn web development JavaScript » Dynamic client-side scripting » Introducing JavaScript objects
- Web technology for developers » JavaScript » JavaScript reference » Classes

You Don't Know JS: this & Object Prototypes

- Chapter 5: Prototypes

Modular JS programming

PROTOTYPES

A Prototype-based Language

- JavaScript is an object-based language based on prototypes, rather than being class-based
  - classes exist but they are "syntactical sugar", primarily
- Every JS object has a hidden (internal) property `[[Prototype]]` that points to a second object associated with it (or it is null)
  - Read with `Object.getPrototypeOf(object)`
  - Change with `Object.setPrototypeOf(object, prototype)`
  - Usually also accessible with `.__proto__` (double underscores) – but *deprecated!*

A Prototype-based Language

- This second object is known as an *object prototype*
- Such object also has a [[Prototype]] property, that links to a 3rd object
- …until the [[Prototype]] is null
- Usually, only `Object` (top-level object) points to a null prototype

- Classes and constructor functions also have a `.prototype` attribute, that points to prototype objects for objects created by them
  - Do not confuse `.prototype` and `[[Prototype]]`

Prototype Chaining

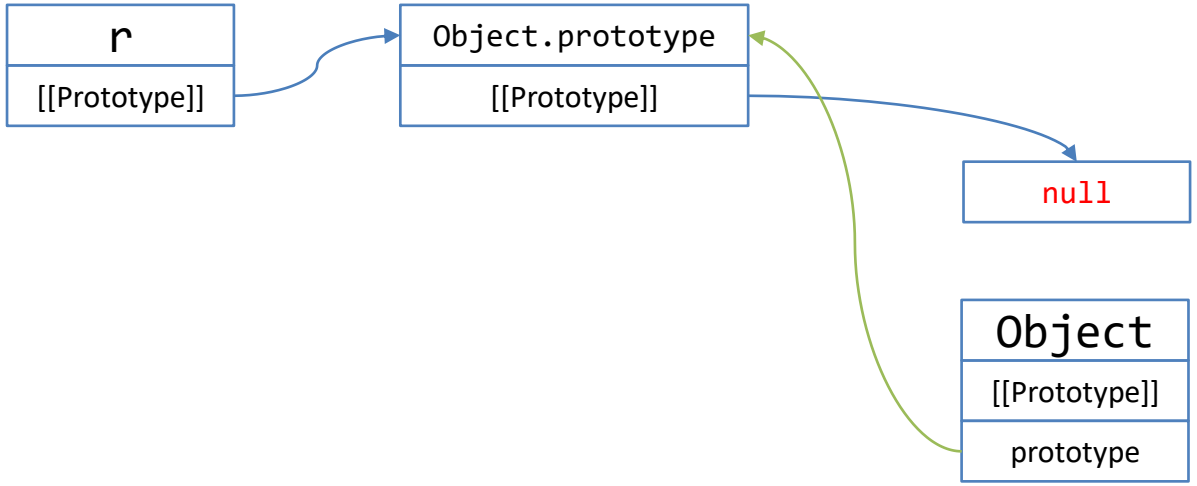
```
function Person(name) {
  this.name = name;
}

const p = new Person('Fulvio');

const d = new Date();

const r = {min: 0, max: 30};

console.log(p); // Person {name: "Fulvio"}
console.log(d); // Thu Apr 09 2020 21:06:29
GMT+0200 (Central European Summer Time)
console.log(r); // Object {min: 0, max: 30}
```



