

Estructuras definidas por el usuario en JavaScript.



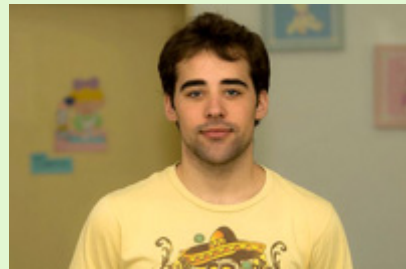
Caso práctico

Antonio ha avanzado mucho en su estudio de JavaScript, viendo los principales objetos con los que puede trabajar, sus métodos y propiedades.

Ha llegado el momento, de ver cómo puede organizar los datos que genera en su aplicación de JavaScript, de forma que le permita gestionarlos más eficientemente.

También verá cómo crear funciones, el alcance de las variables y cómo utilizarlas en su código. Y para terminar **Juan**, le va a explicar un poco más en detalle, cómo podrá crear nuevos objetos y asignarles propiedades y métodos en JavaScript.

Antonio ha comenzado a analizar más a fondo parte del trabajo que tiene que realizar, y comenta con su directora **Ada**, la posibilidad de hacer algunos bocetos de las innovaciones y mejoras que quiere aportar al proyecto. **Ada** está muy contenta con los progresos de **Antonio** y está deseando ver los bocetos.



En esta unidad de trabajo se hace una introducción a las estructuras de datos y en particular a los Arrays. Se explicará el objeto Array, cómo crear un array, recorrerlo, borrar elementos en el array y las propiedades y métodos de este objeto. También se ven los diferentes tipos de arrays.

Verás una introducción a las funciones, los parámetros y el ámbito de las variables y veremos lo qué son funciones anidadas y funciones predefinidas del lenguaje. Para terminar con la unidad veremos cómo crear objetos a medida definidos por el usuario con sus propias propiedades y métodos.



[Ministerio de Educación y Formación Profesional](#) (Dominio público)

1.- Arrays.



Caso práctico

Antonio comienza a estudiar las estructuras de datos que le permitirán almacenar la información que gestionará en sus scripts, de una forma mucho más organizada. Existen muchos tipos de estructuras de datos, pero en este caso **Antonio** profundizará en una de ellas "los arrays".

Verá cómo crearlos, recorrerlos para consultar su información, borrar elementos del array, y las propiedades y métodos disponibles para su gestión.

Terminará estudiando lo que son arrays paralelos y arrays multidimensionales, los cuales le permitirán ampliar la capacidad y ventajas que aporta un array estándar.



En los lenguajes de programación existen estructuras de datos especiales, que nos sirven para guardar información más compleja que una variable simple. En la segunda unidad vimos cómo crear variables y cómo almacenar valores (simples), dentro de esas variables.


Hay muchos tipos de estructuras de datos (listas, pilas, colas, árboles, conjuntos,...), que se pueden utilizar para almacenar datos, pero una estructura de las más utilizadas en todos los lenguajes es el array. El **array**, es como una variable o zona de almacenamiento continuo, donde podemos introducir varios valores en lugar de solamente uno, como ocurre con las variables normales.

Los arrays también se suelen denominar **matrices** o **vectores**. Desde el punto de vista lógico, una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos o más dimensiones).

Se puede considerar que todos los arrays son de una dimensión: la dimensión principal, pero los elementos de dicha fila pueden a su vez contener otros arrays o matrices, lo que nos permitiría hablar de **arrays multidimensionales** (los más fáciles de imaginar son los de una, dos y tres dimensiones). No es aconsejable utilizar arrays de más dimensiones, si nuestra lógica de programa no la requiere, ya que hacen difícil su comprensión.

Los arrays son una estructura de datos, adecuada para situaciones en las que el acceso a los datos se realiza de forma aleatoria e impredecible. Por el contrario, si los elementos pueden estar ordenados y se va a utilizar acceso secuencial sería más adecuado usar una lista, ya que esta estructura puede cambiar de tamaño fácilmente durante la ejecución de un programa.

Los arrays nos permiten guardar un montón de elementos y acceder a ellos de manera independiente. Cada elemento es referenciado por la posición que ocupa dentro del array. Dichas posiciones se llaman **índices** y siempre son correlativos.

En JavaScript cuando trabajamos con arrays se utiliza la  **indexación base-cero**: en este modo, el primer elemento del array será la componente 0, es decir tendrá el índice 0.

