DOM AND EVOLUTION OF JS

Since we already used DOM and in this lecture we will talk about it I will show you what is it.

The Document Object Model usually referred to as the DOM, is an essential part of making websites interactive. It is an interface that allows a programming language to manipulate the content, structure, and style of a website. JavaScript is the client-side scripting language that connects to the DOM in an internet browser.

Almost any time a website performs an action, such as rotating between a slideshow of images, displaying an error when a user attempts to submit an incomplete form, or toggling a navigation menu, it is the result of JavaScript accessing and manipulating the DOM. In this article, we will learn what the DOM is, how to work with the document object and the difference between HTML source code and the DOM.

A document object is a built-in object that has many properties and methods that we can use to access and modify websites. In order to understand how to work with the DOM, you must understand how objects work in JavaScript.

Example: document.body.style.background = 'red';

EVOLUTION OF JAVASCRIPT

Before the mid-1990s, the web was not much of a major force. No primary language, with HTML being the major means of making web pages. However in 1995, the National Center of Supercomputing Applications (NCSA) made moves to change this, they released the world’s first popular web browser. It was called the NCSA Mosaic.

To challenge the NCSA Mosaic, Marc Andreessen brought in a majority of the developers who built the NCSA Mosaic to a company called Netscape.

Marc Andreessen after the success of Netscape Navigator felt the web wasn't complete just. It lacked a primary programming language. In his terms, the web needed a "glue language".To overcome this challenge, Netscape entered into a license agreement with Sun's Microsystems the owner of a popular programming language Java. The "glue language" would not have anything in relation to Java, except a bit of similarity in syntax.

-JQuery 2006

-Node.js 2009

-Backbone, Angular - SPA

In the hope of getting adopted by programmers, Netscape passed JavaScript to the European Computer Manufacturers Association (ECMA) for standardization in 1995. This brought about ECMAScript which uses the majority of JavaScript's original syntax and served as the standard for JavaScript ever since. Therefore ECMAScript could be used in place of JavaScript.

ECMAScript 6 is also known as ES6 and ECMAScript 2015.

Some people call it JavaScript 6.

This chapter will introduce some of the new features in ES6.

- JavaScript `let`
- JavaScript `const`
- JavaScript Arrow Functions
- JavaScript Classes

- Default parameter values
- `Array.find()`
- `Array.findIndex()`

let Example

```
var x = 10;

// Here x is 10

{

  let x = 2;

  // Here x is 2

}

// Here x is 10
```

const is same as let but it can’t be reasigned

JavaScript Arrow Functions

One of the most heralded features in modern JavaScript is the introduction of arrow functions, sometimes called 'fat arrow' functions, utilizing the new token `=>`.

These functions two major benefits - a very clean concise syntax and more intuitive scoping and `this` binding.

Those benefits have sometimes led to arrow functions being strictly preferred over other forms of function declaration.

Arrow Function Syntax

Arrow functions have a single overarching structure, and then an number of ways they can be simplified in special cases.

The core structure looks like this:

```
(argument1, argument2, ... argumentN) => {
  // function body
}
```

A list of arguments within parenthesis, followed by a 'fat arrow' (`=>`), followed by a function body.

This is very similar to traditional functions, we just leave off the `function` keyword and add a fat arrow after the arguments.

However, there are a number of ways to 'sugar' this up that make arrow functions *dramatically* more concise for simple functions.

First, if the function body is a single expression, you can leave off the brackets and put it inline. The results of the expression will be returned by the function. For example:

CLASSES

Whenever a new function is created in javascript, Javascript engine by default adds a prototype property to it, this property is an object and we call it “prototype object”. By default, this prototype object has a constructor property which points back to our function, and another property __proto__ which is an object, have a look at the following:

A JavaScript class is a type of function. Classes are declared with the class keyword. We will use function expression syntax to initialize a function and class expression syntax to initialize a class.

