**FORMS**

A SEARCHING EXAMPLE:

Texto

Descripción generada automáticamente

This form has a name attribute of search, and contains two controls: an input field where a user can enter a search phrase, and a button to submit the form. The form can also be submitted by pressing Enter.

The action attribute is the URL that the form will be submitted to so it can be processed on the server side. The input field also has a name attribute of searchInput that is used to access the information inside it.

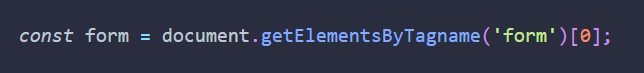
ACCESSING FORM ELEMENTS:

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. Even though there is only one form in our example, a collection will still be returned, so we have to use index notation to return the first (and only) form object, like so:

Texto

Descripción generada automáticamente con confianza baja

This is the equivalent of using the following method that we learned in chapter 6:



Instead of using a numerical index, we can use the name attribute to identify a form:

Texto

Descripción generada automáticamente

 square bracket notation can be used (this is also required if the form’s name attribute contains any invalid characters, such as spaces or dashes):

Texto

Descripción generada automáticamente

A form object also has a method called elements that returns an HTML collection of all the elements contained in the form. In this case the form contains two

Texto

Descripción generada automáticamente con confianza media

We can also access the form controls using their 'name' attributes as if it was a property of the form object.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

FORM PROPERTIES AND METHODS:

The form.submit() method will submit the form automatically. Note that submitting a form using this method won’t trigger the form submit event that's covered in the next section.

QUESTION:

SO, WHAT DOES FORM.SUBMIT() FUNCTION DOES? WHY DOESN’T IT TRIGGER THE SUBMIT METHOD?

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:

QUESTION:

HOW IS IT THAT WE CAN SUBMIT THE FORM WITH AN IMAGE? WHAT IS IT USED FOR ON A WEB PAGE?

The form.reset() method will reset all the form controls back to their initial values specified in the HTML.

A button with a type attribute of reset can also be used to do this without the need for additional scripting:

The form.action property can be used to set the action attribute of a form, so it’s sent to a different URL to be processed on the server:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

FORM EVENTS:

The focus event occurs when an element is focused on. In the case of an <input> element, this is when the cursor is placed inside the element (either by clicking or tapping on it or navigating to it using the keyboard).

The blur event occurs when the user moves the focus away from the form element.

The change event occurs when the user moves the focus away from the form element after changing it. So if a user clicks in an input field and makes no changes, and then clicks elsewhere, the change event won’t fire, but the blur event will.

SUBMITTING A FORM:

we can use JavaScript to intercept the form before it’s sent by adding a submit event listener.

Texto

Descripción generada automáticamente

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.

We can actually stop the form from being submitted to that URL altogether by using the preventDefault() method

Texto

Descripción generada automáticamente

RETRIEVING AND CHANGING VALUES FROM A FORM:

Text input element objects have a value property that can be used to retrieve the text inside the field.

Similar functionality can be produced in modern browsers using the placeholder attribute in the HTML markup.

This has slightly different behavior in that the placeholder text is not actually a value of the input field, so it won’t be submitted as the field’s value if the user fails to fill it in.

FORM CONTROLS:

Some common types of form control are:

* <input> fields, including text, passwords, check boxes, radio buttons, and file uploads
* <select> menus for drop-down lists of options
* <textarea> elements for longer text entry
* <button> elements for submitting and resetting forms

INPUT ATTRIBUTES ON HTML5:

The autofocus attribute give focus to this element when a page loads.

The placeholder attribute will insert the value provided in the input field until the user enters some text.

The maxlength attribute will limit the number of characters that can be entered in the field to the value given

INPUT FIELDS:

The type='text' attribute isn’t required (we didn’t use it in the search example as text is the default), but it is advisable to use it as it makes the intended purpose of the field explicit, helping with maintenance, readability and future-proofing.

TEXT INPUT FIELDS:

PASSWORD INPUT FIELDS:

CHECKBOX INPUT FIELDS:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Notice that all the checkbox elements have the same 'name' property of 'powers'. This means they can be accessed as an HTML collection, like so:

form.powers;

We can then iterate over this collection using a for loop to see if each checkbox was checked. Checkbox objects have a checked property that tells us if it has been checked or not. It is a boolean property, so can only have the values true or false. The value property is used to set the name of the power that can be used if the checkbox has been checked.

hero.powers = [];

*for* (*let* i=0; i < form.powers.length; i++) {

*if* (form.powers[i].checked) {

hero.powers.push(form.powers[i].value);

}}

This creates a powers property for our hero object that starts as an empty array. We then iterate over each checkbox to see if it was checked in the form. If it was, we add the 'value' property of the checkbox to the powers array using the push method.

hero.powers = [...form.powers].filter(box => box.checked).map(box => box.value);

This uses the spread operator to turn the node list into an array. This then allows us to use the filter() method that returns an array containing only the check boxes that were checked (this is because their 'checked' property will be truthy). We then chain the map() method to the end, which replaces each checkbox in the array with its 'value' property. This array is then returned and stored in the hero.powers variable.

Note that a checkbox can be set to true using JavaScript by setting its 'checked' property to true. For example, we could make the first checkbox in the list of powers appear checked with this line of code:

document.forms.hero.powers[0].checked = true;

Checkboxes can also be checked initially using the 'checked' attribute in the HTML:

<input type='checkbox' value='Flight' name='powers' checked>

RADIO BUTTON INPUT FIELDS:

All these radio buttons have the same 'name' attribute of 'category'. This is used to group them together ― only one radio button can be checked in a group that has the same name attribute. It also means we can access an HTML collection of all the radio buttons in that group using the property of the same name

form.category;

If you examine this array after the form has been submitted, it will look similar to the example below:

[input, input, input, value: "Antihero"]

The value of the radio button that was selected is stored in form.category.value (in this case it is "Antithero"). This means we can assign a category property to our hero object by adding the following code to the makeHero() function in main.js:

hero.category = form.category.value;

Each radio button has a 'checked' property that returns the boolean values true and false, depending on if it has been selected or not. It’s possible to change the 'checked' property to true using JavaScript, but because only one radio button can be checked at once, all the others with the same 'name' property will change to false. So the following line of code would check the 'antihero' radio button, but the 'hero' and 'villain' radio buttons would then be unchecked:

form.type[2].checked = true;

Radio buttons can also be checked initially using the 'checked' attribute in the HTML:

<input type='radio' name='type' value='Villain' checked>

HIDDEN INPUT FIELDS:

Hidden fields can be created using input fields with type='hidden'.