Los arrays se introdujeron en JavaScript a partir de la versión 1.1, es decir que en cualquier navegador moderno disponible actualmente no tendrás ningún tipo de problema para poder usar arrays.



Autoevaluación

En JavaScript los índices de los arrays comienzan en 1. ¿Verdadero o falso?

☐ Verdadero ☐ Falso

Falso

Los arrays en JavaScript comienzan con su índice en 0.

1.1.- Trabajando con Arrays.

Un array es una de las mayores estructuras de datos proporcionadas para almacenar y manipular colecciones de datos. A diferencia de otros lenguajes de programación, los arrays en JavaScript son muy versátiles, en el sentido de los diferentes tipos de datos que podemos almacenar en cada posición del array. En JavaScript un array puede contener elementos de diferentes tipos, lo que también los hace muy versátiles.



[Everaldo Coelho \(GNU/GPL\)](#)

En programación, **un array se define como una colección ordenada de datos**. Lo mejor es que pienses en un array como si fuera una tabla que contiene datos, o también como si fuera una hoja de cálculo.

JavaScript emplea un montón de arrays internamente para gestionar los objetos HTML en el documento, propiedades del navegador, etc. Por ejemplo, si tu documento contiene 10 enlaces, el navegador mantiene una tabla con todos esos enlaces. Tú podrás acceder a esos enlaces por su número de enlace (comenzando en el 0 como primer enlace). Si empleamos la sentencia de array para acceder a esos enlaces, el nombre del array estará seguido del número índice (número del enlace) entre corchetes, como por ejemplo: `document.links[0]` , representará el primer enlace en el documento.

En la unidad anterior, si recuerdas, teníamos colecciones dentro del objeto `document`. Pues bien, cada una de esas colecciones (`anchors[]`, `forms[]`, `links[]`, e `images[]`) será un array que contendrá las referencias de todas las anclas, formularios, enlaces e imágenes del documento.

A medida que vayas diseñando tu aplicación, tendrás que identificar las pistas que te permitan utilizar arrays para almacenar datos. Por ejemplo, imagínate que tienes que almacenar un montón de coordenadas geográficas de una ruta a caballo: ésto si que sería un buen candidato para emplear una estructura de datos de tipo array, ya que podríamos asignar nombres a cada posición del array, realizar cálculos, ordenar los puntos, etc. Siempre que veas similitudes con un formato de tabla, será una buena opción para usar un array.

Creación

Recorrido

Trabajando con sus elementos

Propiedades y métodos

Arrays paralelos

Arrays multidimensionales

Creación

Para crear un array, podemos utilizar varias sintaxis.

Por ejemplo si tenemos las siguientes variables:

```
let coches = [ 'Seat', 'Audi', 'BMW', 'Toyota' ];  
let numeros = [ 1, 5, 3, 9, 6, 4 ];
```

```
let diferentes = [ 'Pepe', 5, 'Juan', false ];
```

También es posible utilizar la sintaxis `new Array()` y luego ir asignando valores a los diferentes elementos, como por ejemplo:

```
let coches = new Array();  
coches[0] = 'Seat';  
coches[1] = 'Audi';  
coches[2] = 'BMW';  
coches[3] = 'Toyota';
```

En JavaScript también es posible utilizar arrays asociativos, es decir, asociaciones clave-valor. Veamos un ejemplo:

```
let edades = new Array();  
edades['Juan'] = 20;  
edades['Ana'] = 18;  
edades['Pedro'] = 25;  
console.log(edades);  
console.log(edades['Juan']);  
console.log(edades['Ana']);  
console.log(edades['Pedro']);
```

Recorrido

Para recorrer un array podemos utilizar uno de los bucles que ya conocemos. Por ejemplo, dado el siguiente array:

```
let numeros = [ 1, 5, 3, 9, 6, 4 ];
```

Podríamos utilizar un bucle `for` para recorrerlo:

```
for (let i = 0; i < numeros.length; i++) {  
  console.log(numeros[i]);  
}
```

También existe un bucle especial para recorrer arrays en JavaScript; el denominado `for-in`:

```
for (let i in numeros) {  
  console.log(numeros[i]);  
}
```

Incluso podemos utilizar otras construcciones como la función `forEach`:

```
numeros.forEach(numero => console.log(numero));
```

Es más, podemos utilizar las funciones `map`, `reduce` y `filter` para realizar algunas acciones comunes con los arrays:

```
let cuadrados = numeros.map(numero => {return numero * numero});  
console.log(cuadrados);
```

En el siguiente enlace se muestran ejemplos detallados de cómo utilizar estas nuevas características.