Prototype Chaining

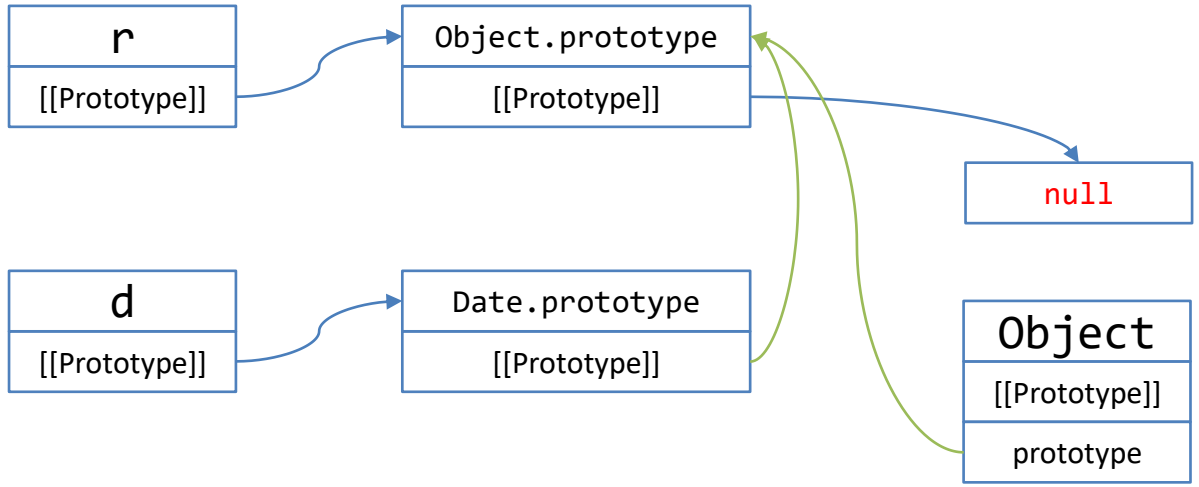
```
function Person(name) {
  this.name = name;
}

const p = new Person('Fulvio');

const d = new Date();

const r = {min: 0, max: 30};

console.log(p); // Person {name: "Fulvio"}
console.log(d); // Thu Apr 09 2020 21:06:29
GMT+0200 (Central European Summer Time)
console.log(r); // Object {min: 0, max: 30}
```



Prototype Chaining

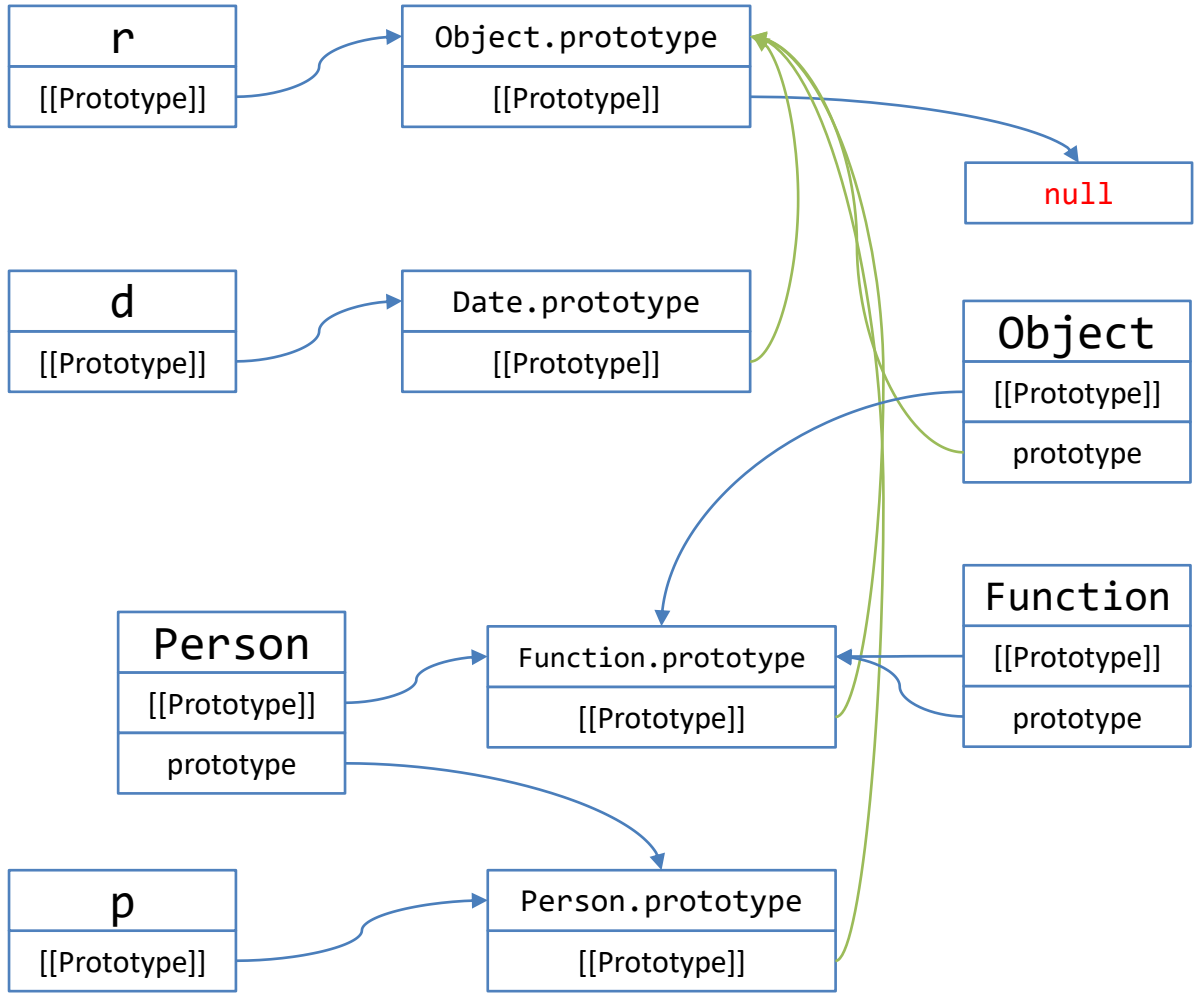
```
function Person(name) {
  this.name = name;
}

const p = new Person('Fulvio');

const d = new Date();

const r = {min: 0, max: 30};

console.log(p); // Person {name: "Fulvio"}
console.log(d); // Thu Apr 09 2020 21:06:29
GMT+0200 (Central European Summer Time)
console.log(r); // Object {min: 0, max: 30}
```



