Chapter VIII – Forms

* The legacy DOM had a useful property called document.forms that returns an HTML collection of all the forms in the document in the order they appear in the markup.
* The form.submit() method will submit the form automatically.
* A form can be submitted manually by the user employing a button or input element with a type attribute of submit , or even an input element with a type attribute of image :
* <button type='submit'>Submit</button>
* <input type='submit' value='Submit'>
* <input type='image' src='button.png'>
* The form.reset() method will reset all the form controls back to their initial values specified in the HTML.

Chapter XII – Object Oriented Programming or OOP

* Object-oriented programming (OOP for short) is a style of programming that involves separating the code into objects that have properties and methods. This approach has the benefit of keeping related pieces of code encapsulated in objects that maintain state throughout the life of the program. The objects can also be reused or easily modified, as required.
* Object-oriented programming is often used to model representations of objects in the real world. There are three main concepts in OOP: encapsulation, polymorphism and inheritance.
* **Polymorphism**: My juicer isn't the only appliance I own that has an 'on' button, although the way the on button works is slightly different for each appliance. My juicer also uses the same electrical outlet as other appliances in my kitchen. I can also place various types of fruit into it and it still juices them. These examples demonstrate the concept of polymorphism: the same process can be used for different objects. In OOP, this means various objects can share the same method, but also have the ability to override shared methods with a more specific implementation.
* **Inheritance:** I’d really like the next model up from my juicer, as it can deal with more types of fruit and it’s a bit quieter. Even though it has these extra features, I’m sure that inside it uses many of the same parts that my juicer has. This demonstrates the concept of inheritance: taking the features of one object then adding some new features. In OOP, this means we can take an object that already exists and inherit all its properties and methods. We can then improve on its functionality by adding new properties and methods.
* **Classes:** Many object-oriented languages, such as Java and Ruby, are known asclass-basedlanguages. This is because they use a class to define a blueprint for an object. In my juicer example, the juicer class would represent the design of the juicer, and each juicer that’s made on the production line would be instances of that class.

JavaScript didn't have classes before ES6, and used the concept of using actual objects as the blueprint for creating more objects. This is known as aprototype-basedlanguage. In the juicer example, this might involve building an actual prototype juicer then using this prototype as the basis for making all the other juicers.

Construnction Function

Créate object using the object literal notation :

Example:

const dice = {

sides: 6,

roll() {

return Math.floor(this.sides \* Math.random() + 1)

}

}

Alternative way to créate an object is use a contructor function

Example:

const Dice = function(sides=6){

this.sides = sides;

this.roll = function() {

return Math.floor(this.sides \* Math.random() + 1)

}

}

The keyword this is used to represent the object that will be returned by the constructor function. In the previous example, we use it to set the sides property to the argument that is provided to the constructor function, or 6, if no argument is provided. It also adds a method called roll() , which returns a random number from 1 up to the number of sides the dice has.

We can now create aninstanceof the dice constructor function using the new operator.

const redDice = new Dice();

<< Dice { sides: 6, roll: [Function] }

Each new object that’s created using this function will inherit the properties and methods defined in the function. This means that redDice will have a sides property and roll() method.

* JavaScript contains a number of built-in constructor functions such as Object , Array , and Function that can be used to create objects, arrays and functions instead of literals.
* ES6 introduced the newclass declarationsyntax that does exactly the same thing as a constructor function, but looks much similar to writing a class in a class-based programming language. Here is the dice example again, using a class declaration:

class Dice {

constructor(sides=6) {

this.sides = sides;

}

roll() {

return Math.floor(this.sides \* Math.random() + 1)

}

}

To créate a new instance we use the new operator and we add

Example:

const blueDice = new Dice(20);

<< Dice { sides: 20 }

* All objects have a constructor property that returns the constructor function that created it, if you créate an object using the literal notation the constructor property Will return that.
* All classes and constructor functions have a prototype property that returns an object.