These are not displayed by the browser, but have a 'value' attribute that can contain information that is submitted with the form.

Note that the information in these fields is in no way secret, as it’s visible in the HTML, so shouldn’t be used for sensitive data.

The value of a hidden input field can be changed using JavaScript in the same was as any other input field.

QUESTION:

IS ITS BEHAVIOR SIMILAR TO VISIBILITY: HIDDEN, OR DISPLAY: NONE IN CSS?

IT IS SIMILAR TO DISPLAY: NONE

FILE INPUT FIELDS:

A file input field can be created using input fields with type='file'.

These are used to upload files, and most browsers will provide a browse button or similar that lets users select a file from their file system.

OTHER INPUT FIELDS:

There are lots of new input types included in HTML5, such as number, tel and color

SELECT DROP-DOWN LIST:

Select drop-down lists can be used to select one or more options from a list of values. The 'multiple' attribute is required if more than one option is to be selected.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Note that the 'selected' attribute can be used to set the initial value in the HTML. In this example, the blank option that provides the instructional message 'Choose a City' has this attribute, so it’s shown when the page loads.

The 'name' attribute of the <select> element is used to access it in JavaScript as a property of the form object:

form.city;

If only one item was selected, this will return a reference to that selection; otherwise a collection will be returned containing each selection.

Each selection object has a value property that’s equal to the 'value' attribute of the <option> tag that was selected.

It is also possible to find out the index of the option that has been selected, using the selectedIndex property. For example, if a user selected 'Gotham City' from the menu, form.city.selectedIndex would return 2 because it’s the third option in the list. This can then be used to access the actual text contained in the selected option:

Texto

Descripción generada automáticamente

From the example above, it should be clear that you can access the text of any option using index notation. For example, the following code returns the text from the first option:

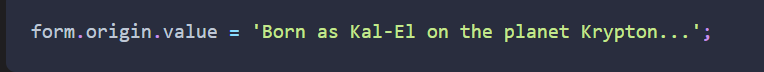
Texto

Descripción generada automáticamente con confianza media

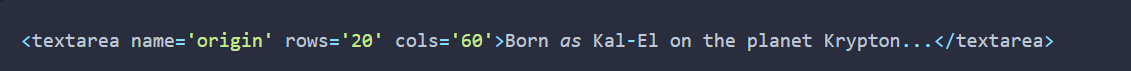
TEXT AREAS:

They work in much the same way as input fields. We access them using the 'name' attribute, and use the value property to see what text was entered.

It is also possible to change the value in the form directly:



The initial value of a text area can be set in the HTML by placing the text between the opening and closing tags:



BUTTONS:

The default type is 'submit'

The other type is simply 'button'. This doesn’t need to be inside a form element and has no default behavior. It simply creates a clickable button that can have an event listener attached to it:

There is also a type of 'menu' that can be combined with <menu>, <menuitem> and <li> tags to create a dropdown menu when it’s clicked on, although support for this is fairly patchy at present.

I NEED A HERO!

FORM VALIDATION:

* A required field is completed
* An email address is valid
* A number is entered when numerical data is required
* A password is at least a minimum number of characters

Validation can occur on the client side using JavaScript, and on the server side.

It is advisable to use both client-side and server-side validation.

JavaScript validation should be used to enhance the user experience when filling in a form by giving feedback about any errors before it’s submitted. This should then be backed up with more validation performed on the server before the data is eventually saved to a database.  it will ensure that more valid data is sent to the server, which helps to cut down the number of HTTP requests required to send the form back and forward from the server to be corrected.

The API works by simply adding relevant attributes to the form fields. For example, if a field is a required field that must be filled in, all you need to do is add a 'required' attribute to that field and the browser will take care of the rest.

We can improve the usability of the form further by giving instant feedback, instead of waiting for the form to be submitted. This can be achieved by adding the event listener directly to the input field that will fire when the user presses a key (using the keyup event). The feedback can then be inserted inside the label element of the input field, along with a class of error for more direct feedback.

Texto

Descripción generada automáticamente

For this technique to work, we actually add the error message to the HTML in the JavaScript file, regardless of whether the error has been made or not. This is done in the first five lines above, using the DOM to create a <div> element that contains the error message and has a class of 'error'. It’s then added to the <label> element using the append() method. The trick here is that the element will not be visible as it will start with a style declaration of display: none;. This will be updated dynamically as the keyup event fires.

The validateInline() function is called every time the event is triggered. We start by assigning the variable heroName to the value entered in the input field, but we also apply the toUpperCase() method to it. This will allow us to check if it begins with an 'x' or 'X' without having to check both separately.

We then use an if-else block to check if the error has been made using the startsWith() method, which will return the first letter of a string. If it starts with an 'X' then we change the style of the error element to display: block, which will make it visible.

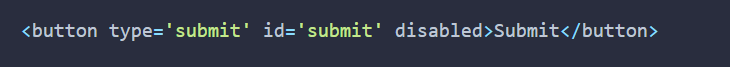
The code inside the else block is run if there is no error, so it resets the style of the error element to display: none, making the error disappear.

To make this technique work, we need to add some custom styling to the error element to make sure it isn't visible initially, and to make the message stand out.

In a real application, you might end up having to validate many different elements according to various different rules. If this is the case, it would make sense to write some more generic addError() and removeError() functions to deal with the different types of validation you might want to apply to the various elements in a form.