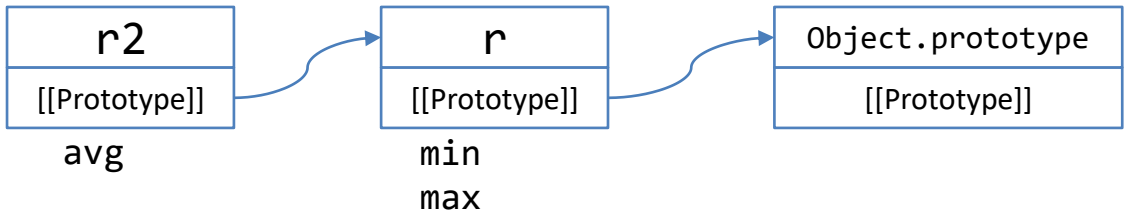
Object.prototype

- Prototype chains usually end at `Object.prototype`
  - Its `[[Prototype]]` is `null`
- `Object.prototype` defines many properties and methods that are common to all JS objects
  - .toString(), .valueOf(), .getPrototypeOf(), .setPrototypeOf(), .toSource(), .isPrototypeOf(), .hasOwnProperty(), …
- All objects created by object literals (i.e., `{}`) have the *same* prototype object: `Object.prototype`

Chaining objects

- `Object.create(obj)` will create a new object and link its prototype to the obj
- The resulting object may be modified to add new properties

```
const r = {min: 0, max: 30};
const r2 = Object.create(r);
r2.avg=15;
```