A constructor is useful when you want to create multiple similar objects with the same properties and methods. It's a convention to capitalize the name of constructors to distinguish them from regular functions. Consider the following code:

Example: class Hero

SINGLE PAGE APPLICATIONS

A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[1] or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, Interaction with the single-page application often involves dynamic communication with the web server behind the scenes.

WHY WE NEEDED FRAMEWORKS

Imagine building your house and deciding to build every small material on your own. Making a wood or brick to build a house would be crazy time-consuming. The same principle we use with building websites. To fully understand each of this framework we need to understand why do they exist and what they solved.

What JS frameworks resolved

- no one javascript file
- less code
- It is not possible to write complex, efficient and easy to maintain UIs with Vanilla JavaScript.
- state and variables
- rendering faster

- keeping the UI in sync with the state is hard

1. As you can see you only have one JS file now. Can you imagine what would happen if you continue adding new stuff to it. It would become really big and hard to maintain.

2. Working with frameworks means writing less code.

3. Writing everything in the same file means more chances of having the same names for variables which is equal to errors

4.Rendering

Every talk and article about the Virtual DOM will point out that although today's JavaScript engines are extremely fast, reading from and writing to the browser's DOM is slow.

This isn't exactly true. The DOM is fast. Adding and removing DOM nodes doesn't take much more than setting a property on a JavaScript object. It's a simple operation.

What is slow, however, is the layout that browsers have to do whenever the DOM changes. Every time the DOM changes, the browser needs to recalculate the CSS, do layout, and repaint the web page. This is what takes time.

Browser makers are continually working to shorten the time it takes to repaint the screen. The biggest thing that can be done is to minimize and batch the DOM changes that make redraws necessary.

Re-render the whole component: React. When the state of a component changes it renders a DOM in memory and compares it with the existing DOM. Actually since that would be very expensive it renders a Virtual DOM and compares it with the previous Virtual DOM snapshot. Then it calculates the changes and performs the same changes to the real DOM. This process is called reconciliation.

5.State and keeping UI in the sync with the state

WHAT IS STATE:

With water, simply put it in your freezer (below 32F) and its *state* will change from liquid to a solid. If you put it on a hot stovetop (above 212F), its *state* will change from liquid to gas. All of this can be done just by *changing one value*: its temperature which means the temperature is the state of water.

State is a JavaScript object that stores a component's dynamic data and determines the component's behavior. Because the state is dynamic, it enables a component to keep track of changing the information in between renders and for it to be dynamic and interactive.

But that's not the biggest problem. The biggest problem is always **updating the UI on every state change**. Every time the state is updated there is a lot of code required to update the UI. When adding an email address in the example it takes 2 lines of code to update the state, but 13 lines to update the UI. And we made the UI as simple as possible!!

We will need to implement a lot of ad-hoc presentation code to mutate the DOM efficiently. And **if we make any minimal mistake, the UI will be out of sync from the data**: missing information, showing wrong information, or completely messed up with elements not responding to the user or even worse, triggering wrong actions (e.g. clicking on a delete button deletes the wrong item).

Why is good to use frameworks

- They are based on components;
- They have a strong community;
- They have plenty of third party libraries to deal with things;
- They have useful third party components;
- They have browser extensions that help debugging things;
- They are good for doing single page applications.

REACT.JS

- React is a JavaScript library - one of the most popular ones, with over 100,000 stars on GitHub.
- React is not a framework (unlike Angular, which is more opinionated).
- React is an open-source project created by Facebook.
- React is used to build user interfaces (UI) on the front end.
- React is the view layer of an MVC application (Model View Controller)

  One of the most important aspects of React is the fact that you can create components, which are like custom, reusable HTML elements, to quickly and efficiently build user interfaces. React also streamlines how data is stored and handled, using state and props.

-Compilers (babel, webpack)

JSX: JavaScript + XML

As you've seen, we've been using what looks like HTML in our React code, but it's not quite HTML. This is JSX, which stands for JavaScript XML.

With JSX, we can write what looks like HTML, and also we can create and use our own XML-like tags. Here's what JSX looks like assigned to a variable.

