JavaScript "Tips of the Day"

These were "JavaScript Tips of the Day" in 2021 Lectures

JavaScript Tips

1. Survival Skills (use semicolons)
2. Don't pretend it's another lang
3. Functions, Scope, Closures
4. Spread Syntax
5. Literal Objects
6. Class-based Objects
7. Functions are objects
8. Traditional Object-Oriented Programming
9. Beware of this
10. ES6 modules
11. Typing and Casts

Some others will be mixed in with THREE.js programming

Tip 1

Survival Example

Use Semicolons!

A JavaScript Survival Example

Principle:

JavaScript is designed to "keep going" in the face of problems

Design Decison:

No error if you leave out a semi-colon

If you forget a semi-colon, the compiler will guess where it is needed!

But, sometimes it guesses wrong

Survival Secret:

Use semi-colons where appropriate (to end statements)

Use an editor that reminds you when you forget

Tip 2

Do not pretend it is "some other" language

Tip 3

Understand scope, functions, closures

There is a whole separate video on this topic

Tip 4

Spread Syntax and Arguments

Spread Syntax

`pent[0]` is an array `[200,100]`

`context.moveTo()` takes 2 parameters x, and y

`context.moveTo(pent[0][0],pent[0][1])` is clunky

`context.moveTo(...pent[0])` uses the **spread operator**

Flexible Arguments

```
function f(a,b) {
    console.log(`a=${a} b=${b}`);
}
f(1,2)       // a=1 b=2
f(1)         // a=1 b=undefined
f(1,2,3)     // a=1 b=2
```

Extra arguments: **ignored**

Unbound arguments: `undefined`

`undefined` is a value

note also: template literal

Tip 5

Literal Objects

JavaScript tip of the day

```
dots = [ { "x":10, "y":20 },  { "x":-5 , "y":10 }, /* and more */ ]
```

The quotes are optional if the keys are tokens:

```
dots = [ { x:10, y:20 },  { x:-5 , y:10 }, /* and more */ ]
dot = { x:5, y:10 }
```

These make objects

```
dot.x  ===  dot["x"]
```

Tip 6

Class-based Objects

JavaScript has "class-based" objects

```
class Dot {
    constructor(x,y) {
        this.x = x;
        this.y = y;
     }
};
dot = new Dot(5,10);
dots = [ new Dot(10,20), new Dot(-5,10), /* and more */ ]
```

Act like simple objects (hastables)

```
dot["v"] = dot["x"]; dot.w = dot.y;
```

How to do objects?

- Javascript has flexible mechanisms
  - literal objects, prototype chains, method patching, ...
- "Traditional" mechanisms built on these
  - `class` instance, inheritence, ...

- literals are concise
- Traditional is simple
- Flexible can be handy, but can be confusing
  - Have good reasons for using flexibility
- Historic code uses old mechanisms

JavaScript Tip 7

Functions and Closures

A review of Tip 3

JavaScript tip of the day

`function` *defines* a function

those functions are objects

```
function a() {
    console.log("Two");
    return function() {
        console.log("Four");
    }
}
console.log("One");
let f = a();
console.log("Three");
f();
```

Closure!

`function` *defines* a function

those functions are objects

can access surrounding variables

(including parameters)

```
function a(v) {
    return function() {
        console.log(v);
    }
}
let f = a(1);
let g = a(2);
g();
```

JavaScript Tip 8

Traditional Object-Oriented Programming 2

Constructors and Methods

Class, Constructor, Method ...

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

    draw(context) {
        context.fillRect(this.x, this.y, this.height, this.width);
    }
}
```

JavaScript Tip 9

Beware of this

Beware of this!

`this` is a **keyword** not a **variable**

it does not behave like a variable - it is **not** lexically scoped

it has different meanings depending on context

W3 schools lists **6** different meanings of this!

Only use `this` when you know what it means

1. inside of methods
2. save a copy when you know what it means

This in methods

In a constructor:

`this` refers to the new (initially empty) object

In a method:

`this` refers to the object the method was called on

Except: Somethings redefine `this`

- Inner functions and event handlers
- special functions (call, apply, maybe others)

`this` is not lexical!

```
class MyButton {
    constructor(say) {
        let button = document.createElement("Button");
        document.getElementById("buttons-here").append(button);
        button.innerText = "Speak";
        this.word = say;
        button.onclick = function() {
            console.log(`Says ${this.word}`);   // undefined!
        }
    }
}
```

This and That

```
class MyButton2 {
    constructor(say) {
        let button = document.createElement("Button");
        document.getElementById("buttons-here").append(button);
        button.innerText = "Speak";
        this.word = say;
        let that=this;      // lexical variable - behaves nicely
        button.onclick = function() {
            console.log(`Says ${this.word}`);   // undefined!
            console.log(`Really says ${that.word}`);
        }
    }
}
```

Methods are (and are not) special

```
class MyClass {
  constructor(a) {
    this.a = a;
    this.a1 = function() {console.log(this.a);}
  }
  a2() { console.log(this.a);}
}
let inst = new MyClass(5);
// we can add a method to the instance - after the fact!
inst.a3 = function() {console.log(this.a);}
inst.a1();
inst.a2();
inst.a3();
```

`this` is defined at **call time**

JavaScript Tip 10

ES6 Modules

JavaScript Tip of the day: ES6 Modules

in html:

```
<script src="1-1.js" type="module" defer></script>
```

in JavaScript:

```
import { functionGallery } from "./1-curves.js";
```

in the module:

```
export function functionGallery(context, t, tangentScale) {
```

you need to pick what to import and export!

JavaScript Tip 11

Typing

Types in JavaScript

Javascript is **dynamically typed** - it figures out the types at run time

- variables can hold different types

```
let x;        // x = undefined
x=5;          // x is an integer
x="five";     // x is now a string
x={v:5};      // x is now an object
```

- functions can return different types

```
function intOrString(){
    return Math.random() > .5 ? 1 : "tails";
}
```

What does this return?

```
let canvas = document.getElementById("mycanvas");
```

- JavaScript just know it returns *something*
- If you read the documentation, you know its an `HTMLElement`
  - `getElementById` can return any element

- I wrote the HTML and know it should be a `HTMLCanvasElement`

JavaScript Tip: Unsafe Casts

Document what you know!

- a gift to your future self (and other programmers)
- tools can read comments too!

```
const canvas = ( /** @type {HTMLCanvasElement} */ document.getElementById("mycanvas"));
```

In some other languages, if the compiler doesn't know the type at compile time, it is an error. Casts are more than just documentation.

JavaScript Tip: Explicit Type Checks

What if your assumption is wrong?

Check to be sure!

```
let canvas = document.getElementById("mycanvas");
if (!(canvas instanceof HTMLCanvasElement))
    throw new Error("Canvas is not an HTML Canvas Element");
```

1. Catches errors (no element ,wrong element)
2. Need to handle error (`throw` might not be a good idea)
3. Serves as documentation of expected type
    - Good tools can infer types from it

JavaScript Tips

1. Survival Skills (use semicolons)
2. Don't pretend it's another lang
3. Functions, Scope, Closures
4. Spread Syntax
5. Literal Objects
6. Class-based Objects
7. Functions are objects
8. Traditional Object-Oriented Programming
9. Beware of this
10. ES6 modules
11. Typing and Casts

Some others will be mixed in with THREE.js programming