[Uso de las funciones `map`, `filter` y `reduce`.](#)

Trabajando con sus elementos

Una vez creado un array, siempre podremos añadir nuevos elementos de forma dinámica utilizando un índice no utilizado.

```
numeros[6] = 15;  
console.log(numeros);
```

Debemos tener en cuenta que si no añadimos el nuevo elemento al final, las posiciones intermedias se quedarán con valores indefinidos.

Podemos eliminar un elemento del array mediante la función `delete`, pero esta función no reduce el tamaño del array, simplemente borra el elemento:

```
delete numeros[6];  
console.log(numeros.length); // Muestra por consola: 7  
console.log(numeros[6]); // Muestra por consola: undefined
```

Si lo que queremos es también reducir su tamaño deberemos utilizar la función `splice`; a esta función le indicamos la posición a partir de la cual empezamos a eliminar y el

número de elementos:

```
numeros.splice(5, 2);  
console.log(numeros);           // Muestra por consola: [ 1, 5, 3, 9, 6 ]
```

Propiedades y métodos

Propiedades del objeto array:

Propiedades del objeto Array

Propiedad	Descripción
constructor	Devuelve la función que creó el prototipo del objeto array.
length	Ajusta o devuelve el número de elementos en un array.
prototype	Te permite añadir propiedades y métodos a un objeto.

Métodos del objeto array:

Métodos del objeto Array

Métodos	Descripción
concat()	Une dos o más arrays, y devuelve una copia de los arrays unidos.
join()	Une todos los elementos de un array en una cadena de texto.
pop()	Elimina el último elemento de un array y devuelve ese elemento.
push()	Añade nuevos elementos al final de un array, y devuelve la nueva longitud.
reverse()	Invierte el orden de los elementos en un array.
shift()	Elimina el primer elemento de un array, y devuelve ese elemento.
slice()	Selecciona una parte de un array y devuelve el nuevo array.
sort()	Ordena los elementos de un array.
splice()	Añade/elimina elementos a un array.
toString()	Convierte un array a una cadena y devuelve el resultado.

Métodos	Descripción
unshift()	Añade nuevos elementos al comienzo de un array, y devuelve la nueva longitud.
valueOf()	Devuelve el valor primitivo de un array.

Arrays paralelos

El usar arrays para almacenar información, aporta una gran versatilidad a las aplicaciones. Pero en algunos casos, podría ser muy útil hacer búsquedas en varios arrays a la vez, de tal forma que podamos almacenar diferentes tipos de información, en arrays que estén sincronizados.

Cuando tenemos dos o más arrays, que utilizan el mismo índice para referirse a términos homólogos, se denominan **arrays paralelos**.

Veamos el siguiente ejemplo:

```
let personajes = [ 'Bob Esponja', 'Calamardo', 'Patricio' ];
let color = [ 'amarillo', 'beige', 'rosa' ];
for (let i = 0; i < personajes.length; i++) {
  console.log(`${personajes[i]} es de color ${color[i]}`);
}
```

Arrays multidimensionales

Como ya sabemos, un array puede almacenar elementos de diferentes tipos, por lo que un elemento de un array, a su vez puede ser otro array y a eso es a lo que se conoce como arrays multidimensionales.

Veamos el mismo ejemplo anterior, pero con un array multidimensional.

```
let personajes = [
  [ 'Bob Esponja', 'amarillo'],
  [ 'Calamardo', 'beige' ],
  [ 'Patricio', 'rosa' ]
];
for (let i in personajes) {
```

```
console.log(`${personajes[i][0]} es de color ${personajes[i][1]}`);  
}
```



Autoevaluación

Un array en JavaScript siempre debe contener elementos del mismo tipo.
¿Verdadero o falso?

☐ Verdadero ☐ Falso

Falso

Como hemos visto, un array puede contener elementos de diferentes tipos.

2.- Funciones.



Caso práctico

Juan está siguiendo los progresos de **Antonio**, y le recomienda empezar a ver funciones. Las funciones son una herramienta muy potente en los lenguajes de programación, ya que le van a permitir realizar tareas de una manera mucho más organizada, y además le permitirán reutilizar un montón de código en sus aplicaciones.