DISABLING THE SUBMIT BUTTON:

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

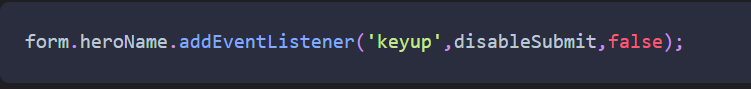


This can be changed programmatically using the disabled property of the <button> element. The following function will disable the button if an input field is empty:

Texto

Descripción generada automáticamente

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



CLASSES:

CONSTRUCTOR FUNCTIONS:

This is a function that defines the properties and methods of an object. Here is the dice example rewritten as a constructor function:

Captura de pantalla de un celular

Descripción generada automáticamente

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.

QUESTION:

HOW IS IT THAT IT RETURNS A RANDOM NUMBER FROM 1 TO 6?

**OBJECT-ORIENTED PROGRAMMING**

Encapsulation

When I use my juicer, I put the fruit into the machine, press the 'on' button and out comes the juice. I haven’t a clue how it does it—only that it makes a very loud noise! This demonstrates the concept of encapsulation: the inner workings are kept hidden inside the object and only the essential functionalities are exposed to the end user, such as the 'on' button. In OOP, this involves keeping all the programming logic inside an object and making methods available to implement the functionality, without the outside world needing to know *how* it’s done.

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.

We can now create an instance of the dice constructor function using the new operator.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The parentheses are not required when instantiating a new object using a constructor function. The following code would also achieve the same result:

Texto, Logotipo

Descripción generada automáticamente

The parentheses are required, however, if any default arguments need to be provided.

For example, if we want to create another Dice object with four sides, we would have to add 4 as an argument

This returns an object that was assigned to the variable redDice, which is said to be an instance of the Dice constructor function. We can confirm this using the instanceof operator:

Texto

Descripción generada automáticamente con confianza media

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:

Texto

Descripción generada automáticamente

Array constructor functions exhibit some strange behavior regarding the arguments supplied, however. If only one argument is given, it doesn’t create an array with that argument as the first element, as you might expect. It sets the array’s length property instead, and returns an array full of undefined!

ES6 CLASS DELCARATIONS:

ES6 introduced the new class declaration syntax 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:

Una captura de pantalla de un celular

Descripción generada automáticamente

By convention, the names of constructor functions or class declarations are capitalized, which is the convention used for classes in class-based programming languages.

To create an instance of the Dice class, the new operator is again used:

The ES6 class declarations are preferable to the constructor function syntax because they are more succinct, easier to read and all code in a class definition is implicitly in strict mode, so doesn't need the 'use strict' statement. Using ES6 class declarations also avoids a number of pitfalls associated with constructor functions. For example, an error is thrown when trying to call a class constructor without using the new operator, whereas doing the same thing with a constructor function can cause a lot of problems that are hard to track down:

Texto

Descripción generada automáticamente

THE CONSTRUCTOR PROPERTY:

All objects have a constructor property that returns the constructor function that created it:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

When an object literal is used to create a new object, we can see that in the background, the Object constructor function is being used:

Texto

Descripción generada automáticamente

We can use the constructor property to instantiate a copy of an object, without having to reference the actual constructor function or class declaration directly. For example, if we wanted to make another copy of the redDice object, but if the name of its constructor was unknown, we could use the following:

Texto

Descripción generada automáticamente

STATIC METHODS:

The static keyword can be used in class declarations to create a static method. These are sometimes called class methods in other programming languages. A static method is called by the class directly rather than by instances of the class.

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

This method is called from the Dice class like so:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

Static methods are not available to instances of the class. So, in our example, the instances of Dice such as redDice and blueDice cannot call the static description() method:

PROTOTYPAL INHERITANCE:

JavaScript uses a prototypal inheritance model. This means that every class has a prototype property that is shared by every instance of the class. So any properties or methods of a class’s prototype can be accessed by every object instantiated by that class.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

THE PROTOTYPE PROPERTY:

what if you want to augment the class with extra methods and properties after it has been created? It turns out that you can still do this using the **prototype** property of the class. This is particularly useful if you don't have access to the class declaration, but still want to add properties and methods to the class.

All classes and constructor functions have a prototype property that returns an object:

Texto

Descripción generada automáticamente con confianza media

we can add new properties by assignment:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Now if we create a new Turtle instance, we can see that it inherits the weapon property and attack() method from the Turtle.prototype object, as well as receiving the name property and sayHi() method from the class declaration:

FINDING OUT THE PROTOTYPE:

There are a number of ways to find the prototype of an object. One way is to go via the constructor function’s prototype property:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Many JavaScript engines also support the non-standard \_\_proto\_\_ property. but its use has been deprecated in favor of the setPrototypeOf() method.

Texto

Descripción generada automáticamente

Every object also has a isPrototypeOf() method that returns a boolean to check if it’s the prototype of an instance:

Texto

Descripción generada automáticamente

OWN PROPERTIES AND PROTOTYPE PROPERTIES:

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:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

So what’s the difference between an object's own properties and prototype properties? Prototype properties are shared by every instance of the Turtle class. This means they’ll all have a weapon property, and it will always be the same value.

THE PROTOTYPE IS LIVE!

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.

It is not possible to overwrite the prototype by assigning it to a new object literal if class declarations are used:

Texto

Descripción generada automáticamente

Even though it looks like the prototype has been reassigned to an empty object literal, we can see see it hasn't actually changed:

It is possible to do this if constructor functions are used, and it can cause a lot of headaches if you accidentally redefine the prototype. This is because any instances that have already been created will retain the properties and methods of the old prototype, but will not receive any of the new properties and methods that are subsequently added to the redefined prototype.

This is another reason why it’s recommended to use class declarations instead of constructor functions.

OVERWRITING PROTOTYPE PROPERTIES:

An object instance can overwrite any properties or methods inherited from its prototype by simply assigning a new value to them.

