Diploma Web Appication Development: Introduction

ICT50220 Diploma of Information Technology(Front-End Web Development)

Code	Title
ICTWEB517	Create web-based programs
ICTWEB546	Validate application design against specifications

Sessions

- A session is a component of study
- Sessions may include:
  - Notes
  - Demonstrations
  - Challenges
  - Out of class activities

Introduction to State

Last session

We discussed the following 3 actions in our JS

- **Data:** retrieving or accessing the data we need
- **Template:** the reusable HTML/CSS we place our data structure in
- **Render:** taking these template and applying them to the DOM (Document Object Model)

What is State?

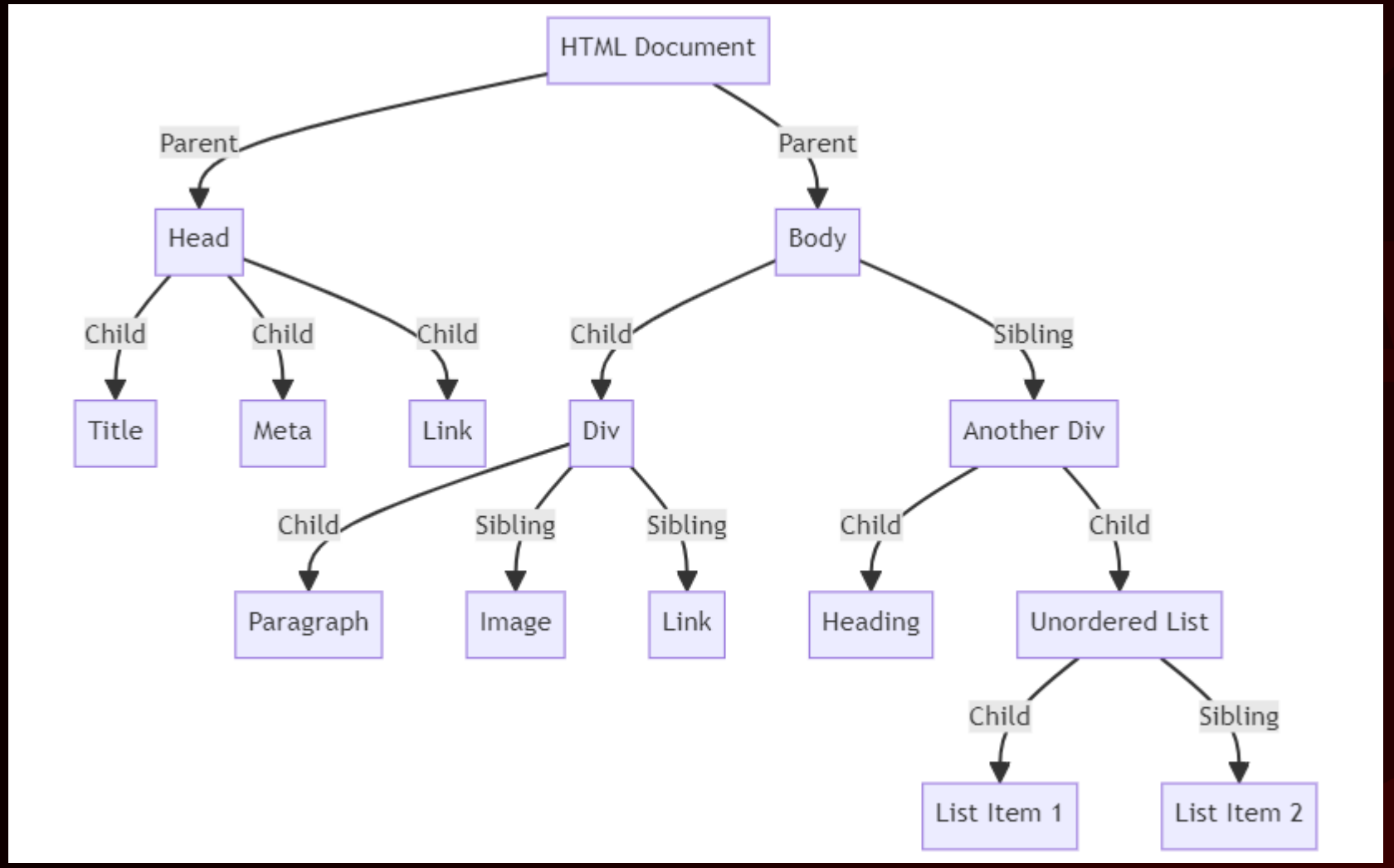
- State is just data.
- However, there is a time-bound aspect to this data.
- State is data at a particular moment in time.
- Our DOM can have multiple states when the User acts

Why do we use State?