Antonio verá cómo crear funciones, cómo pasar parámetros, el ámbito de las variables dentro de las funciones, cómo anidar funciones y las funciones predefinidas en JavaScript. **Antonio** está muy ilusionado con este tema, ya que a lo largo de las unidades anteriores ha estado utilizando funciones y es ahora el momento en el que realmente verá toda la potencia que le pueden aportar a sus aplicaciones.

En unidades anteriores ya has visto y utilizado alguna vez funciones. **Una función es la definición de un conjunto de acciones pre-programadas**. Las funciones se llaman a través de eventos o bien mediante comandos desde nuestro script.

En el siguiente apartado detallaremos todo lo que debes tener en cuenta para trabajar con funciones. ¡Vamos a ello!



Autoevaluación

En JavaScript no podemos definir nuevas funciones y debemos atenernos a las que ya vienen definidas por defecto en el lenguaje. ¿Verdadero o falso?

☐ Verdadero ☐ Falso

Falso

En JavaScript sí podemos definir nuevas funciones y, a su definición y uso es a lo que nos dedicaremos en este apartado.

2.1.- Trabajando con funciones.

Siempre que sea posible, tienes que diseñar funciones que puedas reutilizar en otras aplicaciones, de esta forma, tus funciones se convertirán en pequeños bloques constructivos que te permitirán ir más rápido en el desarrollo de nuevos programas.

Si conoces otros lenguajes de programación, quizás te suene el término de *método* o *procedimiento*. En JavaScript no vamos a distinguir entre **procedimientos** (que ejecutan acciones), o **funciones** (que ejecutan acciones y devuelven valores). **En JavaScript siempre se llamarán funciones.**



[Everaldo Coelho \(GNU/GPL\)](#)

Una función es capaz de devolver un valor a la instrucción que la invocó, pero esto no es un requisito obligatorio en JavaScript. Cuando una función devuelve un valor, la instrucción que llamó a esa función, la tratará como si fuera una expresión.

Vayamos por pasos y veamos cómo crearlas, utilizarlas, etc.

Sintaxis

Parámetros

Ámbito de las variables

Funciones anidadas

Funciones predefinidas

Sintaxis

Te mostraré algunos ejemplos en un momento, pero antes de nada, vamos a ver la sintaxis formal de una función:

```
function nombreFuncion ( [parametro1]...[parametroN] )
{
    // instrucciones
    [return valor]
}
```

Fíjate que si nuestra función va a **devolver algún valor** emplearemos la palabra reservada `return`, para hacerlo.

Esta es la sintaxis tradicional para definir funciones, pero hoy día es más recomendable el uso del operador flecha (`=>`) para ello. Nosotros utilizaremos esta sintaxis para ir acostumbrándonos a ella ya que en muchos casos nos ahorrará líneas de código y hará nuestro código más legible. La sintaxis para utilizar el operador flecha es la siguiente:

```
const nombreFuncion = ([parametro1]...[parametroN]) => {
    // instrucciones
}
```

```
[return valor]
}
```

Los nombres que puedes asignar a una función, tendrán las mismas restricciones que tienen los elementos HTML y las variables en JavaScript. Deberías asignarle un nombre que realmente la identifique, o que indique qué tipo de acción realiza. Puedes usar palabras compuestas como `chequearCorreo` o `calcularFecha`, y fíjate que las funciones suelen llevar un verbo, puesto que las funciones son elementos que realizan acciones.

Una recomendación que te hacemos, es la de que las funciones sean muy específicas, es decir que no realicen tareas adicionales a las inicialmente propuestas en esa función.

Para realizar una llamada a una función lo podemos hacer con:

```
nombreFuncion( ); // Esta llamada ejecutaría las instrucciones programadas dentro de
```

Otro ejemplo de uso de una función que devuelve un valor y queremos recogerlo en una variable es utilizando una asignación:

```
variable = nombreFuncion( ); // En este caso la función devolvería un valor que se a
```

Las funciones en JavaScript también son objetos, y como tal tienen métodos y propiedades. Un método, aplicable a cualquier función puede ser `toString()`, el cuál nos devolverá el código fuente de esa función.

Parámetros

Cuando se realiza una llamada a una función, muchas veces es necesario pasar parámetros (también conocidos como argumentos). Este mecanismo nos va a permitir enviar datos entre instrucciones.