Texto

Descripción generada automáticamente

These properties will now become an 'own property' of the instance object:

Any own properties will take precedence over the same prototype property when used in methods:

Imagen que contiene Logotipo

Descripción generada automáticamente

WHAT SHOULD THE PROTOTYPE BE USED FOR?

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.

Texto

Descripción generada automáticamente

Methods are likely to be the same for all instances of a constructor, so it's fine to add methods to the prototype:

Texto

Descripción generada automáticamente

PUBLIC AND PRIVATE METHODS:

In our Ninja Turtle example, the name and weapon properties are said to be public, as can be seen if we query their value:

This means they can also be changed to any value, using assignment:

This is something you may want to avoid if your objects are public facing ― giving users or external services too much access to properties and methods could be a recipe for disaster!

Fortunately, we can use the concept of variable scope to keep some properties and methods private inside of a class declaration. This will prevent them from being accessed or changed. Instead, we will provide a getter method to return the values of any private properties.

Pantalla negra con letras blancas

Descripción generada automáticamente

The \_color property is created as a variable inside the scope of the constructor function inside the class declaration.

THAT’S WHY IT DOESN’T USE ‘THIS’, BUT ‘LET’. BECAUSE IS A VARIABLE CREATED INSIDE.

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:

This means that any private properties can only be changed in a controlled way, so we can stop certain assignments from being made by screening the data before any changes are made to a private property.

BECAUSE NOW, THEY ARE FUNCTIONS.

For example, we could insist that the color property is a string:

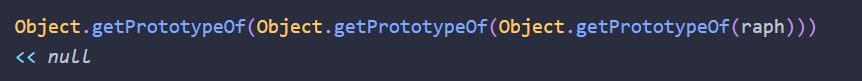
Texto

Descripción generada automáticamente

INHERITANCE:

The examples we’ve seen so far have all demonstrated inheritance by inheriting properties and methods from the prototype. But the prototype is just another object, so it also has its own prototype, which in turn has its own prototype... and so on, creating a chain of inheritance.

THE PROTOTYPE CHAIN:



Texto

Descripción generada automáticamente

This is the end of the prototype chain for our Turtle instance, and shows that all prototype chains end at the Object() constructor function.

Imagen de la pantalla de un video juego

Descripción generada automáticamente con confianza baja

THE OBJECT CONSTRUCTOR:

The prototype of the Object constructor function has a large number of methods that are inherited by all objects. The reason why the prototype appears as an empty object literal is because all of its methods are not enumerable.

ENUMERABLE PROPERTIES:

Properties of objects in JavaScript are said to be enumerable or non-enumerable. If they aren't enumerable, this means they will not show up when a for-in loop is used to loop through an object’s properties and methods.

There is a method called propertyIsEnumerable() that every object has (because it’s a method of Object.prototype) that can be used to check if a property is enumerable. We can see in the following example that the eat() method we created earlier is enumerable (in fact, all properties and methods that are created by assignment are enumerable):

Texto

Descripción generada automáticamente

INHERITANCE USING EXTENDS:

A class can inherit from another class using the extends keyword in a class declaration.

weapons that don't really apply to normal turtles, they are for ninja turtles. Instead of polluting the Turtle class with these properties, it would be a good idea to create a sub-class or child class of the Turtle class called ninjaTurtle.

Texto

Descripción generada automáticamente

Inside the child class declaration, the keyword super refers to the parent class, and can be used to access any properties and call any methods of the parent class. In the example above we use it to call the constructor function of the Turtle class.

POLYMORPHISM:

The concept of polymorphism means that different objects can have the same method, but implement it in different ways.

It’s often a useful exercise to override the toString() method using the prototype, so something more meaningful is displayed. For example, we could edit the Turtle() class declaration so it includes a more descriptive toString() method:

Texto

Descripción generada automáticamente

ADDING METHODS TO BUILT-IN OBJECTS:

This practice is known as **monkey-patching**, but it’s mostly frowned upon in the JavaScript community,

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:

Captura de pantalla con la imagen de una pantalla

Descripción generada automáticamente

Texto

Descripción generada automáticamente

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:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Another useful method that arrays lack is a decent delete() method. There is the delete operator that we met in Chapter 3, but the way this works is not very intuitive as it leaves a value of null in place of the item that’s removed. In that chapter, we saw that it’s possible to remove an item completely from an array using the splice() method. We can use this to create a new method called delete() that removes an item from the array at the index provided:

Texto

Descripción generada automáticamente

A useful example of monkey-patching is to add support for methods that are part of the specification, but not supported natively in some browsers. An example is the trim() method, which is a method of String.prototype, so all strings should inherit it. It removes all whitespace from the beginning and the end of strings, but unfortunately this method is not implemented in Internet Explorer version 8 or below. This can be rectified using this polyfill code that will use the built in String.prototype.trim if it exists, and if it doesn't, it monkey-patches the String prototype with the function provided (this is because of lazy evaluation when using the || operator):

Texto

Descripción generada automáticamente

While monkey-patching built-in objects can seem a good way to add extra or missing functionality, it can also add unexpected behavior. The current consensus in the JS community is that this shouldn't be done,

An alternative way to avoid causing problems is to use extends to subclass a built class and create your own class. For example, you could create your own array class by extending the built in array class, like so:

Texto

Descripción generada automáticamente

To create one of your new array objects, use the new keyword:

Texto

Descripción generada automáticamente

Now we can check that our delete() method works:

Un reloj digital en la pantalla

Descripción generada automáticamente con confianza media

An obvious problem with this is that you would have to use this more unwieldy syntax instead of array literals,

PROPERTY ATTRIBUTES AND DESCRIPTORS:

We’ve already seen that all objects are collections of key-value paired properties. It turns out that each property has a number of attributes that provide information about the property. These attributes are stored in a property descriptor, which is an object that contains values of each attribute.

All object properties have the following attributes stored in a property descriptor:

* value ― This is the value of t
* he property and is undefined by default
* writable ― This boolean value shows whether a property can be changed or not, and is false by default
* enumerable ― this boolean value shows whether a property will show up when the object is displayed in a for in loop, and is false by default
* configurable ― this boolean value shows whether you can delete a property or change any of its attributes, and is false by default.