- initially, we build web apps without using JS-based state or data.
- however, we can't edit, delete or check items off.

```
<ul id="list">
        <li>Harry Potter I</li>
        <li>Harry Potter II</li>
        <li>Harry Potter III</li>
</ul>
```

The DOM Tree



Manipulating the DOM manually?

```
var addBook = function (book) {
    if (!book || book.length < 1) return;
    var list = document.querySelector('#list');
    var listItem =document.createElement('li');
    listItem.textContent = book;
    list.appendChild(listItem);
};
```

- the **addBook()** adds items to our list
- it gets the **#list** element form the DOM.
- we use **createElement()** method to create a new list item.
- we add the book name to the **list item** with the **textContent** property.
- we then use the **appendChild()** method to inject the **list item** into the UI.

Limits of Manual Manipulation?

- for simple apps like this, manual DOM manipulation is a valid way to do things
- but things get complicated once you start adding more features and functionality.

```
var addBook = function (book) {
    if (!book || book.length < 1) return;
    var list = document.querySelector('#list');
    var listItem =document.createElement('li');
    listItem.textContent = book;
    list.appendChild(listItem);
};
```

Manipulating the DOM manually?

1. Let users create multiple lists
2. Allow list items to be deleted.
3. Support list item editing.
4. Provide a way to clear the entire list.
5. Save the data to (and loaded it from) local storage.

Consider how you would solve this list of added functionality manually?

Challenge – Stage 1…

Stage 1

```
// Challenge - Stage 1
let addBook = function (book) {
    // If there's no book to do, do nothing
    if (!book || book.length < 1) return;
    // Get the list
    let list = document.querySelector('#list');
    // Create a new list item
    let listItem = document.createElement('li');
    listItem.textContent = book;
    // Append the item to the list
    list.appendChild(listItem);
};

let book = "Harry Potter IV"

addBook(book)
```

Using state

State-based UI provides a simpler way to manage more complex web applications.

1. you store all the data in a **JavaScript** object.
2. you then use JavaScript to build the **DOM** based on the current **state** of the data.

Using state

Lets take our list app from the previous slide and convert it to a state-based UI approach.

```
<ul id="list">
        <li>Harry Potter I</li>
        <li>Harry Potter II</li>
        <li>Harry Potter III</li>
</ul>
```

Making it work

We need three things:

1. the data object
2. the template for how the UI should look based on different data states
3. a way to render the template into the DOM

```
<ul id="list">
        <li>Harry Potter I</li>
        <li>Harry Potter II</li>
        <li>Harry Potter III</li>
</ul>
```

How to create UI based…

This is, at a high level, how bigger JS frameworks like React and Vue work.

We would then create an **#app** element in HTML so we have a place to ***render*** our list of items into the DOM

```
// The data
let data = {
    books: [
                'Harry Potter I’,
                'Harry Potter II’,
                'Harry potter III’
        ]
};
```

How to create UI based…

```
// The template
let template = function (props) {
    let html =
        '<ul>' +
            props.books.map(function (book) {
                return '<li>' + book + '</li>';
                    }).join('') +
                '</ul>';
             return html;
         };
```

We can create a template function that accepts the **data** as an argument and uses it to create an HTML string for our UI.

Rendering the UI

```
// Render the template into the UI
let app = document.querySelector('#app');
app.innerHTML = template(data);
```

we render the HTML string into the UI using the **innerHTML** property.

Updating the UI

```
// Update the UI
data.books.push('Harry Potter IV');
data.books.push('Harry Potter V');
app.innerHTML = template(data);
```

- to update the UI, update the data object.
- then we render a new version of the template into the UI.
- how might this work with new data from an API call?