Para pasar parámetros a una función, tendremos que escribir dichos parámetros entre paréntesis y separados por comas.

A la hora de definir una función que recibe parámetros, lo que haremos es, escribir los nombres de las variables que recibirán esos parámetros entre los paréntesis de la función.

Veamos el siguiente ejemplo:

```
const saludar = (nombre) => {  
    alert(`Hola ${nombre}`);  
}
```

Si llamamos a esa función desde el código:

```
saludar('Bob Esponja'); //Mostraría una alerta con el texto: Hola Bob Esponja.
```

Los parámetros que usamos en la definición de la función *serán variables locales a la función*, y se inicializarán automáticamente en el momento de llamar a la función, con los valores que le pasemos en la llamada. En el siguiente apartado entraremos más en profundidad en lo que son las variables locales y globales.

Otro ejemplo de función que devuelve un valor podría ser el siguiente:

```
const calcularMayor = (num1, num2) => {  
    return (num1 > num2) ? num1 : num2;  
}
```

Esta función la podemos invocar lo mismo que la anterior, simplemente que debemos tener en cuenta que devuelve un valor:

```
console.log(calcularMayor(7, 5));
```

Ámbito de las variables

Ha llegado la hora de distinguir entre las variables que se definen fuera de una función, y las que se definen dentro de las funciones.


Las variables que se definen fuera de las funciones se llaman **variables globales**. Las **variables** que se definen dentro de las funciones se llaman variables locales.

Una **variable global** en JavaScript tiene una connotación un poco diferente, comparando con otros lenguajes de programación. Para un script de JavaScript, el alcance de una variable global, se limita al documento actual que está cargado en la ventana del navegador. Sin embargo cuando inicializas una variable como variable global, quiere decir que todas las instrucciones de tu script (incluidas las instrucciones que están dentro de las funciones), tendrán acceso directo al valor de esa variable. Todas las instrucciones podrán leer y modificar el valor de esa variable global.

En el momento que una página se cierra, todas las variables definidas en esa página se eliminarán de la memoria para siempre. Si necesitas que el valor de una variable persista de una página a otra, tendrás que utilizar técnicas que te permitan almacenar esa variable.

En contraste a las variables globales, una **variable local será definida dentro de una función**. Antes viste que podemos definir variables en los parámetros de una función, pero también podrás definir nuevas variables dentro del código de la función.

El alcance de una variable local está solamente dentro del ámbito de la función. Ninguna otra función o instrucciones fuera de la función podrán acceder al valor de esa variable.

Reutilizar el nombre de una variable global como local es uno de los  bugs más sutiles y por consiguiente más difíciles de encontrar en el código de JavaScript. La variable local en momentos puntuales ocultará el valor de la variable global, sin avisarnos de ello. Como recomendación, no reutilices un nombre de variable global como local en una función, y tampoco declares una variable global dentro de una función, ya que podrás crear fallos que te resultarán difíciles de solucionar.

Como ejemplo de ello puedes ver el siguiente ejemplo:

```
let nombre = 'Pepe';
const prueba = () => {
  let nombre = 'Pedro';
  console.log(nombre);    // Muestra por consola: Pedro
}
console.log(nombre);      // Muestra por consola: Pepe
prueba();
console.log(nombre);      // Muestra por consola: Pepe
```

Funciones anidadas

En JavaScript podemos anidar unas funciones dentro de otras. Es decir podemos programar una función dentro de otra función.

Cuando no tenemos funciones anidadas, cada función que definamos será accesible por todo el código, es decir serán funciones globales. Con las funciones anidadas, podemos encapsular la accesibilidad de una función dentro de otra y hacer que esa función sea privada o local a la función principal. Tampoco te recomiendo el reutilizar nombres de funciones con esta técnica, para evitar problemas o confusiones posteriores.

Una buena opción para aplicar las funciones anidadas, es cuando tenemos una secuencia de instrucciones que necesitan ser llamadas desde múltiples sitios dentro de una función, y esas instrucciones sólo tienen significado dentro del contexto de esa función principal. En otras palabras, en lugar de romper la secuencia de una función muy larga en varias funciones globales, haremos lo mismo pero utilizando funciones locales.