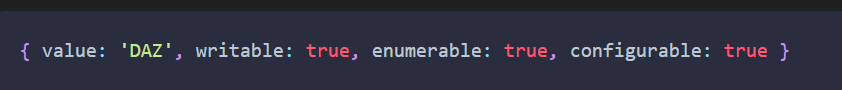
So far, we’ve just set properties by assignment, which only allows you to set the value attribute of the property. It’s also possible to set each of the property attributes by using a property descriptor.

For example, consider the following object, which has the single property of name:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

The property descriptor for the name property might look like this:



The disadvantage with this is that it can only be used to set the value attribute of the property. But it’s not possible to set the writable, enumerable, and configurable attributes in this manner.

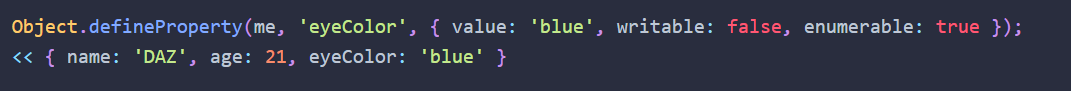
GETTING AND SETTING PROPERTY DESCRIPTORS:

The Object() constructor function has a number of methods for getting and defining property descriptors. We can see these values using the Object.getOwnPropertyDescriptor() method:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Instead of using assignment, we can add properties to an object using the Object.defineProperty() method. This provides more fine-grained control when adding new properties, as it allows each attribute to be set. The first argument is the object to which you want to add the property, followed by a property descriptor containing the attributes you want to set. Any attributes left out will take the default values:



As you can see, the object is returned with the new property added. The example above has created a property called eyeColor that is effectively read-only (because the writable attribute was set to false). If we try to change it by assignment, it will look as if it has changed:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

QUESTION:

WHY DOES IT RETURN THE NEW VALUE HERE, AND NOT BELOW? DOES IT MEAN THAT IT CHANGED?

But in reality, it hasn’t:

Texto

Descripción generada automáticamente con confianza media

GETTERS AND SETTERS:

An object property descriptor can have get() and set() methods instead of a value attribute. All objects must have one or the other, they can't have both. The get() and set() methods can be used to control how a property is set using assignment and the value that is returned when a property is queried.

They are particularly useful if a property relies on the value of another property.

For example, if we add age and retirementAge properties to the me object, we can then create a yearsToRetirement property that depends on these properties:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

USING THE GETTER

The setter also allows the age to be changed by setting the yearsToRetirement property:

Pantalla de un reloj

Descripción generada automáticamente con confianza media

USING THE SETTER

These getter and setter methods allow much more fine-grained control over how assignment works. It also means we can change the way assignment works, and use the get() method to return anything we like, regardless of what value was set using assignment.

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

* USING THE SETTER
* USING THE GETTER

The get and set property descriptors are particularly useful for controlling the getting and setting of properties in classes.

The next example shows how we can create a Dice class that uses a get function that will return a description of the number of sides, rather than just the actual number, and a set function that prohibits a non-positive number of sides to be set:

Texto

Descripción generada automáticamente

The number of sides can now be assigned in the usual way, but it will act a little differently:

Texto

Descripción generada automáticamente

TO MANAGE THE WAY THAT A PROPERTY OF AN OBJECT IS ASSIGNED AND RETURNED.

These getter and setter methods give you much more power in controlling the way property assignment works. However, they should be used sparingly and with care, as changing the expected behavior of an assignment has the potential to cause a lot of confusion.

OF COURSE. IT DOESN’T AFFECT THE USER BUT IT CAN AFFECT ME AND MY CODE, FOR EXAMPLE WHEN I USE THIS:

Captura de pantalla de un celular

Descripción generada automáticamente

“THIS.SIDES” WON’T RETURN JUST THE NUMBER BUT THAT PHRASE.

CREATING OBJECTS FROM OTHER OBJECTS:

It’s possible to avoid using classes altogether

THESE OBJECTS WILL BE THE PROTOTYPES OF OTHER OBJECTS INSTEAD OF CLASSES.

The Object() constructor function has a method called create that can be used to create a new object that is an exact copy of the object that is provided as an argument. The object that is provided as the argument acts as the prototype for the new object.

WE FIRST HAVE TO CREATE AN OBJECT LITERAL THAT WILL BE PASSED AS THE ARGUMENT. THEN, THE CREATE() FUNCTION WILL CREATE A COPY OF THAT OBJECT.

For example, we can create a Human object that will form the basis for other Human objects. This is simply created as an object literal:

Pantalla de computadora con letras

Descripción generada automáticamente

This will act as the prototype for all other Human objects. Its name is capitalized as it acts in a similar way to a class in class-based programming languages, and it’s only used to create Human objects. It should follow the same rules for prototypes that we saw earlier ― it will contain all the methods that Human objects have, as well as any properties that won’t change very often. In this case, the properties are arms and legs, and the method is walk().

We can create an instance of Human using the Object.create() method:

Texto

Descripción generada automáticamente

This will create a new object that inherits all the properties and methods from the Human object:

This is because the Human object is the prototype of the lois object:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Extra properties can then be added to each instance using assignment:

Texto

Descripción generada automáticamente

QUESTION:

WHAT IS THE PRINCIPAL DIFFERENCE BETWEEN THIS TO USING CLASSES? ADVANTAGES OR DISADVANTAGES? WHICH IS PREFERABLE?

An alternative way is to add a second argument to the Object.create() method containing properties that are to be added to the new object:

*const* jimmy = Object.create(Human, { name: { value: 'Jimmy Olsen', enumerable: true }, job: { value: 'Photographer', enumerable: true } });

The Human object will be the prototype for any objects created using it as an argument and remember that prototypes are live. This means that any changes made to the Human object will be reflected in all the objects created this way.