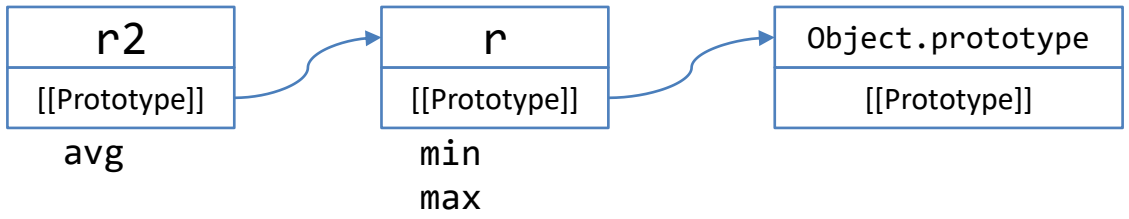
Accessing “inherited” properties

- Prototypes are used in accessing object properties
  - Not “real” inheritance
- Reading properties
  - If the property is defined on the object, use it
  - If it’s not defined, JS will search on the [[Prototype]] chain
    - If it’s found somewhere, its value is used
    - If ‘null’ is reached, then return undefined
- Writing properties
  - Doesn’t follow the prototype chain (*)
  - If it’s not defined on the object, a new one is created
    - And may shadow a same-name property on the prototype chain

(*) not really true: read-only inherit properties and setters of inherited properties behave differently

Example

```
const r = {min: 0, max: 30};
const r2 = Object.create(r);
r2.avg=15;
```




```
> r.min
0
> r.min=5
5
> r2.min
5
```

Setting a property higher on the chain affects all the objects below

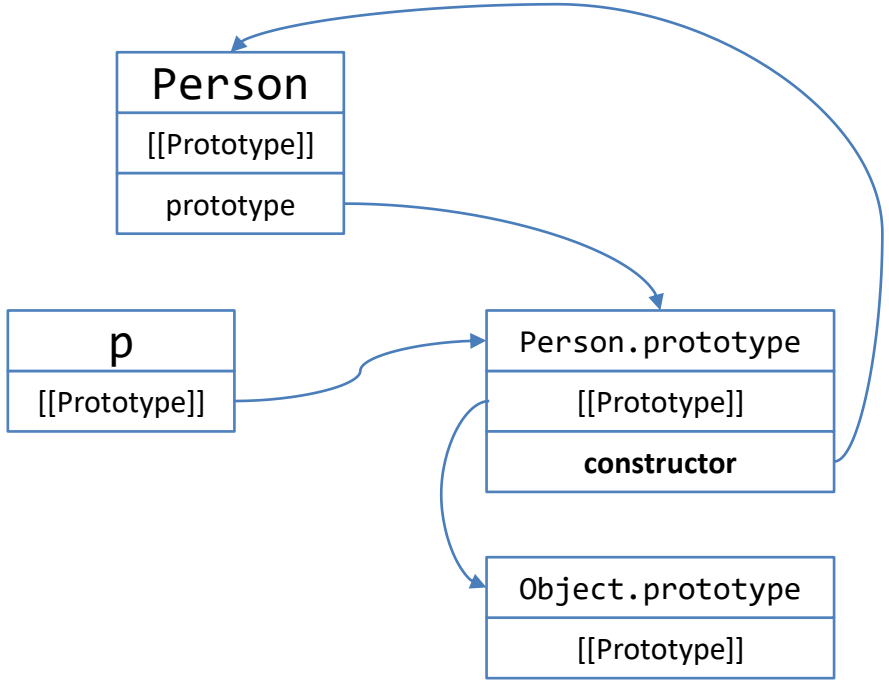
```
> r.max
30
> r2.max=50
50
> r.max
30
```

Setting a property low on the chain shadows the upper properties

Constructor functions

- When an object is created by a Constructor Function (with `new`), the prototype is set to the `.prototype` attribute of the function
- Function prototypes are automatically generated
- They contain a `.constructor` property that refer to the function itself

```
const p = new Person('Fulvio');
```



Which properties are visible?