Example:

Turtle.prototype;

<< Turtle {}

* Finding out the prototype one way is to go via the constructor function’s prototype property example:

raph.constructor.prototype;

<< Turtle { attack: [Function], weapon: 'Hands' }

Another way is to use the Object.getPrototypeOf() method, which takes the object as a parameter:

Object.getPrototypeOf(raph);

<< Turtle { attack: [Function], weapon: 'Hands' }

* The \_\_proto\_\_ property was formalized in ES6 because it was already implemented in most browsers, and many JavaScript libraries already used it. It is not considered part of the official specification, and it’s recommended that getPrototypeOf() is used instead.

raph.\_\_proto\_\_

<< Turtle { attack: [Function], weapon: 'Hands' }

* Every object has a hasOwnProperty() method that can be used to check if a method is its own property, or is inherited from the prototype
* So what’s the difference between an object's own properties and prototype properties? Prototype properties are shared byeveryinstance of the Turtle class. This means they’ll all have a weapon property, and it will always be the same value. If we create another instance of the Turtle class, we’ll see that it also inherits a weapon property that has the same value of 'Hands':

Every time an instance of the Turtle class queries the weapon property, it will return 'Hands'. This value is the same for all the instances and only exists in one place ― as a property of the prototype. This means that it only exists in memory in one place, which is more efficient than each instance having its own value. This is particularly useful for any properties that are the same.

* The prototype object is live, so if a new property or method is added to the prototype, any instances of its class will inherit the new properties and methods automatically, even if that instance has already been created. For example, the raph object has a weapon property and attack() method that are inherited from Turtle.prototype . But the leo object that was createdbeforewe added these to the prototype will also have access to them.
* An object instance can overwrite any properties or methods inherited from its prototype by simply assigning a new value to them.
* The prototype can be used to add any new properties and methods after the class has been declared. It should be used to define any properties that will remain the same for every instance of the class.
* Object.getPrototypeOf() method to find the prototype of the instance class.
* This class declaration is similar to before, and defines properties and methods for anormalturtle. In our previous example, we then started adding more specific properties such as weapons that don't really apply to normal turtles, they are forninja turtles. Instead of polluting the Turtle class with these properties, it would be a good idea to create asub-classorchild classof the Turtle class called ninjaTurtle. This is created in a similar fashion, using a class declaration, but notice the use of the extends keyword:

Example:

class NinjaTurtle extends Turtle {

constructor(name) {

super(name);

this.weapon = 'hands';

}

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

}

* It is possible to add more methods to the prototype of JavaScript’s built-in objects — such as Number , String , and Array — to add more functionality. This practice is known asmonkey-patching, but it’s mostly frowned upon in the JavaScript community, despite it being an incredibly powerful technique (The Ruby programming community, on the other hand, generally embrace monkey-patching, so it is quite common in Ruby code examples.) .

As an example, we can add isOdd() and isEven() methods to the Number wrapper object’s prototype. These methods will then be available to number primitives:

Number.prototype.isEven = function() {

return this%2 === 0;

}

Number.prototype.isOdd = function() {

return this%2 === 1;

* Arrays are powerful objects, but seem to have some basic methods missing in JavaScript that are found in other languages. We can add a first() and last() methods that return the first and last items in the array:

Array.prototype.first = function() {

return this[0];

}

Array.prototype.last = function() {

return this[this.length -1];

}

* basic mixin functionality is provided by the Object.assign() method. This will assign to the object provided as the first argument all of the properties from any objects provided as further arguments
* ES6 introduced the for-of syntax for arrays and this does not require a nested function to be used, so this remains bound to the superman object.

Chapter XV – Modern JavaScript Development

* jQuery was released in 2006, originally as a DOM manipulation library. It has since grown much bigger, and now provides hundreds of methods for selecting nodes, as well as traversing the DOM, animation effects, Ajax and events. It also has its own testing library:QUnit.
* The jQuery library uses the $ symbol as a convenient alias for the the global jQuery object that contains all of jQuery's methods. This prevents the global scope from being polluted with any of jQuery’s methods. The $ symbol has become synonymous with jQuery, and you can confidently expect that any mention of it implies that jQuery is being used.
* This would then be imported into your main JavaScript file, main.js using the following code:

import { PI } from './pi.js';

* To import these functions into the main.js file, you’d add this line of code:

import { mean, variance } from './stats.js';

* This structure has folders for the different types of file that will be used in your project such as Javascript, CSS, Images and Tests. All your JavaScript files then go in the JS folder, all your stylesheet files go in the CSS folder, all your HTML files go in the views folder etc.

A typical folder structure based on type might look like the following:

* src
  + CSS
  + JS
  + views
  + tests
* dist
  + index.html
  + bundle.min.js
* package.json
* node\_modules
* README

Practice:

<https://codepen.io/michell17/pen/oNZbgNQ>