OBJECT-BASED INHERITANCE:

The Human object can also act like a ‘super-class’, and become the prototype of another object called Superhuman. This will have all the properties and methods that the Human object has, but with some extra methods:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

This method relies on the name and realName properties. It can be a good idea to create default values in the prototype so the method will still work. In this case, we can use names that prompt some real data to be added:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Now we can use the Superhuman object as a prototype to create more objects based on it:

Texto

Descripción generada automáticamente con confianza baja

Once a Superhuman object has been created, we can overwrite the default properties by assignment:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Now we can see that it has inherited the change() method from the Superhuman object:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

This method of adding custom properties is certainly more long-winded than using a constructor function, where the initial values are passed as an argument to the constructor function. COMPARED TO OTHER METHODS.

This can be fixed by adding a init() method to the Superhuman object that accepts initialization properties:

Texto

Descripción generada automáticamente

WE DID THIS BECAUSE WE WANTED TO CREATE A PROTOTYPE FOR SUPERHUMANS, NOT JUST HUMANS, WE CREATED THE PROTOTYPE, THEN WE ADD THE VALUES BY ASSIGNMENT, BUT WE COULD USE INIT() FUNCTION TO DO IT IN ONE STEP.

Now a new object can easily be created and initialized:

Una captura de pantalla de un celular

Descripción generada automáticamente

A new object can also be created and initialized in a single line by adding the call to the init() method at the end of the line that creates the object. This is an example of chaining (a technique that will be explained in more detail later in the chapter):

Texto

Descripción generada automáticamente

EVEN EASIER.

OBJECT PROTOTYPE CHAIN:

Creating objects from objects will create a prototype chain.

The instanceof operator will not work when objects have been created this way. It only works when using constructor functions to create objects.

Diagrama

Descripción generada automáticamente

Because of this chain, the superman object has all the properties and methods of the Human and Superhuman objects:

Texto

Descripción generada automáticamente

MIXINS:

A mixin is a way of adding properties and methods of some objects to another object without using inheritance. It allows more complex objects to be created by ‘mixing’ basic objects together.

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:

Texto

Descripción generada automáticamente

When objects are copied by assignment, they are only copied by reference. This means that another object is not actually created in memory; the new reference will just point to the old object. Any changes that are made to either objects will affect both of them. Arrays and functions are objects, so whenever they’re copied by assignment they will just point to the same object. And when one changes, they all change. This is known as making a shallow copy of an object. A deep or hard copy will create a completely new object that has all the same properties as the old object. The difference is that when a hard copy is changed, the original remains the same. But when a shallow copy is changed, the original changes too.

This affects our mixin function when we try to copy a property that is an array or object, as can be seen in this example:

*const* a = {};

*const* b = { numbers: [1,2,3] };

Object.assign(a,b);

<< { numbers: [1,2,3] }

a now has a reference to the numbers property in the b object, rather than its own copy. Any changes made to either object will affect them both:

b.numbers.push(4);

<< 4

b.numbers

<< [1,2,3,4]

a.numbers *// This has also changed*

<< [1,2,3,4]

To avoid only a shallow copy, we're going to create our own mixin() function that will assign all properties of an object to another object as a *deep* copy.

This means that *every* object will inherit this method and be able to use it to augment itself with the properties and methods from other objects.

Texto

Descripción generada automáticamente

This code looks very complicated at first glance, so let's dive into it and see what’s happening.

The first parameter is the object that we are applying the mixin to. The second parameter uses the rest parameter ...objects to allow multiple objects to be 'mixed in' at once. These will be available in the function as an array called objects.

We then use a for-of loop to iterate through each object in this array.

Next we iterate through each property in the object using the Object.keys() iterable.

The next line is the important part that ensures a deep copy. The problematic properties that are not deep copied are arrays and objects. Both of these return 'object' when the typeof operator is used. If that is the case, we need to do something different than just use Object.assign() to copy the property.

If the property is an object, we use a ternary operator to check whether it is an array or an object using the Array.isArray() method. If it is an array, then its constructor function will be Array. We create a new array literal, otherwise we create a new object literal.

Then we apply the mixin method recursively to add each property one at a time to the literal that was just created, instead of just using assignment.

And finally, the else statement states that Object.assign should still be used for any properties that are not arrays or objects because a shallow copy will work fine for those.

Let’s test this to see if it makes a deep copy:

Pantalla negra con letras blancas

Descripción generada automáticamente con confianza media

It works as expected ― all the properties from the objects b and c are mixed into the object a, and the array numbers is not copied by reference ― any changes to it only affect the object they are acted on.

The mixin() function is a particularly powerful way of dealing with objects, and has a number of uses.

USING MIXINGS TO ADD PROPERTIES:

One use for the mixin() function is to add a large number of properties to an object all at once. For example, we can instantiate a new Superhuman object, then add all its individual properties in one go, instead of one at a time, as we did earlier, while avoiding having to use the more verbose property descriptor notation:

wonderWoman = Object.create(Superhuman);

Instead of assigning each property, one at a time:

Texto

Descripción generada automáticamente

We can just mix in an object literal and add both properties at once:

Captura de pantalla de un celular

Descripción generada automáticamente

USING MIXINS TO CREATE A COPY() FUNCTION:

Another use of the mixin() function is to create a copy() method that can be used to make an exact, deep copy of an object:

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

The copy function also takes a parameter called target, which is the object to be copied. The first thing we do is create a new object based on the prototype of the object that we are copying.

The mixin() function is then used to add all the properties and methods of the object to this new object, effectively making an exact copy of itself.

We can now use this function to make exact copies of objects, as demonstrated below with a clone of the superman object:

Texto

Descripción generada automáticamente

FACTORY FUNCTION:

Our copy() function can now be used to create a factory function for superheroes. A factory function is a function that can be used to return an object.

Our factory function will be based on the Superhuman object:

Texto

Descripción generada automáticamente

This uses our copy() function to make a copy of the Superhuman object, then uses the mixin() function to augment any properties and methods of any objects that are provided as arguments. These properties and methods overwrite any default properties of the superHuman object. This allows us to provide an initialization object literal as an argument:

Texto

Descripción generada automáticamente

The createSuperhuman() function is an example of a factory function that can now be used to create as many superhuman objects as required.