Method	Own properties	Inherited properties
`for(v in obj)`	Yes	Yes
`xxx = obj.v`	Yes	Yes
`obj.v = xxx`	Yes	No (*)
`Object.entries(obj)`	Yes	No
`Object.values(obj)`	Yes	No
`Object.keys(obj)`	Yes	No
`Object.getOwnPropertyNames(obj)`	Yes	No

Class-based vs. Prototype-based Languages

Category	Class-based (Java)	Prototype-based (JavaScript)
Class vs. Instance	Class and instance are distinct entities.	All objects can inherit from another object.
Definition	Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Creation of new object	Create a single object with the `new` operator.	Same.
Construction of object hierarchy	Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inheritance model	Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Extension of properties	Class definition specifies *all* properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an *initial set* of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Prototype Inheritance

- Goal: create different objects that share inherited properties
  - Simulate class inheritance from class-based languages
- Method: use different Constructor Functions
  - Link the corresponding prototypes

Prototype Inheritance: Step-by-Step Example

- We would like to describe a generic **person**, her interests and some of her "basic" activities…
- We may add methods as properties in the constructor

```
function Person(name, age, game) {
  this.name = name;
  this.age = age;
  this.game = game;

  this.play = function() {
    console.log(`${this.name} is playing ${this.game}`);
  };
}
```

Prototype Inheritance: Step-by-Step Example

- We may add methods as properties in the prototype object of the constructor function

```
Person.prototype.showAge = function() {
  console.log(`${this.name} is ${this.age} years old`);
};
```

https://www.thecodeship.com/web-development/methods-within-constructor-vs-prototype-in-javascript/

Where to define method functions?

In the constructor function body

- Slower to create: function is re-declared for every new instance
- Faster to call: local property
- Memory per each instance
- May be redefined on a single instance
- Can access local variables (via closure)

As a prototype property

- Faster to create: declared only once
- Slower to call: must go through prototype
- Uses less memory
- Always identical for all instances
- Can’t access local variables

```
function Person(name, age, game) {
  this.play = function() {
    console.log(`${this.game}`);
  };
}
```

```
Person.prototype.showAge = function() {
  console.log(`${this.age} years old`);
};
```

Prototype Inheritance: Step-by-Step Example

- We may create new persons, and call their methods
- `.play()` is found as a property of `joe`
- `.showAge()` is found as a property of `joe`'s prototype object: `Person.prototype`

```
const joe = new Person('Joe', 25, 'chess');

joe.play();     // Joe is playing chess

joe.showAge(); // Joe is 25 years old
```

Prototype Inheritance: Step-by-Step Example

- For creating a `Student` we must inherit from `Person`
- Define a `Student` constructor function
- First, construct the `Person` part of the object
- Then, add specific attributes and methods (student-only ones)
  - Might also delete some constructed properties
- Finally link the prototype chains

Prototype Inheritance: Step-by-Step Example

- Using `Person.call` to construct a new `Person` object, but binding '`this`' to the constructed `Student`
- Adding the `school` property

```
function Student(name, age, game, school) {
  Person.call(this, name, age, game);
  this.school = school;
};
```

Prototype Inheritance: Step-by-Step Example

- Does it work?
- `.play()` is a property set in the constructor: it's inherited
- `.showAge()` is set in the prototype… but in `Person`'s prototype, not in `Student`'s

