TWeb

Object-oriented JavaScript

Bertil Chapuis

Overview of Today's Class

- Wat (Video)
- Quiz about last week's lecture
- Correction of last week's assignment
- Object-oriented JavaScript
- Manipulating DOM objects
- Drawing in the HTML Canvas
- More about Chrome DevTools
- Introduction of next week's assignment

Wat

A lightning talk by Gary Bernhardt from CodeMash 2012

The sarcasm in this talk does not represent anyone's actual opinion. For a more serious take on software, try Destroy All Software Screencasts: 10 to 15 minutes every other week, dense with information on advanced topics like Unix, TDD, OO Design, Vim, Ruby, and Git.

Quiz

Speakup

You can answer to the following Quiz on Speakup.

http://www.speakup.info/

Room Number: **XXXXX**

Once connected, answer to the first test question.

Question 1

Cochez la ou les affirmations correcte(s) à propos de Javascript:

- JavaScript est un language compilé
- JavaScript est un language interpreté
- JavaScript est un language statique
- JavaScript est un language dynamique
- JavaScript est un language orienté objet
- JavaScript est un language orienté prototype
- Aucune affirmation correcte

❓ Question 2

Quelle est la visibilité (scope) d'une variable définie avec le mot clé var en dehors d'une fonction?

- Sa visibilité est globale
- Sa visibilité est locale
- Sa visibilité est limitée au bloque courant
- Aucune réponse correcte

❓ Question 3

Quelle est la valeur imprimée par le programme suivant?

```
console.log("PI = ${Math.PI}");
```

- `"PI = 3.141592653589793"`
- `"PI = ${Math.PI}"`
- `"PI = 3.14"`
- `"PI = ${3.141592653589793}"`
- Aucune réponse correcte

Question 4

Quelle est la valeur imprimée par le programme suivant?

```
let i = 0;
console.log(i++);
```

- `0`
- `1`
- `Error`
- `undefined`
- Aucune réponse correcte

❓ Question 5

Quelle est la valeur imprimée par le programme suivant?

```
var x = "1";
if (x == 1) {
    var x = 2;
}
console.log(x);
```