QUESTION:

ISN’T IT THE SAME TO CREATE A CLASS AND OVERRIDE ITS PROPERTIES? IN WHICH CASES WE WOULD USE THIS?

Inheritance allows us to add functionality to objects by inheriting properties and methods …

USING THE MIING FUNCTION TO ADD MODULAR FUNCTIONALITY:

Inheritance allows us to add functionality to objects by inheriting properties and methods from other objects. While this is useful, it can be undesirable to create a chain of inheritance ― sometimes we just want to add properties and methods without linking the two objects together. The mixin() function lets us encapsulate properties and methods in an object, then add them to other objects without the overhead of an inheritance chain being created.

One way to think about the difference between prototypal inheritance and inheritance from mixin objects is to consider whether an object *is* something or whether it *has* something. For example, a tank *is a* vehicle, so it might inherit from a Vehicle prototype. The tank also *has a* gun, so this functionality could be added using a gun mixin object. This gives us extra flexibility, since other objects might also use a gun, but not be a vehicle, such as a soldier object, for example. The soldier object might inherit from a Human prototype and also have the gun mixin.

We can use this idea to add superpowers to our superhero objects used earlier. All the superheroes are super human, so they inherited any common traits from a Superhuman prototype. But they also have superpowers, and each superhero has a different mix of powers. This is a perfect use case for mixin objects: we can create some superpower mixin objects that can then be added to any of our superhero objects as required.

Here are some examples of superpowered mixin objects:

Texto

Descripción generada automáticamente

Now we can add the relevant superpowers to each object in a modular fashion using the mixin() function:

Texto

Descripción generada automáticamente

Now we can see they have gained some extra methods:

Texto

Descripción generada automáticamente

We can also add the mixins as an argument to the createSuperhero() factory function that we made earlier to create a superhero object with all the relevant methods from the start:



In one assignment we have created a superhero object that’s inherited all the default properties from the Superhuman object, has the correct name details and any relevant powers:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

CHAINING FUNCTIONS:

If a method returns this, its methods can be chained together to form a sequence of method calls that are called one after the other. For example, the superman object can call all three of the superpower methods at once:

Texto

Descripción generada automáticamente

This is a technique that is commonly used by a number of JavaScript libraries, most notably jQuery. It helps to make code more concise by keeping multiple method calls on the same line, and with some clever method naming it can make the calls read almost like a sentence; the Jest testing library that we used in Chapter 10 makes use of this.

A big drawback with this technique is that it can make code more difficult to debug. If an error is reported as occurring on a particular line, there is no way of knowing which method caused the error, since there are multiple method calls on that line.

It’s worth keeping in mind that if a method lacks a meaningful return value, it might as well return this so that chaining is possible.

BINDING THIS:

We saw earlier that the value of this points to the object calling a method. It allows us to create generalized methods that refer to properties specific to a particular object. Be aware of a certain problem when a function is nested inside another function, which can often happen when using methods in objects, especially ones that accept callback functions. The problem is that the value of this loses its scope, and points to the global object inside a nested function, as can be seen in this example:

Texto

Descripción generada automáticamente

The findFriends() method fails to produce the expected output because this.name is actually referencing the name property of the global window object, which has the value of undefined.

There are a couple of solutions to this problem.

USE THAT = THIS:

A common solution is to set the variable that to equal thisbefore the nested function, and refer to that in the nested function instead of this. Here is the example again, using that:

Texto

Descripción generada automáticamente

You might also see self or \_this used to maintain scope in the same way.

USE BIND(THIS):

The bind() method is a method for all functions and is used to set the value of this in the function. If this is provided as an argument to bind() while it’s still in scope, any reference to this inside the nested function will be bound to the object calling the original method:

Texto

Descripción generada automáticamente

USE FOR-OF INSTEAD OF FOREACH():

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:

Texto

Descripción generada automáticamente

USE ARROW FUNCTIONS:

Arrow functions were introduced in ES6, and one of the advantages of using them is that they don't have their own this context, so this remains bound to the original object making the function call:

Texto

Descripción generada automáticamente

For this reason, arrow functions should be used when anonymous functions are required in callbacks (and they require less typing as well!)

BORROWING METHODS FROM PROTOTYPES:

It’s possible to borrow methods from objects without having to inherit all their properties and methods. This is done by making a reference to the function that you want to borrow (that is, without parentheses so that it isn’t invoked).

For example, the batman object doesn’t have any of the superpower methods that the superman object has, but we can create a reference to them that can then be used by another object. For example, we can create a fly() function by referencing the superman object’s fly method:

*const* fly = superman.fly;

This method can now be called on another object using the call method that all functions have, and that we learned about in Chapter 11:

Texto

Descripción generada automáticamente

BORROWING ARRAY METHODS:

One of the most common uses of borrowing methods was to borrow methods from arrays in ES5. There are many *array-like* objects in JavaScript, such as the arguments object that’s available in functions, and the node lists that many of the DOM methods return. These act like arrays but are missing a lot of the methods arrays have — often it would be convenient if they had them.

For example, the arguments object can use the slice() method from the Array constructor’s prototype by assigning a variable that points to it:

*const* slice = Array.prototype.slice;

This method can then be called on the arguments object using the call() method:

slice.call(arguments, 1, 3);

The call() method takes the object that the function is to be applied to as its first argument, then the usual arguments come afterwards.

The method can also be borrowed directly from an array literal, like so:

[].slice.call(arguments, 1, 3)

An array-like object can effectively be turned into an array using the slice() method with no arguments:

*const* argumentsArray = Array.prototype.slice.call(arguments);

Most of these techniques are not needed from ES6 onwards as the Array.from() method can be used to turn an array-like object into an array:

*const* argumentsArray = Array.*from*(arguments);

Alternatively, the spread operator can be used to easily turn an array-like object into an array like so:

*const* argumentsArray = [...arguments];

COMPOSITION OVER INHERITANCE:

There are a number of benefits to object-oriented programming, but there are also some problems that come with inheritance.