Ejemplo de una función anidada:

```
const calcularHipotenusa = (cateto1, cateto2) => {  
  const calcularCuadrado = (x) => { return x * x; }  
  return Math.sqrt(calcularCuadrado(cateto1) + calcularCuadrado(cateto2));  
}  
console.log(calcularHipotenusa(1, 2)); // Muestra por consola: 2.23606797749979
```

Funciones predefinidas

Te mostraremos aquí una lista de funciones predefinidas, que se pueden utilizar a nivel global en cualquier parte de tu código de JavaScript. Estas funciones no están asociadas a ningún objeto en particular. Típicamente, estas funciones te permiten convertir datos de un tipo a otro tipo.

Funciones globales o predefinidas en JavaScript:

Funciones globales en JavaScript.

Función	Descripción
decodeURI()	Decodifica los caracteres especiales de una <u>URL</u> excepto: , / ? : @ & = + \$ #
decodeURIComponent()	Decodifica todos los caracteres especiales de una URL.
encodeURI()	Codifica los caracteres especiales de una URL excepto: , / ? : @ & = + \$ #
encodeURIComponent()	Codifica todos los caracteres especiales de una URL.
escape()	Codifica caracteres especiales en una cadena, excepto: * @ - _ + . /
eval()	Evalúa una cadena y la ejecuta si contiene código u operaciones.
isFinite()	Determina si un valor es un número finito válido.
isNaN()	Determina cuando un valor no es un número.
Number()	Convierte el valor de un objeto a un número.
parseFloat()	Convierte una cadena a un número real.
parseInt()	Convierte una cadena a un entero.

Función	Descripción
unescape()	Decodifica caracteres especiales en una cadena, excepto: * @ - _ + . /

3.- Objetos.



Caso práctico

Antonio ha hecho una pausa para tomar café, y charla con **Juan** de todo lo que ha aprendido sobre las funciones. Comenta lo interesante que ha sido, pero le surge una duda, ya que cuando estudió los objetos en JavaScript, vio que tenían propiedades y métodos, y los métodos tienen el mismo formato que las funciones que acaba de estudiar. **Juan** le dice que efectivamente eso es así, porque en realidad los métodos son funciones que se asignan a un objeto determinado, y que a parte de los objetos que ya ha estudiado en unidades anteriores, en JavaScript podrá crear sus propios objetos, con los métodos (funciones) y propiedades que él quiera.



Terminan su café y los dos vuelven al despacho para buscar documentación sobre cómo crear nuevos objetos en JavaScript.

En unidades anteriores has visto como toda la información que proviene del navegador o del documento está organizada en un modelo de objetos, con propiedades y métodos. Pues bien, JavaScript también te da la oportunidad de crear tus propios objetos en memoria, objetos con propiedades y métodos que tú puedes definir a tu antojo. Estos objetos no serán elementos de la página de interfaz de usuario, pero sí que serán objetos que podrán contener **datos (propiedades) y funciones (métodos)**, cuyos resultados si que se podrán mostrar en el navegador. El definir tus propios objetos, te permitirá enlazar a cualquier número de propiedades o métodos que tú hayas creado para ese objeto. Es decir, tú controlarás la estructura del objeto, sus datos y su comportamiento.

En el siguiente apartado veremos cómo crear nuestros objetos y cómo utilizarlos. ¡Manos a la obra!

3.1.- Trabajando con objetos.

Un objeto en JavaScript es realmente una colección de **propiedades**. Las propiedades pueden tener forma de datos, tipos, funciones (métodos) o incluso otros objetos. De hecho sería más fácil de entender un objeto como un array de valores, cada uno de los cuales está asociado a una propiedad (un tipo de datos, método u objeto). Un momento: ¿un método puede ser una propiedad de un objeto? Pues en JavaScript parece que sí.



[Everaldo Coelho \(GNU/GPL\)](#)

Una función contenida en un objeto se conoce como un método.

Los métodos no son diferentes de las funciones que has visto anteriormente, excepto que han sido diseñados para ser utilizados en el contexto de un objeto, y por lo tanto, tendrán acceso a las propiedades de ese objeto. Esta conexión entre propiedades y métodos es uno de los ejes centrales de la orientación a objetos.

Pero vamos poco a poco.

Definición

Creación

Propiedades

Constructor

Métodos

Herencia

Definición

Para definir una clase, que será el molde que nos permitirá crear objetos de dicha clase, simplemente debemos utilizar la siguiente sintaxis:

```
class NombreClase {  
    // Cuerpo de la clase  
}
```

Para nombrar una clase comenzamos su identificador con la primera letra en mayúsculas.