Difference between HTML & JSX

- className is used instead of class for adding CSS classes, as class is a reserved keyword in JavaScript.
- Properties and methods in JSX are camelCase - onclick will become onClick.
- Self-closing tags *must* end in a slash - e.g. <img />

JavaScript expressions can also be embedded inside JSX using curly braces, including variables, functions, and properties.

JSX is easier to write and understand than creating and appending many elements in vanilla JavaScript, and is one of the reasons people love React so much.

Only a single parent tag is allowed - show example

A JSX expression must have only one parent tag. We can add multiple tags nested within the parent element only.

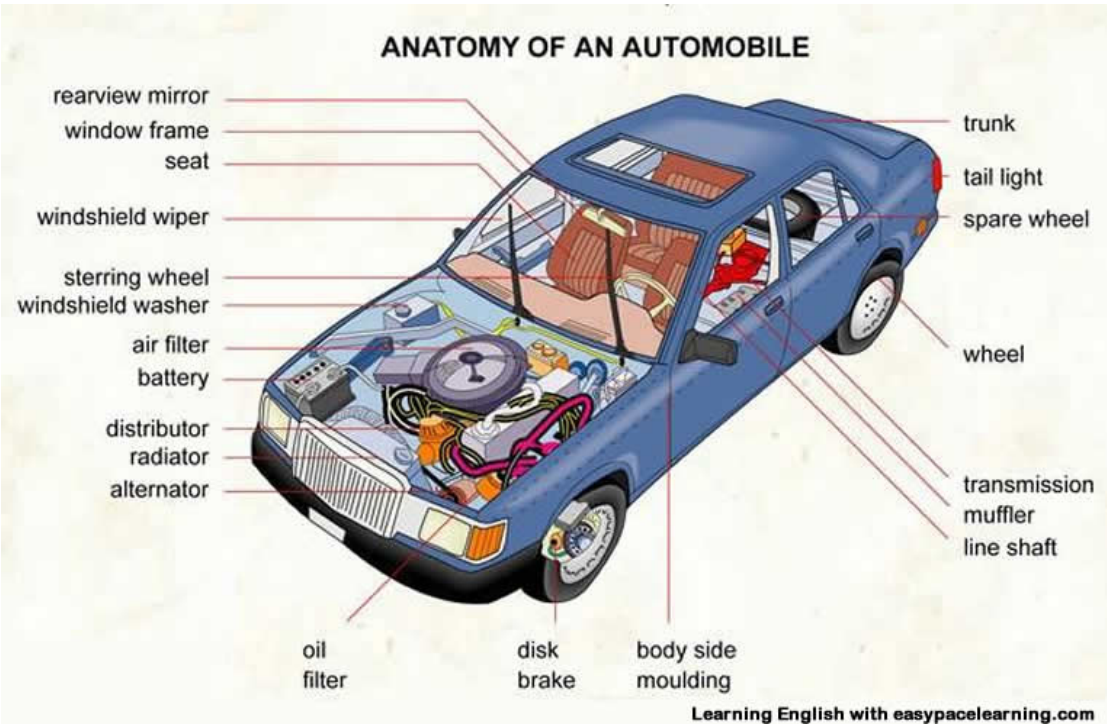
COMPONENTS

Example: Car, Greeting

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and returns HTML via a render function.

Almost everything in React consists of components, which can be class components or simple components. Components are the building blocks of any React app and a typical React app will have many of these.

Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.