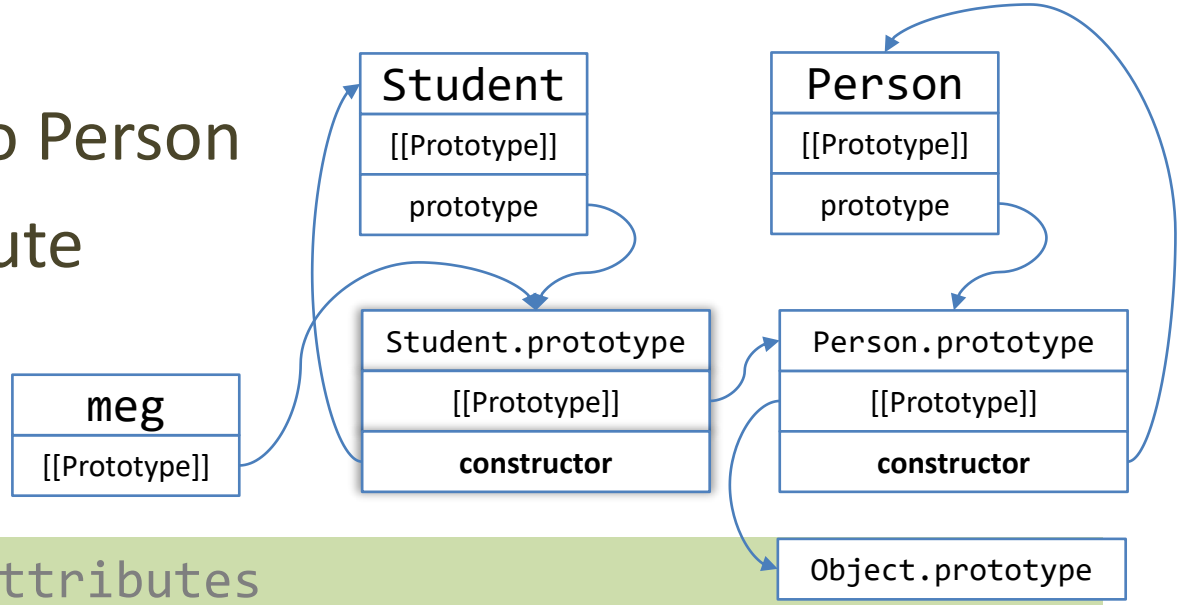
```
const meg = new Student('Meg', 21, 'tennis', 'engineering') ;

meg.play();     // Meg is playing tennis

meg.showAge(); // TypeError: meg.showAge is not a function
```

Prototype Inheritance: Step-by-Step Example

- We must link Student's prototype to Person
- And restore the 'constructor' attribute




```
// make students search for Person's attributes
Student.prototype=Object.create(Person.prototype);

// restore the correct 'constructor' property
Student.prototype.constructor=Student;
```

Prototype Inheritance: Step-by-Step Example

- We may also re-define methods on the inherited objects

```
Student.prototype.showAge = function() {
  console.log('As a student...');
  Person.prototype.showAge.call(this); // 'super'
};
```

Prototype Inheritance: Step-by-Step Example

```
'use strict';

function Person(name, age, game) {
  this.name = name;
  this.age = age;
  this.game = game;

  this.play = function() {
    console.log(`${this.name} plays ${this.game}`);
  };
}

Person.prototype.showAge = function() {
  console.log(`${this.name} is ${this.age} years`);
};

const joe = new Person('Joe', 25, 'chess');
joe.play();
joe.showAge();
```

```
function Student(name, age, game, school) {
  Person.call(this, name, age, game);
  this.school = school;
};

Student.prototype=Object.create(Person.prototype);
Student.prototype.constructor=Student;

Student.prototype.showAge = function() {
  console.log('As a student...');
  Person.prototype.showAge.call(this); // 'super'
};


const meg = new Student('Meg', 21, 'tennis', 'engineering');
meg.play();
meg.showAge();
```

JavaScript: The Definitive Guide, 7th Edition
Chapter 9. Classes

Mozilla Developer Network

- Learn web development JavaScript » Dynamic client-side scripting » Introducing JavaScript objects
- Web technology for developers » JavaScript » JavaScript reference » Classes

Modular JS programming

CLASSES

Classes

- Classes are primarily *syntactical* sugar over JavaScript's existing prototype-based inheritance
  - included from ES6
- They are special functions, based on the `class` keyword
- Two ways to define a class:
  - **class declaration**
  - **class expression**
- An object can be instantiated with the `new` keyword

Class **Declaration**

- Classic way to define a class:
  - `class` + chosen name of the class
- Class declarations are not hoisted
  - you cannot instantiate a class before declaring it
    - you should not, in any case!