- `1`
- `"1"`
- `2`
- `"2"`
- `null`
- Aucune réponse correcte

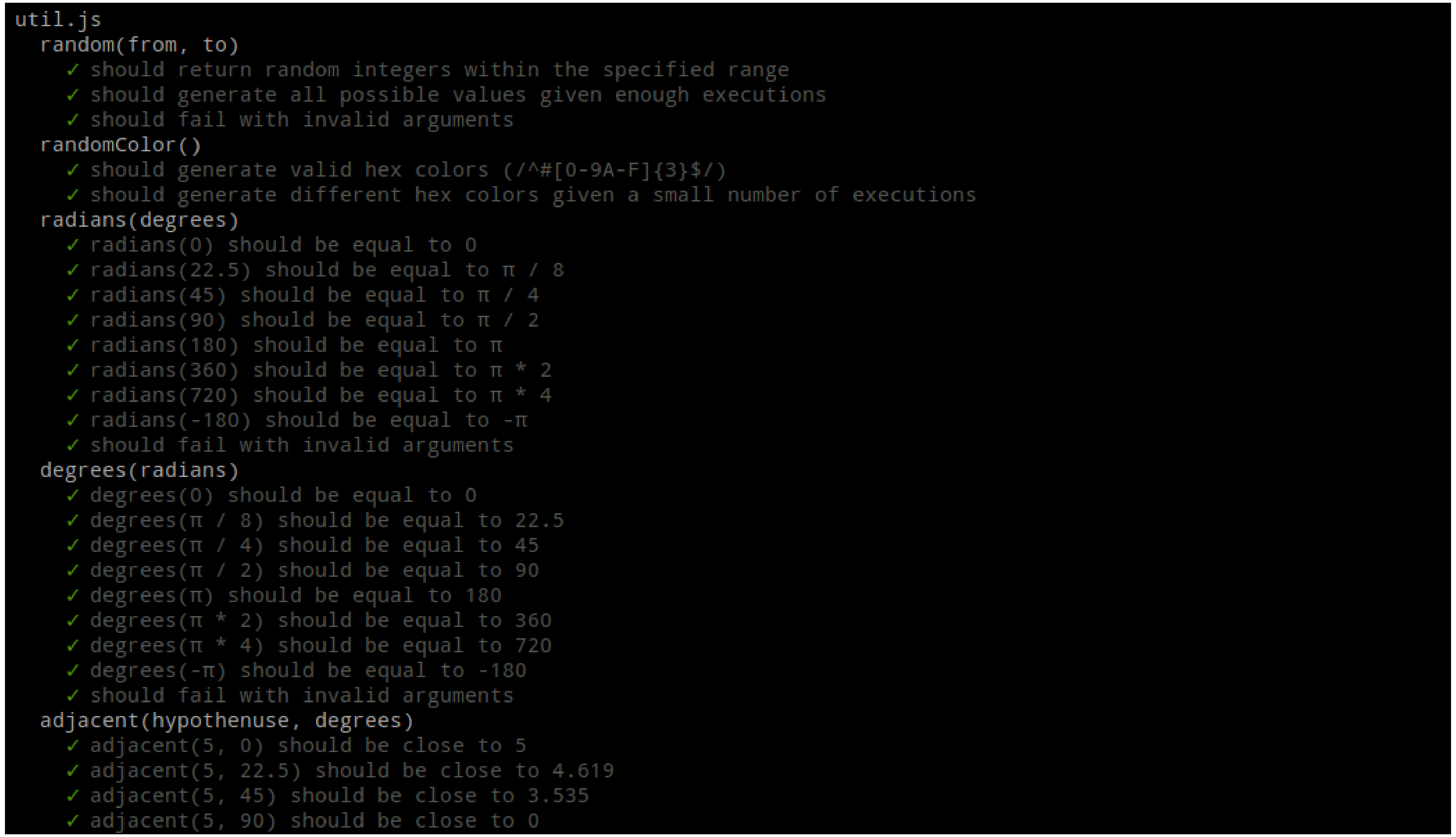
Question 6

Quelle est la valeur imprimée par le programme suivant?

```
function fun(value) {
  let v = value;
  return () => v++;
}
let f = fun(10);
for (let i = 0; i < 10; i += 2) f();
console.log(f());
```

- `10`
- `11`
- `14`
- `15`
- `16`
- `20`
- Aucune réponse correcte

Correction



```
util.js
  random(from, to)
    ✓ should return random integers within the specified range
    ✓ should generate all possible values given enough executions
    ✓ should fail with invalid arguments
  randomColor()
    ✓ should generate valid hex colors (/^#[0-9A-F]{3}$/)
    ✓ should generate different hex colors given a small number of executions
  radians(degrees)
    ✓ radians(0) should be equal to 0
    ✓ radians(22.5) should be equal to π / 8
    ✓ radians(45) should be equal to π / 4
    ✓ radians(90) should be equal to π / 2
    ✓ radians(180) should be equal to π
    ✓ radians(360) should be equal to π * 2
    ✓ radians(720) should be equal to π * 4
    ✓ radians(-180) should be equal to -π
    ✓ should fail with invalid arguments
  degrees(radians)
    ✓ degrees(0) should be equal to 0
    ✓ degrees(π / 8) should be equal to 22.5
    ✓ degrees(π / 4) should be equal to 45
    ✓ degrees(π / 2) should be equal to 90
    ✓ degrees(π) should be equal to 180
    ✓ degrees(π * 2) should be equal to 360
    ✓ degrees(π * 4) should be equal to 720
    ✓ degrees(-π) should be equal to -180
    ✓ should fail with invalid arguments
  adjacent(hypothenuse, degrees)
    ✓ adjacent(5, 0) should be close to 5
    ✓ adjacent(5, 22.5) should be close to 4.619
    ✓ adjacent(5, 45) should be close to 3.535
    ✓ adjacent(5, 90) should be close to 0
```

✋ Questions ?

