**Forms**

Form elements can be accessed by their names. Ex. Const input = form.searchInput (form name = name of element).

const form = document.forms['search'];

form.addEventListener ('submit', search, false);

function search() {

alert(' Form Submitted');

}

Now reload the page and click on the*Submit*button. You should see an alert dialog saying*Form Submitted*. After you click*OK*, the browser tries to load a nonexistent page (the URL should end in something similar to '.../search?searchInput=hello'). This is because when the event fired, our search() function was invoked, displaying the alert dialog. Then the form was submitted to the URL provided in the 'action' attribute for processing, but in this case, the URL isn't a real URL, so it doesn't go anywhere. Back-end processing isn't covered in this book, so we'll keep this as a 'dummy' URL and focus on using JavaScript to process the information instead.

We can actually stop the form from being submitted to that URL altogether by using the preventDefault() method that we saw in the last chapter. Add the following line to the search function:

function search(event) {

alert('Form Submitted');

event.preventDefault();

}

Now reload search.html and try submitting the form. You’ll see that the alert dialog still appears, but after you click*OK*, the form doesn't try to submit itself to the dummy URL.

**Disabling the submit button without event.default**

A submit button can be disable by added the disabled attribute to the <input> element:

<button type='submit' id='submit' disabled>Submit</button>

This can be changed programmatically using the disabled property of the <button> element. The following function will disable the button if an input field is empty:

function disableSubmit(event) {

if(event.target.value === ''){

document.getElementById('submit').disabled = true;

} else {

document.getElementById('submit').disabled = false;

}

}

We can apply this to the heroName field by adding the following event handler that will fire every time a key is pressed:

form.heroName.addEventListener('keyup',disableSubmit,false);

**OOP in JavaScript**

**You can check if something belongs to a object using instanceOf**

Example:

const greenDice = new redDice.constructor(10);

greenDice instanceOf Dice

<< true

Static Methods can only be called by the original class and not by instances. Pertaining to the example above, greenDice, an instance of the class Dice, could not call upon a static method. The static method can only be called by the class, in this case; Dice.

**Getters and setters**

class Turtle {

constructor(name,color) {

this.name = name;

let \_color = color;

this.setColor = color => { return \_color = color; }

this.getColor = () => \_color;

}

}

The \_color property is created as a variable inside the scope of the constructor function inside the class declaration. This makes it impossible to access outside of this scope. The getColor() and setColor() methods are known as*getter and setter methods*and they form a closure over this variable and provide controlled access to the property instead:

raph = new Turtle('Raphael','Red');

<< Turtle { name: 'Raphael', setColor: [Function], getColor: [Function] }

raph.getColor();

<< 'Red'

raph.setColor(4);

<< 4

**Extends keyword**

Makes a subclass of a class. Super calls on the parent classes’ methods

class NinjaTurtle extends Turtle {

constructor(name) {

super(name);

this.weapon = 'hands';

}

attack() { return `Feel the power of my ${this.weapon}!` }

}

**Its possible to create instance of objects with the Object.create() method.**

Example: const sam = Object.create(Human)

**Object.assign()**

Object.assign() can be used to assign properties from one object and given it another. (Only able to be copied by reference) Meaning: New reference only points to old reference when called.

**When using this with nested functions:**

Make a variable called ‘that’ and set it equal to this. Then in the new function you can reference the old ‘this’ with ‘that’.

**Ch 15 Modern Javascript Development**

* JavaScript libraries provide methods to make common tasks easier to achieve.
* Libraries can make programming much easier, but you should think carefully about whether you require a library, and which one is best for your needs.
* jQuery and Lodash are two popular libraries that provide a large number of useful and well-tested functions.
* npm and Yarn are package managers that can be used to install JavaScript packages, as well as any dependencies that they require.
* A module is a self-contained piece of code that provides functions and methods that can then be used in other files and by other modules.
* ES6 added support for modules, allowing code to be abstracted into their own self-contained files and imported into another file.
* The MVC pattern is used to organize code into distinct sections that are responsible for different elements of an application.
* Template files can be used to separate view code from JavaScript; they also enable dynamic code and programming logic to be used to generate markup.
* React and Vue.js are popular JavaScript view libraries that render components and keep track of their state.
* Minification is the process of removing any redundant characters from the code in order to reduce its file size.
* Files can be compressed on the server using the gzip compression tool.
* Webpack can be used to bundle multiple files into a single bundle, and automate common tasks such as transpiling, minifying code and running tests.
* Before code is deployed, it should be concatenated into a single file, minified and compressed. The script tag should be placed just before the closing </body> tag to ensure that all elements on the page have loaded before the script runs.