Review of MAD I

Created	@December 23, 2021 12:07 AM
Type	Lecture
Week	1
Lecture #	1
Lecture URL	https://youtu.be/Szn0FoMfx10
Notion URL	https://21f1003586.notion.site/Review-of-MAD-I-43965bfcc28b479ebe4b0452b499a4af

Recap

yay

- What is an app? (at least in our context)
  - An application or program used for interacting with a computer system
  - Allow the user to perform some task, useful to them, ~~*or not, I mean, social media isn't useful either*~~
- Components
  - **Backend** → Store data, processing logic, relation between data elements etc.
  - **Frontend** → User-facing views, abstract for machine interaction
  - Naturally implies a client-server or request-response type of architecture
- ***Why Web?***
  - Close to universal platform with clear client-server architecture
  - Low barrier to entry - trivial to implement simple pages and interaction
  - High degree of flexibility - possible to implement highly complex systems

Review of the Web Application Development Model

- **Presentation** → HTML for semantic content, CSS for styling
- **Logic** → Backend logic highly flexible - we used Python with Flask
- ***Application Structure***

    - Model - View - Controller
        - Good compromise between understandibility and flexibility
- ***System Architecture***
    - REST principles + sessions - how to build stateful applications over stateless protocols
    - APIs → separate data from the view
    - RESTful APIs → useful for basic understanding, but not strict adherence to REST
- ***Others***
    - Security, Validation, Logins and RBAC (Role-based access control), database and frontend choices

Moving Forward

- Advanced Frontend Development
    - Exploring JavaScript and how to use it
    - JavaScript, APIs, Markup - the JAMStack
    - Vue.js as a candidate frontend framework
- Other topics of interest
    - Asynchronous messaging, Email
    - Mobile / Standalone apps, PWA/SPA
    - Performance measurements, benchmarking, optimization
    - Alternatives to REST
    - etc.

JavaScript Origins & Overview

Created	@December 23, 2021 6:44 PM
Type	Lecture
Week	1
Lecture #	2
Lecture URL	https://youtu.be/bSqRlNBzwfs
Notion URL	https://21f1003586.notion.site/JavaScript-Origins-Overview-ad5d5315addd474d9c3401318546f0d0

JavaScript Origins

- Originally created in 1995 as a scripting language for Netscape Navigator
- Intended to be used as a “glue” language
  - Stick modules from other languages together
  - Not really meant for much code
- Primarily meant to assist “applets” in Java - hence JavaScript
  - Trademark issues, name changes
- Issues
  - Slow
  - Limited capability

Power

- Glue was useful ...
- Then Google Maps, Google Suggest etc. (~2005)
  - Pan around map, zoom in/out seamlessly - fluid user interface
  - Load only what is needed
- Described and named Ajax - Garrett 2005
  - Asynchronous JavaScript and XML
- Allowed true web applications that behave like desktop applications
- Evolved considerably since then ...

    - Much more on this approach moving forward

Standardization

- Move beyond Netscape needed
- ECMA (European Computer Manufacturers Association) - standard 262
- Subsequently called ECMAScript
    - To avoid trademark issues with Java
- In practice:
    - Language standard is called ECMAScript (versions)
    - Implementation and use → JavaScript
- Significant changes in ES6 - 2015
    - Yearly releases since then
    - “Feature readiness” oriented approach

What version to use?

- ES6 has most features of modern languages (modules, scoping, classes, etc.)
- Some older browsers may not support all of this
- Possible approaches ...
    - Ignore old browsers and ask the user to upgrade
    - Package browser along with the application → useful for limited cases, like Visual Studio Code (Electron apps)
    - Polyfills → include libraries that emulate newer functionality on older browsers
    - Compilers → BabelJS - convert new code to older compatible version
- Backend
    - Node.js, Deno → directly use JS as a scripting language like Python

Implications of JS Origins

- Ease of use given priority over performance (to start with)
- Highly tolerant of errors - fail silently
    - This makes debugging difficult
    - Strict mode → `use strict;`
- Ambiguous syntax variants
    - Automatic semicolon insertion
    - Object literals v/s code blocks `{}`
    - Function → statement or expression? This impacts parsing
- Limited I/O support → errors “logged” to “console”
- Closely integrates with presentation layer → DOM APIs
- Asynchronous processing and the Event loop - very powerful

Using JavaScript

- Not originally meant for direct scripting
    - Usually not run from command line, like Python
- Need HTML file to load the JS as a script
    - Requires browser to serve the file
    - Links and script tags, etc.
    - May not directly work when loaded as a file

- Node.js allows execution from the command line

DOM

- Document Object Model
  - Structure of the document shown on the browser
- DOM can be manipulated through JS APIs
- One of the most powerful aspects of JavaScript
- **Input** → clicks, textboxes, mouseover, ...
- **Output** → text, colours/styles, drawing, ...