Object-oriented Javascript

Recall JavaScript's Types

ECMAScript defines 7 **primitive** (Immutable) types for values.

```
3.14; // Number
true; // Boolean
"Heig-vd"; // String
undefined; // Undefined
null; // Null
9007199254740992n; // BigInt
Symbol("Symbol") // Symbol
```

ECMAScript defines a special mutable type called **object** for collections of properties (objects and array).

```
{prop: "value"}; // Object
```

In a dynamic language you don't specify the type when you declare a variable and the type of a variable can change.

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#Data_types

JS More about Object *

An object is a collection of properties, and a property is an association between a name (or key) and a value.

```
let car = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
}
```

You access the properties of an object with a simple **dot-notation** (property names: `"^[a-z]+(_[a-z]+)+$"`):

```
let car = new Object();
car.make = 'Ford';
car.model = 'Mustang';
car.year = 1969;
```

Properties of JavaScript objects can also be accessed or set using a **bracket notation**:

```
car['make'] = 'Ford';
car['model'] = 'Mustang';
car['year'] = 1969;
```

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

Consider the Array object *

The JavaScript `Array` object is a global object that is used in the construction of arrays; which are high-level, list-like **objects**.

```
var fruits = ['Apple', 'Banana', 'Pear'];
fruits[fruits.length - 1]; // Pear
fruits.1; // Uncaught SyntaxError: Unexpected number (not allowed by the dot-notation: ^[a-z]+(_[a-z]+)+$)
fruits.property = 'value'; // This is another property of the object
```

Recall that the `for...in` statement iterates over the properties of an object. **Do not use it with arrays**!

Prefer the classic `for` statement:

```
for (let i = 0; i < fruits.length; i++) console.log(fruits[i]);
```

Or the `for...of` statement:

```
for (let fruit of fruits) console.log(fruit);
```

Where do the array functions come from?

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Introducing Object.prototype *

Nearly all objects in JavaScript are instances of `Object`.

A typical object **inherits** properties (including methods) from `Object.prototype`.

Changes to the prototype are seen by all objects through **prototype chaining**.

The properties and methods can be **overridden** further along the prototype chain.

```
var fruits = ['apple', 'banana', 'pear'];

console.log(fruits.toString()); // apple,banana,pear

Array.prototype.toString = function() {
    return `Array of size ${this.length}`;
}

console.log(fruits.toString()); // Array of size 3!
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype

Method Invocation

A function stored as a property of an object is called a **method**.

When a **method** is called, `this` is bound to that object.

```
var apple = {
    color: 'red',
    toString: function() {
        return `This fruit is ${this.color}!`;
    }
}
console.log(apple.toString()); // This fruit is red!
```

Function Invocation

When a function which is not the property of an object is invoked, `this` is bound to the **global** object.

This is an error in the design of the language as it prevent the definition of helper funtions.

```
var color = 'blue';
var apple = {
    color: 'red',
    toString: function() {
        function helper() {
            return `This fruit is ${this.color}!`;
        }
        return helper();
    }
}
console.log(apple.toString()); // This fruit is blue!
```

This issue can be addressed with:

- The `apply(this, args)`, `call(this, arg, ...)` or `bind(this)` methods of a `Function` object that redefine `this`.
- The arrow function expression that do not define its own `this` and takes the one present in its scope.

Constructor Invocation

Objects inherits the properties of their prototype, hence JavaScript is class-free.

If a function is invoked with the `new` operator:

- a new object that inherits from the function's prototype is created
- the function is called and `this` is bound to the created object
- if the function doesn't return something else, `this` is returned

```
function Fruit(color) {
    this.color = color;
}

Fruit.prototype.toString = function() {
    return `This fruit is ${this.color}!`;
}

var apple = new Fruit("red");
console.log(apple.toString()); // This fruit is red!
```

JS Prototype Inheritence

With prototypes, inheritence could be achieved using the `Object.create(obj)` or `Object.assign(target, source)`. The idea consists in using an existing object as the prototype of the newly created object.

