Building a TODO list using APIs

Event Listeners

Event listeners are interfaces that allow other objects to listen to certain events dispatched by other objects and attach a handler function to those events. The event listener is a part of the Web API.

Create and Dispatch Events

We can create custom events and invoke it in javascript. These events can be created using the DOM API in javascript which can be accessed through the **document** object.

We can create an event using the **createEvent()** method and then initialize the event with a name using the **initEvent()** method of the event. The event can be dispatched using the **dispatchEvent()** method.

Example:

```
const e = document.createEvent("Event")
e.initEvent("demo",true,true);

document.dispatchEvent(e)
```

Adding Event Listeners

Event listeners can be added using the **addEventListeners()** method.

Example:

```
const e = document.createEvent("Event")
e.initEvent("demo",true,true);

document.addEventListener("demo", (event) => {
    console.log(event)
},false)
document.dispatchEvent(e)
```

The first argument of the **addEventListener()** method is the event name and the second argument is the event handler.

The third argument denotes the order in which the event handlers will be executed when an event is dispatched on a child component and its parent component also has an event handler for the same event.

Event Capturing vs Event Bubbling

Suppose you have an element A and one of its ancestors as element B. The two elements have event handlers for the same event. Then the order in which it will be executed can be decided in two phases:

Bubbling:

In this phase the event handler of the target element is first executed and then it moves upwards and checks for event handlers for its ancestors and executes them.

Capturing:

In this phase the event handler of the top most ancestors is first checked and executed if present and it moves downwards to the target element and executes the event handlers if they are present.

By default the browser does event propagation in the bubbling phase. To do it in the capturing phase we need to set the handler capture option as true.

```
const elem = document.getElementById("elem")
```

```
elem.addEventListener("click",(e) => {
    console.log(e)
},true)
```

The event propagation can be stopped with the **stopPropagation()** method.

```
const elem = document.getElementById("elem")
elem.addEventListener("click",(e) => {
    console.log(e)
    e.stopPropagation()
},true)
```

Event Delegation

Bubbling also allows us to take advantage of **event delegation**. This concept relies on the fact that if you want some code to run when you select any one of a large number of child elements, you can set the event listener on their parent and have events that happen on them bubble up to their parent rather than having to set the event listener on every child individually.

So if you want to set the same event handler for multiple elements, you can set it on their common ancestor and the event will bubble up to the ancestor.

IIFEs

IIFEs are Immediately Invoked Function Expressions. As the name says, it is a function that is invoked as soon as it is created. It can be useful when writing a module whose variables should not be exposed to the outer environment.

Example:

```
const val = (() => {
    let counter = 0
    function increase() {
        return ++counter
    }
```

```
    function decrease() {
        return --counter
    }

    return {
        increase,
        decrease
    }
})()

console.log(val.increase()) // 0
console.log(val.decrease()) // 1
```

Array HOFs

These functions are useful for doing array operations in javascript. Each of them takes a callback function as an argument which is called for each element of the array. In each call it is passed the value, index and the current array.

forEach():

This method is used to iterate through the elements of the array. It takes a callback as an argument and it returns nothing.

```
const arr = [1,2,3,4,5]

arr.forEach((val,index,array) => {

})
```

map():

This method is used to map each element present in the array to a new element that it returns. The callback function returns an element and the method returns a new array.

```
const arr = [1,2,3,4,5]
```

```
const array = arr.map((val,index,array) => {
    return val*2
})

console.log(array) // [2,4,6,8,10]
```

filter():

This method is used to remove certain elements from the array which do not satisfy a condition. The callback function returns a boolean value and the method returns a new array

```
const arr = [1,2,3,4,5]

const array = arr.filter((val,index,array) => {
    return val%2 == 0
})

console.log(array) // [2,4]
```

reduce():

The reduce() method executes a user-supplied “reducer” callback function on each element of the array, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

The callback function takes two elements: the previous value, the current value and the current index.

```
const arr = [1,2,3,4,5]

const val = arr.reduce((prev,curr,index) => {
    return prev + curr
})

console.log(val) // 15
```