Earlier in the chapter we created a Turtle class, then extended that class to create a child class called ninjaTurtle. But should the ninjaTurtle class be a child of a Turtle class or a Ninja class? Some languages use multiple inheritance (although JavaScript is not one of them), but this can cause more problems than it solves.

The 'Gorilla Banana' problem occurs when you need a method from an object, so you inherit from that object. The name comes from a quote by Joe Armstrong, the creator of the Erlang programming language:

*You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*

The problem he describes is that if an object requires a banana() method that belongs to the Gorilla class, you have to inherit the whole class to gain access to that method. But as well as the method you wanted, the object also inherits a lot of other properties and methods that are not needed, causing it to become unnecessarily bloated.

A design pattern that seeks to solve these problems is to use 'composition over inheritance'. This approach advocates creating small objects that describe single tasks or behaviors and using them as the building blocks for more complex objects. This is similar to the idea of pure functions that we discussed in the last chapter. These single-task objects are easier to test and maintain and can be combined together, using a mixin function, to create more complex objects. Composition over inheritance sees objects as building blocks that go together to make other objects rather than classes that are monolithic structures layered on top of each other.

If you do decide to use classes, it’s recommended to make them 'skinny' ― meaning they don't have too many properties and methods. Another good practice when creating classes is to keep inheritance chains short. If you have long lines of inheritance, the objects at the end of these chains will usually end up being bloated with properties and methods they don't need. It also causes problems if any of the objects in the chain need to change, as these changes will also affect other objects in the chain. A good rule of thumb is to only inherit once, keeping the inheritance chain to just two objects makes unpicking any issues far easier.

If you want to use a particular method from a class, but it has lots of properties and methods you don't need, then it would be preferable to just 'borrow' the method instead, as we saw in the last section. So, borrow the banana method from the Gorilla class instead of inheriting the whole Gorilla!

Texto

Descripción generada automáticamente

An even better approach would be to move the banana() method into a separate object then add it as a mixin to the Gorilla class, and any other objects that required it.

The author (and general all-round JavaScript genius) Eric Elliot has [a lot to say about this](https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3) that is worth reading.

MODULAR JAVASCRIPT:

COUPLING:

ES6:

For a long time, JavaScript didn't support modules, but native support for them was finally added in ES6. They allow you to keep parts of code in self-contained files.

There are a few important points about modules that are worth keeping in mind:

* All code in modules is always in strict mode without the need for 'use strict' and there is no way to opt out of this.
* A module has its own global scope, so any variables created in the top-level of a module can only be accessed within that module.
* The value of this in the top level of a module is undefined, rather than the global object.
* You can't use HTML-style comments in modules (although this isn't very common in any JavaScript program these days).

A ES6 module file is just a normal JavaScript file, but uses the keyword export to specify any values or functions that are to be made available from the module. This highlights another important fact about modules – not everything in the module needs to be used.

For example, a very simple Pi module would have the following code saved in a file called 'pi.js':

*export* *const* PI = 3.1415926;

This would then be imported into your main JavaScript file, main.js using the following code:

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

This would then allow you to use the variable PI inside the main.js file.

Functions can also be exported from a module. For example, we could create a library for our stats functions that we used earlier:

Texto

Descripción generada automáticamente

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

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

Now the mean() and variance() functions can be used in the main.js file. Notice that the square() and sum() functions are not available because they were not exported in the module. This effectively makes them private functions of the stats module.

You can be selective in which values or functions to import from the module. For example, if you only wanted to use the mean() function, you could use the following line of code instead:

*import* { mean } *from* './stats.js';

If there are lots of values and functions that need to be imported, then everything in a module file can be imported using the wildcard symbol \* along with a namespace for the imported values and functions using the following notation:

*import* \* *as* stats *from* './stats.js';

This will then import all the functions from the stats.js module and they’ll be given a namespace of stats. So, the mean function could be used as follows:

stats.mean([2,6,10]);

**Default exports** refer to a single variable, function or class in a module that can be imported without having to be explicitly named. The syntax for default exports is purposely easier to read because this is how modules were designed to be used.

The following example demonstrates how this would be done for a variable:

Texto

Descripción generada automáticamente con confianza media

The next example demonstrates exporting a single default function:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

The last example shows how to export an object as the default value:

Texto

Descripción generada automáticamente

#### **Don't Use More Than One Default Export**

Having more than one default export will result in a syntax error.

To import these default values, you would use the following code:

Texto

Descripción generada automáticamente

The big difference with default exports is that you don't need to use curly braces or make any mention of the value that is being imported, making the statement read more elegantly.

#### **Aliases**

The alias that is assigned to the imported module does not have to match its name in the actual module. For example, you could import the square function in the following way:

*import* sq *from* './square.js';

The function would then be called using sq() rather than square():

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

NODE.JS MODULES:

Node.js had already implemented modules before they were introduced in ES6, and used a slightly different notation called [Common JS modules](http://wiki.commonjs.org/wiki/Modules/1.1). At the time of writing it is proving difficult to merge the two notations in an elegant way, although it is expected that Node.js will support ES6 modules in some way in the future. Despite this, I expect you will continue to see the Common JS module pattern used by Node.js tutorials for a long time to come.

A Common JS module is created in a separate file, and the module.exports method is used to make any functions available to other files, in a similar way to ES6 modules. For example, we could create a module for squaring numbers using the following code inside a file called squareFunction.js:

module.exports = x => x \* x;

This is simply the square() function we saw earlier in the chapter written as an anonymous function that’s assigned to module.exports as if it was a variable.

To use the module, it needs to then be required inside the another JS file (or from within the Node REPL). This is done using the require() method. This takes the file that contains the module as an argument and returns the function that was exported:

*const* square = require('./squareFunction');

The function that was exported in the module is now assigned to the variable square, which is then used to call the function in the usual way:

square(6);<< 36