```
function Fruit(color) {
    this.color = color;
}

Fruit.prototype.toString = function() {
    return `This fruit is ${this.color}!`;
}

function Apple(color, name) {
    Fruit.call(this, color);
    this.name = name;
}

Apple.prototype = Object.assign(Apple.prototype, Fruit.prototype);

var apple = new Apple("red", "golden");
console.log(apple.toString()); // This fruit is red!
```

When a lookup fails on the apple object, it now falls back on the Fruit `prototype`.

JS The Object-oriented Syntax *

JavaScript classes are syntactical sugar over JavaScript's existing prototype-based inheritance.

The syntax allows to define constructors methods, to inherit from other classes with `extend` and to call the parent class with `super`

```
class Fruit {
    constructor(color) {
        this.color = color;
    }
    toString() {
        return `This fruit is ${this.color}!`;
    }
}
class Apple extends Fruit {
    constructor(color, name) {
        super(color);
        this.name = name;
    }
    toString() {
        return super.toString();
    }
}
var apple = new Apple("red", "golden");
console.log(apple.toString()); // This fruit is red!
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Private Properties, Getter and Setter *

The class syntax gives the ability to define static properties with `static`, private properties with `#` and to define property like getters and setters.

```
class Fruit {
    #color;
    get color() {
        return this.#color;
    }
    set color(color) {
         this.#color = color;
    }
}
let apple = new Fruit();
apple.color = 'red';
console.log(apple.color); // red
console.log(apple.#color); // Uncaught SyntaxError: Private field '#color' must be declared in an enclosing class
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

JS Static Properties and 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. Static methods are often used to create utility functions for an application.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}
let p1 = new Point(5, 5);
let p2 = new Point(10, 10);
console.log(Point.distance(p1, p2)); // 7.0710678118654755
```

Static properties must be defined outside of the class declaration:

```
Point.defaultProjection = "EPSG:4326";
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

JS Modules *

ECMAScript 6 provides a mechanisms for splitting JavaScript programs up into separate modules (files) that can be imported when needed.

```
<!-- inside index.html -->
<scriptt type="module" src="file:///home/bchapuis/Projects/github.com/tweb/slides/index.js"></scriptt>
```

```
// inside index.js
import Apple from 'apple.js';
console.log(new Apple());
```

```
// inside apple.js
import Fruit from 'fruit.js';
class Apple extends Fruit {}
export Apple;
```

```
// inside fruit.js
class Fruit {}
export Fruit;
```

Have a look at the documentation for `default` exports and `import * as Module from './modules/module.js';`.

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

Manipulating the DOM

Accessing the Document *

The `Document` interface represents any web page loaded in the browser and serves as an entry point into the web page's content, which is the DOM tree.

Access the `document` object and its properties from the Chrome DevTools

```
document;
document.location;
document.location = "https://heig-vd.ch/";
document.designMode = "true";
Document.referrer
document.writeln("Hello World!")
```

* https://developer.mozilla.org/en-US/docs/Web/API/Document

✋ Accessing the Elements of the DOM *

The `Element` interface represents the HTML elements of loaded in the DOM tree.

```
var element = document.getElementById("id");
var elements = document.getElementsByClassName(className);
var elements = document.getElementsByTagName(tagName);
```

CSS Selectors can also be used to query elements.

```
var element = document.querySelector("ul > li"); // selects the first matching element
var elements = document.querySelectorAll("ul > li"); // selects all matching elements
```

Elements can then be modified in javascript.