References

- JavaScript for impatient programmers
- Mozilla Developer Network (MDN)
- LearnJavaScript.Online

Utilities

- Babel.js
  - Compiler converts new JS to older compatible forms
- JS Console
  - Interactive console to try out some code
- Replit
  - Complete application development

Basics of JavaScript

Created	@December 24, 2021 6:17 PM
Type	Lecture
Week	1
Lecture #	3
Lecture URL	https://youtu.be/SE-Vb1A-VY8
Notion URL	https://21f1003586.notion.site/Basics-of-JavaScript-3b3ed49ccc434362bf0f97cc638f3b72

Basic Frontend Usage

- Frontend JavaScript → must be invoked from HTML page
  - In context of a “Document”
  - Will not execute if loaded directly
- Scripting Language → no compilation step
- Loosely structured → no specific header, body etc.

Identifiers — the words of a language

Reserved words →

`await break case catch class const continue debugger default delete do else export extends finally for function if import in instanceof let new return static super switch this throw try typeof var void while with yield`

Literals (values) →

`true false null`

Others to avoid →

`enum implements package protected interface private public Infinity NaN undefined async`

Statements and Expressions

- Statement →
  - Piece of code that can be executed

```
if (...) {
  // do something
}
```

Standalone operation or side effects

- Expression →
  - Piece of code that can be executed to obtain a value to be returned

```
x = 10;
"Hello World"
```

Anywhere you need a “value” - function argument, math expression etc.

Data Types

- **Primitive data types** → built into the language
  - `undefined, null, boolean, number, string (+ bigint, symbol)`
- **Objects** →
  - Compound pieces of data
- **Functions** →
  - Can be handled like objects
  - Objects can have functions → methods
  - Functions can also have objects associated with them

JavaScript - Identifiers, Expressions, and Variables

Created	@December 23, 2021 7:06 PM
Type	Lecture
Week	1
Lecture #	4
Lecture URL	https://youtu.be/A1UMI9rfZ8Q
Notion URL	https://21f1003586.notion.site/JavaScript-Identifiers-Expressions-and-Variables-828875b63a354030a7be891cde737a8d

Comments

```
// This is a single line comment
/*
This is
a multi
line comment
*/
```

To declare a variable

```
let x = 10;
const anotherVariable = '42';

// Less used keyword to declare a variable
var doNotUseThisKeyword = True;

// We can also unicode characters as variable names, but please don't
```

To print something to the `console`

```
console.log("Hi Mom!");
```

Scope

A scope basically means "where is this particular variable visible?"

Why not to use `var`

```
var x1 = 10;
{
  console.log(x1);
  var x2 = 15;
}
console.log(x2);
```

The output will be ...

```
10
15
```

Replace the `var` with `let`

```
let x1 = 10;
{
  console.log(x1);
  let x2 = 15;
}
console.log(x2);
```

The output will be ...

```
10
ReferenceError: x2 is not defined
```

So, the `let` keyword respects the scope of a variable, meanwhile the `var` keyword does not

We can also replace the keyword `let` with `const`, it works almost the same, it's just that the variables will be constant

Strings

```
let s = "Hello";
console.log(s);
console.log(s.length);
console.log(s[0]);
console.log(s.substring(2, 4));  // Start at the begin index and go on till the end index - 1
```

The output will be ...

```
Hello
5
H
ll
```

String Templates

```
let st_temp = `${s} World`;  // s variable is used from the previous code block
console.log(st_temp);
```

The output will be ...

```
Hello World
```

Operators & Coercion

```
console.log(3 + 4);
console.log('3' + '4');
console.log('3' + 4);  // Coercion is happening from here, converting both to the same type
```

```
console.log(3 + '4');
console.log('3' * '4');
```

The output will be ...

```
7
34
34
34
12
```

Loose equality

```
console.log(3 == 4);
console.log(3 == 3);
console.log('3' == 3);  // Coercion is applied here, converted to the same type
console.log(undefined == null);
```

The output will be ...

```
false
true
true
true
```

Strict Equality

```
console.log('3' === 3);
console.log(undefined === null);
```

The output will be ...

```
false
false
```

JavaScript - Control Flow and Functions

Created	@December 24, 2021 6:37 PM
Type	Lecture
Week	1
Lecture #	5
Lecture URL	https://youtu.be/DOGOFom15JI
Notion URL	https://21f1003586.notion.site/JavaScript-Control-Flow-and-Functions-b9a119b64fd049668707650518a3ab45

Strings

- Source code is expected to be in Unicode
  - Most engines expect UTF-16 encoding
- String functions like length can give surprising results on non-ASCII words
- Can have variables in other languages as well
  - But it is best to avoid other languages