ELEMENTS

Elements are smallest building blocks of react apps. An element describes what you want to see on the screen.

const element = <h1>Hello, world</h1>;

Composing Components

Example: welcome (3x)

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components. For example, we can create an App component that renders Welcome many times:

STRING CONCATINATION (ES6) - example:

```
const student = {
name: 'John Kagga',
city: 'Kampala'
};

let message = 'Hello ' + student.name + ' from ' + student.city;
console.log(message);
```

//output
Hello John Kagga from Kampala

VS

```
let message = `Hello ${student.name} from ${student.city}`;
```

//output
Hello John Kagga from Kampala

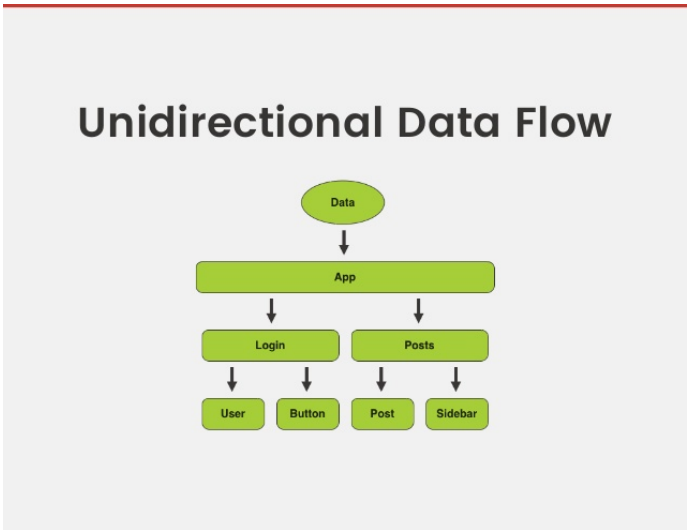
PROPS

Example: greeting, welcome (name, age, gender)

Looking at the *Greeting* component you created earlier, it is clear that it's not a very useful component to have. In real world situations, you will often need to render components dynamically depending on the situation. You, for example might want the *Greeting* component to append your application's current user's name to the end of the greeting to have output like **"Hello Steve"** as opposed to having it render **"Hello World Today!"** every time. Perhaps, you're always saying hello world, and the world never says hello back.

Props are React's way of making components easily and dynamically customisable. They provide a way of passing properties/data down from one component to another, typically from a parent to a child component (unidirectional dataflow).

It's important to note that props are **read-only** and that a component must **never** modify the props passed to it. As such, when a component is passed props as input, it should always return the same result for the same input.



Unidirectional data flow
It means that all data flows from parent to the child. It took me awhile to wrap my head around this.

Unidirectional data flow is a technique that is mainly found in functional reactive programming. It is also known as **one-way data flow,** which means the data has one, and only one way to be transferred to other parts of the application. In essence, this means child components are not able to update the data that is coming from the parent component.

In React, data coming from a parent is called **props**. The major benefit of this approach is that data flows throughout your app in a single direction, giving you better control over it.

STATE

State is a JavaScript object that stores a component's dynamic data and determines the component's behaviour. Because state is dynamic, it enables a component to keep track of changing information in between renders and for it to be dynamic and interactive.

State can only be used within a class component. If you anticipate that a component will need to manage state, it should be created as a class component and not a functional one.

You may wonder how this differs from props? We looked at props in my previous article: React: Components & Props. Recall that props are an object which holds information to control a components behavior. This sounds very similar to state, but lets see how they differ:

- Props are immutable. Once set they can't be changed
- State is observable. It can hold data that may change over time
- Props can be used in either function or class components
- State is limited to class components
- Props are set by the parent component
- State is updated by event handlers

setState()

This is why we use the built-in React method of setState(). It takes a single parameter and expects an object containing our set of values to be updated.

The method will update our state and then call the render() method to re-render the page. Therefore the proper way to update our state, is like so:

What is the lifecycle?

In general, we might define a lifecycle as birth, growth & death. And our React components follow this cycle as well: they're created (mounted on the DOM), they experience growth (by updating) and they die (unmounted from the DOM). This is the component lifecycle!

Within the lifecycle of a component, there are different phases. These phases each have their own lifecycle methods. Lets now take a look at these methods.

The Lifecycle Methods

A component's lifecycle can be broken down into four parts:

- Initialization
- Mounting
- Updating
- Unmounting

componentDidMount()

This method is called after the component is mounted on the DOM. Like componentWillMount(), it is called only once in a lifecycle. Before its execution, the render method is called. We can make API calls and update the state with the API response.