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

Class **Expression**

- Another way to define a class, with two variants:
  - *named*
  - *unnamed*
- The name given to a (named) class expression is local to the class body
  - and accessed through the class' name property
  - it is "myRectangle" and "Rectangle" for the example
- Like class declarations, they are <u>not</u> hoisted

```
// named
let Rectangle = class myRectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Class Body

- The class body is always executed in **strict mode**
- Each class can have only one `constructor()`
  - a constructor can use the `super` keyword to call the constructor of the super class
- Classes can have
  - prototype methods
  - static methods

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

Prototype Methods

- Several types of prototype methods exist
- The syntax for a method is:
  - `methodName() { /* method body */ }`
  - it adds a property named `methodName` to the class and sets the value of that property to the specified function
  - you use this with *objects*, too

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
const square = new Rectangle(10, 10);
console.log(square.calcArea());
```

Prototype Methods: Getters and Setters

- JavaScript defines two methods to create a *pseudo-property*
- **Getters** allow access to a property that returns a dynamically computed or internal value
  - `get` propname()
- **Setters** are used to execute a function whenever a specified property is attempted to be changed
  - `set` propname()
- It is not possible to simultaneously have
  - A getter bound to a property and have that property hold a value
  - A setter on a property that holds an actual value

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get perimeter() {
    return this.calcPerimeter();
  }
  // Setter
  set perimeter(perimeter) {
    this.height = perimeter/2 - this.width;
  }
  // Method
  calcPerimeter() {
    return 2*(this.height + this.width);
  }
}
const square = new Rectangle(10, 10);
square.perimeter = 100;
console.log(square.perimeter);
```

Static Methods

- The `static` keyword defines a static method for a class
- Static methods are called without instantiating their class and cannot be called through a class instance
- The ‘this’ keyword may not be used inside static methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Static method
  static isWider(a, b) {
    return (a.width > b.width)? a: b;
  }
}
const s = new Rectangle(10, 15);
const r = new Rectangle(20, 30);
console.log(Rectangle.isWider(s, r));
```

Subclassing and Super Class Calls

- The `extends` keyword is used to create a class as a child of another class
  - it works with "super classes" defined as construction functions, too
- The `super` keyword is used to call corresponding methods of super class
  - *not* only the constructor!
  - not *only* from the constructor!

```
class Person {
  constructor(first, last, age, gender, interests) {
    this.name = { 'first': first, 'last' : last };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }
  sleep() {
    console.log(`${this.name.first} is sleeping.`)
  }
  play() {
    console.log(`${this.name.first} is having fun.`)
  }
}

class Student extends Person {
  constructor(first, last, age, gender, interests, id) {
    super(first, last, age, gender, interests);
    this.id = id;
  }
}
```

JavaScript: The Definitive Guide, 7th Edition
Chapter 10. Modules

Mozilla Developer Network

- Web technology for developers » JavaScript » JavaScript Guide » JavaScript Modules

Modular JS programming

MODULES

Modules

- Mechanisms for splitting JavaScript programs into separate files that can be imported when needed
- Encapsulate or hide private implementation details and keep the global namespace tidy so that modules can not accidentally modify the variables, functions and classes defined by other modules
- 3 kinds of modules:

1. *Do-It-Yourself* (with classes, objects, IIFE and closures)
2. **ES6 modules** (using `export` and `import`)
    1. ECMA Standard
    2. Supported by recent browsers
    3. Supported by Node (v13+, or v12+ with flag `--experimental-modules`)
3. Node.js modules (using `require()`) – called **CommonJS**
    1. Based on closures
    2. Never standardized by ECMA, but the normal practice with Node

DIY Modules – using IIFE

- Use an IIFE to “protect” all declared variables
  - identity
- Return an object with all the “exported” properties and functions
  - fightCrime, goCivilian
- Use closures to access internal fields

```
const batman = (function () {
  let identity = "Bruce Wayne";

  return {
    fightCrime: function () {
      console.log("Cleaning Gotham");
    },
    goCivilian: function () {
      console.log("Attend events as “
        + identity);
    }
  };
})();
```

http://adripofjavascript.com/blog/drips/understanding-the-module-pattern-in-javascript.html

ES6 Modules

- A module is a JavaScript file that **exports** one or more values (objects, functions or variables), using the `export` keyword
  - each module is a piece of code that is executed once it is loaded
- Any other JavaScript module can **import** the functionality offered by another module by importing it, with the `import` keyword
- Imports and exports must be at the *top level*
- Two main kinds of exports:
  - **named** exports (several per module)
  - **default** exports (one per module)