Non-values

- `undefined`
  - Usually implies not initialized
  - Default unknown state
- `null`
  - Explicitly set to a non-value

Very similar and may be used interchangeably in most places

We have to keep the context in mind when using for clarity of code

Operators and Comparisons

- Addition, subtraction, etc.
  - Numbers, Strings

- Coercion
  - Convert to similar type where operation is defined
  - Can lead to problems — it needs care
- Comparison
  - Loose equality → `==` tries to coerce
  - Strict equality → `===` no coercion
- Important for iteration, conditions

Variables and Scoping

- Any non-reserved identifier can be used as a “placeholder” or “variable”
- Scope →
  - Should the variable be visible everywhere in all scripts or only in a specific area?
  - Namespaces used to limit scope
- `let, const` are used for declaring variables
  - Unlike Python, variables must be declared first
  - Unlike C, their data type need not be declared
  - `var` was originally used for declaring variables, but it has function level scope — so avoid it

`let` and `const`

- `const` → declares an immutable object
  - Value cannot be changed once assigned
  - But only within the scope
- `let` → variable that can be updated

Control Flow

- Conditional execution →
  - `if, else`
- Iteration →
  - `for, while`
- Change in flow →
  - `break, continue`
- Choice →
  - `switch`

Example → Conditions

```
let x = 3;

if (x == 5) {
  console.log("The number was 5");
} else {
  console.log("The number was not 5");
}
```

Example → Iteration

```
for(let x = 0; x < 10; ++x) {
  console.log(x);
}
```

`in` operator gets the index of the elements in an array

```
const v = [10, 20, 30, 40, 50];

for(const x in v) {
  console.log(x);
}
```

So, the output will be ...

```
0
1
2
3
4
```

`of` operator gets the values of the array

```
const v = [10, 20, 30, 40, 50];

for(const x of v) {
  console.log(x);
}
```

The output will be ...

```
10
20
30
40
50
```

Functions

- Reusable block of code
- Can take parameters or arguments and perform computation
- Functions are themselves objects that can be assigned

Function Notation

- Regular declaration

  ```
  function add(x, y) {
    return x + y;
  }
  ```

- Named variable

  ```
  let add = function(x, y) {
    return x + y;
  }
  ```

- Arrow function

  ```
  let add = (x, y) => x + y;
  ```

Anonymous functions and IIFEs

IIFE → *Immediately Invoked Function Expression*

```
let x = function() { return "hello" }  // Anonymous bound

(function() { return "hello" }())      // Declare & invoke
```

Why? Older JavaScript versions did not have good scoping rules

Avoid IIFEs in modern code → poor readability

Example → Functions

```
function add(x, y) {
  return x + y;
}

console.log(typeof(add));
console.log(add(2, 3));

// We can also add values to a function
add.v = {'a': 3, b'b: 6};
console.log(add.v);
console.log(add.v.a);
```

The output will be ...

```
function
5
{a: 3, b: 6}
3
```

Arrow notation function

```
let add1 = (x, y) => (x + y);
```

Anonymous function

```
console.log(function(x, y) { return x + y; }(3, 4));
```

Assigned to variable

```
let add2 = function(x, y) { return x + y; };
```

DOM API

Created	@December 24, 2021 7:10 PM
Type	Lecture
Week	1
Lecture #	6
Lecture URL	https://youtu.be/bu564DhPTKw
Notion URL	https://21f1003586.notion.site/DOM-API-923870995a31428c8a39da6da9827836

Interaction

- `console.log` is very limited
  - variants for error logging, etc.
  - But mostly useful only for limited form of debugging - not production use
- But JS was designed for document manipulation
- Input from DOM → mouse, text, clicks
- Output to DOM → manipulation of text, colours, etc

Example

```
// Assuming we have 2 divs with the id d1 and d2
const d1 = document.getElementById('d1');
d1.innerHTML = 'Welcome to D1';

const d2 = document.getElementById('d2');
d2.innerHTML = 'Goodbye from D2';
```

The same above code, but with some added delay ...

```
async function demo() {
  console.log("Starting ...");
  await new Promise(r => setTimeout(r, 2000));
  const d1 = document.getElementById('d1');
  d1.innerHTML = 'Welcome to D1';
  console.log("After 2s ...");
  await new Promise(r => setTimeout(r, 2000));
  const d2 = document.getElementById('d2');
  d2.innerHTML = 'Goodbye from D2';
  console.log("After 4s ...");
```

```
}

demo();
```

Handling events, `onclick` ...

```
let x = 0;
const d1 = document.getElementById('d1');
d1.innerHTML = `Click count: ${x}`;

d1.addEventListener('click', e => {
  x++;
  d1.innerHTML = `Click count: ${x}`;
  d1.style.fontSize = `${x + 10}px`;
});
```