```
document.getElementsByClassName("remark-slide").forEach(el => el.style = "background-color: black");

element.innerHTML = "<p>Hello, World!</p>";
element.setAttribute("href", "https://www.heig-vd.ch/");
element.className;
element.classList;
element.children;
```

* https://developer.mozilla.org/en-US/docs/Web/API/Element

✋ Listening to DOM Events *

DOM 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.

```
document.onkeydown = function(event) {console.log(event);}
document.addEventListener('keydown', event => console.log(event))
```

Important DOM events include `load`, `click`, `mouseenter`, etc.

```
element.addEventListener('mouseenter', event => doSomething());
```

The propagation of an event in the DOM can be stopped programatically.

```
event.stopPropagation();
```

Event handlers can also be registered from the HTML.

```
<a href="http://www.heig-vd.ch" onclick="event.stopPropagation();">
```

* https://developer.mozilla.org/en-US/docs/Web/Events

DOM Manipulation Libraries *

Libraries such as jQuery and Zepto are intended at simplifying DOM manipulation by extending the DOM and providing helpers.

```
<scriptt src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></scriptt>
```

```
$(document).ready(function(){
  $("p").click(function(){
    $(this).hide();
  });
});
```

They use an imperative style, as they require to specify the changes in the order they should happen.

Today, frameworks that use a declarative style (such as React, Angular or Vue) are often prefered to frameworks that use an imperative style.

JS To Include or Not to Include (Libraries) *

The Peter Parker principle:

Must Read

- Thou shalt not depend on me: analysing the use of outdated JavaScript libraries on the web (NDSS 2017)
- Small world with high risks: a study of security threats in the npm ecosystem (USENIX Security 2019)

Drawing in the HTML Canvas

Initializing a Canvas *

The Canvas API provides a means for drawing graphics via JavaScript and the `canvas` element.

```html
<canvas id="canvas" width="800" height="600" />
```

```javascript
document.getElementById("canvas");
const ctx = canvas.getContext('2d');

// setting context properties
ctx.strokeStyle = 'blue';
ctx.fillStyle = 'green';

// clearing the canvas
ctx.clearRect(0, 0, 100, 100);
```

* https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

JS Drawing in the Canvas *

Writing some text:

```
ctx.fillText("test", 30, 10)
```

Filling a rectangle:

```
ctx.fillRect(10, 10, 150, 100);
```

Drawing an arc:

```
ctx.beginPath();
ctx.arc(50, 50, 10, 0, Math.PI)
ctx.stroke();
```

Free drawing:

```
ctx.beginPath();
ctx.lineTo(20, 20);
ctx.lineTo(50, 50);
ctx.stroke();
```

* https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

Transformations in the Canvas *

Transformations enables more powerful ways to translate the origin to a different position, rotate the grid and even scale it.

Canvas states are stored on a stack:

- When the `save()` method is called, the current drawing state is pushed onto the stack.
- When the `restore()` method is called, the last saved state is popped off the stack and all saved settings are restored.

When you perform transformations on the grid to draw an object you often want to restore a prior state to draw the next object.

```
ctx.fillStyle = 'rgb(0, 0, 255, 0.4)';
ctx.save();
angle = 0;
while (angle < Math.PI/2) {
    ctx.translate(200, 200);
    ctx.rotate(Math.PI / 10);
    ctx.translate(-200, -200);
    ctx.fillRect(170, 170, 60, 60);
    angle += Math.PI / 10;
}
ctx.restore();
```

* https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Transformations

Drawing in the Canvas *

Rendering Loop and Game Loop *

The setTimeout method sets a timer which executes a function or specified piece of code once the timer expires.

The setInterval method, offered on the Window and Worker interfaces, repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

The requestAnimationFrame method tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint.

* https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Timeouts_and_intervals

✋ Questions ?