Default Export

- Modules that only export **single values**
  - one per module
  - You are exporting a values, but not the name of the resource
- Syntax
  - `export default <value>`

```
export default str =>
str.toUpperCase();

// OTHER examples
export default {x: 5, y: 6};

export default "name";

function grades(student) {...};
export default grades;
```

Named Exports

- Modules that export **one or more values**
  - several per module
  - Exports also the names
- Syntax
  - `export <value>`
  - `export {<value>, <...>}`

```
export const name = 'Luigi';

function grades(student) {...};
export grades;

const name = 'Luigi';
const anotherName = 'Fulvio';
export { name, anotherName }
// we can also rename them...
// export {name, anotherName as
teacher}
```

Imports

- To import something exported by another module
- Syntax
  - `import package from 'module-name'`
- Imports are:
  - hoisted
  - read-only views on exports

Import From a **Default** Export

```
--- module1.js ---
export default str =>
str.toUpperCase();
```

```
--- module2.js ---
import toUpperCase from './module1.js';
// you choose the name!

// another example
import uppercase from
'/home/appweb/module1.js';

// usage of the imported function
uppercase('test');
```

Import From a Named Export

```
--- module1.js ---
const name = 'Luigi';
const anotherName = 'Fulvio';

export { name, anotherName };
```

```
--- module2.js ---
import { name, anotherName } from
'./module1.js';

// you can rename imported values, if
you want
import { name as first, anotherName as
second} from './module1.js';
// usage
console.log(first);
```

Other Imports Options

- You can import everything a module exports
  - `import * from 'module'`
- You can import a few of the exports (e.g., if `exports {a, b, c}`):
  - `import {a} from 'module'`
- You can import the default export alongside with any named exports:
  - `import default, { name } from 'module'`

ES6 modules in the browser

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

https://v8.dev/features/modules

- File extension:
  - Preferred: `.mjs` (ensure the server sets `Content-Type: text/javascript`)
  - Also accepted: `.js`
- Load in HTML:
  - `<script type="module" src="main.js"></script>`
  - Only load the "main" modules, others will be loaded by `import` statements
  - Only files loaded with `type="module"` may use `import` and `export`
  - Modules are automatically loaded in `defer` mode
  - Note: locally loading modules (`file:///`) does not work due to CORS

ES6 modules in Node.js

- Node.js started to support ES6 modules only recently
- In Node.js v12 (LTS)
  - Must use the --experimental-modules flag when launching node
  - Must use a file extension of .mjs –or– specify "type": "module" in package.json
  - https://nodejs.org/docs/latest-v12.x/api/esm.html#esm_enabling
- In Node.js v13
  - --experimental-modules is *no longer needed*
  - Must use a file extension of .mjs –or– specify "type": "module" in package.json
  - https://nodejs.org/docs/latest-v13.x/api/esm.html#esm_enabling

CommonJS Modules

- The standard module format in Node.js
- Uses the .js or .cjs extension
- Not natively supported by browsers
  - Unless you use libraries such as RequireJS (https://requirejs.org/)
- It is basically a wrapper around your module code

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

https://nodejs.org/docs/latest-v12.x/api/modules.html

The module wrapper

- Hides top-level variables (var/let/const), by scoping them to the function instead of the global scope
- `module` and `exports` parameters may be used by the developer to export values from the module
- __filename and __dirname contain the module's absolute path

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

Imports

- To import something exported by another module
- `const package = require('module-name')`
  - Looked up in node_modules
- `const myLocalModule = require('./path/myLocalModule');`
  - Looked up in a relative path from __dirname or $cwd

Exports

- Assign your exported variables by creating new properties in the object module.exports (shortcut: exports)
  - `exports.area = (r) => Math.PI * r ** 2;`
  - `exports.circumference = (r) => 2 * Math.PI * r;`
  -
```
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }
  area() {
    return this.width ** 2;
  }
};
```

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/