Por ejemplo, pensemos que queremos gestionar una serie de contactos, para ello podríamos empezar por definir una clase para un `Contacto` genérico:

```
class Contacto {  
    // Cuerpo de nuestra clase Contacto  
}
```

Creación

Para crear objeto de una clase dada simplemente debemos utilizar el operador `new` seguido del nombre de la clase.

```
let pepe = new Contacto();  
let juan = new Contacto();
```

En breve veremos qué estamos haciendo al realizar esta operación, pero por ahora nos quedamos con que de esta forma podemos crear nuevos objetos de dicha clase.

Propiedades

Pero una clase no tendría sentido sin tener unas propiedades que sean específicas de cada uno de los objetos. A eso se le suele llamar estado de un objeto, ya que las propiedades de un objeto dado serán diferentes a las de otro.

Para declarar propiedades dentro de una clase simplemente las podemos declarar en el cuerpo de la clase, aunque como veremos ahora mismo, lo ideal es hacerlo en el método constructor. Para referirnos a las propiedades debemos anteponer la palabra reservada `this` al nombre de la propiedad como veremos en breve.

```
class Contacto {  
  nombre;  
  correo;  
}
```

Ahora podremos crear objetos de dicha clase e inspeccionar sus propiedades.

```
let juan = new Contacto();  
console.log(juan); // Muestra por consola: Object { nombre: undefined, correo: u
```

Podemos acceder a las propiedades de un objeto utilizando el operador `.`

```
let juan = new Contacto();  
console.log(juan.nombre); // Muestra por consola: undefined
```

```
juan.nombre = 'Juan';  
console.log(juan.nombre); // Muestra por consola: Juan
```

Constructor

Como hemos comentado anteriormente, lo ideal es crear y asignar valores a las propiedades en el método constructor. El método constructor es una función especial que es llamada cuando utilizamos el operador `new` de dicha clase. Dicho método, si no es definido, es definido por defecto, pero lo ideal es que nosotros lo definamos y le demos la funcionalidad adecuada, que suele ser inicializar las propiedades.

```
class Contacto {  
  constructor(nombre, correo) {  
    this.nombre = nombre;  
    this.correo = correo;  
  }  
}
```

En el siguiente ejemplo creamos dos objetos de la clase: en el primero utilizamos el constructor definido por nosotros mismos y en el segundo el constructor definido por defecto.

```
let pepe = new Contacto('Pepe', 'pepe@gmail.com');  
console.log(pepe); // Muestra por consola: Object { nombre: "Pepe", correo: "pepe@gmail.com"}  
let juan = new Contacto();  
console.log(juan); // Muestra por consola: Object { nombre: undefined, correo: undefined}
```

Métodos

Los métodos son funciones definidas en la propia clase que determinan el comportamiento de los objetos creados.

Por ejemplo, podemos crear un método para nuestra clase encargado de devolver un mensaje de saludo que incluya los datos del objeto.

```

class Contacto {

    nombre;
    correo;

    constructor(nombre, correo) {
        this.nombre = nombre;
        this.correo = correo;
    }

    saludar() {
        return `Hola, soy ${this.nombre} y mi correo es ${this.correo}`;
    }
}

let pepe = new Contacto('Pepe', 'pepe@gmail.com');
console.log(pepe.saludar()); // Muestra por consola: Hola soy Pepe y mi correo es p

```

También podemos definir métodos que serán comunes a toda la clase. Son los llamados métodos estáticos y para declararlos debemos anteponer la palabra reservada `static` al nombre del método.

Herencia

En JavaScript también podemos crear jerarquías de clases mediante la herencia. Una clase puede heredar de otra previamente declarado y por lo tanto heredará su estructura y su comportamiento.

Por ejemplo, podríamos tener una clase para almacenar contactos telefónicos en la que queremos almacenar además del nombre y el correo, el número de teléfono. También queremos que el método saludar también devuelva información del teléfono.

```

class ContactoTelefonico extends Contacto {

    telefono;

    constructor(nombre, correo, telefono) {
        super(nombre, correo);
        this.telefono = telefono;
    }

    saludar() {
        return `${super.saludar()} y mi teléfono es ${this.telefono}`;
    }
}

```

```
let maria = new ContactoTelefonico('María', 'maria@gmail.com', '666112233');  
console.log(maria.saludar()); // Muestra por consola: Hola soy María y mi correo es maria@gmail.com
```