Update the UI

If want to remove/filter books from the list, we update the array and reset the **innerHTML** again.

```
// Manipulating the data
data.books = data.books.slice(1);
data.books = data.books.pop();
data.books = data.books.filter();

app.innerHTML = template(data);
```

Challenge – Stage 2…

Stage 2

```
// Challenge - Stage 2
// The data
let data = {
    books: ['Harry Potter I', 'Harry Potter II', 'Harry Potter III']
};

// The template
let template = function (props) {
    let html =
            '<ul>' +
                // Loop through the props array wrapping each item in a <li>
                props.books.map(function (book) {
                    return '<li>' + book + '</li>';
                    }).join('') +
                '</ul>';
            return html;
        };

// Render the template into the UI
let app = document.querySelector('#app');
app.innerHTML = template(data);

// Update the UI
data.books.push('Harry Potter IV');
data.books.push('Harry Potter V');
app.innerHTML = template(data);
```

Benefits of using components.

1. we don’t have to worry about the current state of the UI.
2. we don’t have to target elements when we remove, add, or update anything.
3. simply update the data & our template handles the rest.
4. as our apps grow, the UI is easier to manage.

The Constructor Pattern

used to create objects with properties and methods

in JavaScript functions act as constructors for creating instances

```
// Constructor function
function Person(name, age) {
    this.name = name;
    this.age = age;
  }

// Creating instances using the
constructor
let person1 = new Person("Alice", 25);
let person2 = new Person("Bob", 30);
```

Using OOP principles with the Constructor

Encapsulation - properties and methods encapsulated within the constructor

Abstraction - details of implementation abstracted away from the user

```
// Constructor function
function Circle(radius) {
    this.radius = radius;
    // Encapsulation
    this.calculateArea = function() {
        return Math.PI *
        Math.pow(this.radius, 2);
    };
}

// Creating an instance
let myCircle = new Circle(5);
// Abstraction
console.log(myCircle.calculateArea());
```

Challenge 3

Challenge 3

Create a constructor function for a "Book" with properties (title, author) and a method to display book information.

Hints

- Use the **Book** constructor to create instances.
- Call a **displayInfo** method on each instance to display the book information

Using state based components

For our book lists we need a component to use with different elements, data, and templates.

Let's start by creating a new component using a constructor pattern...

```
let MyListComponent =
function (selector, options) {
        …
};
```

Remember this principle…

We need three things:

- the data object
- the template for how the UI should look based on different data states
- a way to render the template into the DOM

Using state-based components

In our constructor, we accept two arguments:

1. the selector for the element to render our template into
2. An object with our data and template.

We save each **option** as a property using the ***this*** keyword.

```
let MyListComponent =
    function (selector, options) {
        this.element =
        document.querySelector(selector);
        this.data = options.data;
        this.template = options.template;
};
```

Getting the data & the template

Now, we ***instantiate*** a new version of our component by using the **new** operator and passing in our options.

```
// The list of books
let app = new MyListComponent('#app', {
    data: {
        books: ['Harry Potter I', 'Harry Potter II',
        'Harry Potter III']
    },
    template: function (props) {
        let html =
            '<ul>' +
                props.books.map(function (book) {
                    return '<li>' + book + '</li>';
                    }).join('') +
                '</ul>';
        return html;
    }
});
```

Using this to render the data

Because we used a constructor pattern:

- we can add a **render()** method to the **MyListComponent.prototype**
- we access to **all** of the properties set to **this** for use in the **render()** function

```
/**
 * Render a new UI
*/
MyListComponent.prototype.render =
function () {


};
```

Injecting the data into the DOM

We will use our access to the properties of **this** to:

- pass the data to the template function
- inject the resulting HTML into the selected element

```
/**
 * Render a new UI
*/
MyListComponent.prototype.render =
function () {
    this.elem.innerHTML =
      this.template(this.data);
};
```

Updating the DOM

When we want to render our UI, run the **render()** method on your specific instance.

Here we add two new books to our list.

```
// Update the UI
app.data.books.push('Harry Potter IV');
app.data.books.push('Harry Potter V');
app.render();

// app.data.books = [];
// app.render();
```

Done

Now we have a simple, reusable state-based UI component that deletes, updates and creates items in the DOM.

Consider how this approach would integrate with our NASA API project (when the API route returns a new data array)?