Garceta
grupo editorial

Técnico Superior en **Desarrollo de Aplicaciones Web**

Desarrollo Web en entorno cliente

con JavaScript

Jorge Sánchez Asenjo

Desarrollo Web en entorno cliente con JavaScript

Jorge Sánchez Asenjo

Técnico Superior en **Desarrollo de Aplicaciones Web**

Desarrollo Web en entorno cliente con JavaScript



Garceta
grupo editorial

Desarrollo Web en entorno cliente con JavaScript

Jorge Sánchez Asenjo

ISBN: 978-84-1728-940-9
IBERGARCETA PUBLICACIONES, S.L., Madrid 2019

Edición: 1.ª
Impresión: 1.ª
N.° de páginas: 464
Formato: 20 x 26 cm

Reservados los derechos para todos los países de lengua española. De conformidad con lo dispuesto en el artículo 270 y siguientes del código penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte sin la preceptiva autorización. Ninguna parte de esta publicación, incluido el diseño de la cubierta, puede ser reproducida, almacenada o trasmitida de ninguna forma, ni por ningún medio, sea éste electrónico, químico, mecánico, electro-óptico, grabación, fotocopia o cualquier otro, sin la previa autorización escrita por parte de la editorial.

Diríjase a CEDRO (Centro Español de Derechos Reprográficos), www.cedro.org, si necesita fotocopiaro escanear algún fragmento de esta obra.

COPYRIGHT © 2019 IBERGARCETA PUBLICACIONES, S.L.
info@garceta.es

Desarrollo Web en entorno cliente con JavaScript

© Jorge Sánchez Asenjo

1.ª edición, 1.ª impresión
OI:xx/2019
ISBN: 978-84-1725-940-9

Deposito Legal: M-29757-2019
Imagen de cubierta: © Pedro Benavente - Fotolia

Impresión:
PRINT HOUSE, marca registrada de Copiar, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Nota sobre enlaces a páginas web ajenas: Este libro puede incluir referencias a sitios web gestionados por terceros y ajenos a IBERGARCETA PUBLICACIONES, S.L., que se incluyen sólo con finalidad informativa. IBERGARCETA PUBLICACIONES, S.L., no asume ningún tipo de responsabilidad por los daños y perjuicios derivados del uso de los datos personales que pueda hacer un tercero encargado del mantenimiento de las páginas web ajenas a IBERGARCETA PUBLICACIONES, S.L., y del funcionamiento, accesibilidad y mantenimiento de los sitios web no gestionados por IBERGARCETA PUBLICACIONES, S.L., directamente. Las referencias se proporcionan en el estado en que se encuentran en el momento de publicación sin garantías, expresas o implícitas, sobre la información que se proporcione en ellas.

PRÓLOGO

Este libro está escrito con la vocación de ser utilizado como libro de texto del módulo de **Desarrollo de Aplicaciones Web en Entorno Cliente** del ciclo de Grado Superior de Formación Profesional de **Desarrollo de Aplicaciones Web.**

Por ello se ha puesto especial atención en ayudar a cumplir los resultados de aprendizaje y contenidos expresados en el R.D. 6896/2010 de 20 de mayo en el que se establece el título de **Técnico Superior en Desarrollo de Aplicaciones Web.**

Como ha ocurrido en otros libros que he escrito, mi pretensión ha sido escribir el libro que me hubiera gustado utilizar para impartir esta materia. Pero también, como desarrollador de aplicaciones web he pensado en todos aquellos alumnos y profesionales de áreas afines que buscaran un buen material para ser tenido en cuenta como referencia en su trabajo con JavaScript.

Efectivamente JavaScript es el objeto fundamental de trabajo de este libro, el lenguaje con el que se desarrollan aplicaciones web en el lado del cliente y el lenguaje que ha crecido tanto en estos años que se ha convertido en el lenguaje predominante en la actualidad. Mi esperanza es que (siendo aun más ambicioso) este libro contribuya a que el lector se prende de este lenguaje, como, de hecho, me ha prendado a mí en estos últimos años.

El libro se divide en unidades, las cuales se estructuran de la siguiente manera:

- Se describen los objetivos y resumen de contenidos al iniciar la unidad.
- Se exponen los contenidos teóricos de la misma tratando de fomentar la aplicación inmediata de los mismos, para lo cual se han utilizado ejemplos y actividades que permitan la rápida aplicación de los conceptos.
- Tras los contenidos se plantean prácticas que se explican y resuelven en el propio texto y que se proponen para realizar a la vez que se estudia la parte teórica. Hay, además, otras prácticas propuestas, cuya solución está a disposición de los docentes que deseen utilizar esta obra como libro de texto en sus clases.
- Hay una página final de resumen de la unidad con las ideas más interesantes vistas en la misma.
- Al final de cada unidad se han elaborado preguntas de tipo test para evaluar la comprensión del texto.

La disposición de los capítulos y contenidos se han escrito en un orden en el que los conocimientos que se van adquiriendo ayudan a asimilar los siguientes. Pero hay tres bloques diferenciados.

- **BLOQUE** 1: **Introducción al desarrollo de las aplicaciones web.** Formado por las siguientes unidades:

- **Unidad 1: Introducción al desarrollo web en el lado del cliente.** Que estudia las bases del funcionamiento de las aplicaciones web y el encaje del lenguaje JavaScript en el proceso de desarrollo de las mismas.
- **Unidad 2: Preparación del entorno de trabajo.** En la que se recorren las principales herramientas que necesitamos como desarrolladores de aplicaciones web. Se explica, en las prácticas, cómo instalar las que, desde este libro, se proponen.

- **BLOQUE 2: Fundamentos de JavaScript.** Bloque fundamental que permite recorrer las bases necesarias para comprender y aplicar el lenguaje JavaScript. El dominio de las unidades que componen este bloque es la clave del éxito para todo futuro desarrollador front-end. Las unidades que componen este bloque son:

  - **Unidad 3: Principios de programación en JavaScript.** Incluye los conceptos básicos del lenguaje: variables, operadores, ejecución de código, sentencias condicionales, bucles, etc. Aunque son temas sencillos para quien ya haya escrito código, no viene nada mal un repaso porque siempre se descubren cosas interesantes y porque hay novedades en JavaScript que pueden sorprender a los programadores veteranos.
  - **Unidad 4: Estructuras básicas de datos.** Strings, arrays, conjuntos y mapas forman el contenido de la unidad donde ya podemos almacenar datos de forma más avanzada.
  - **Unidad 5: Funciones.** Para los conocedores de otros lenguajes, quizá la unidad que más les puede sorprender por las características únicas de JavaScript. Desde el punto de vista del autor de este libro, una de las unidades más importantes, sin duda. Al final además se explica el uso de funciones muy potentes que aporta JavaScript para trabajar con arrays y Sets de forma muy avanzada.
  - **Unidad 6: Programación de Objetos en JavaScript.** Esta es la segunda gran unidad en importancia para conocer a fondo el lenguaje. Las diferencia de concepto y manejo de objetos que posee JavaScript sobre otros lenguajes quedan patentes en este tema. Comprender a fondo el modelo de objetos de JavaScript es, simplemente, imprescindible para profundizar en el lenguaje.

- **BLOQUE 3: Manipulación de los elementos HTML de la aplicación web.** Es el bloque nuclear para alcanzar los resultados de aprendizaje de esta materia. Se hace muy sencillo si el bloque anterior se ha superado correctamente. Dos unidades conforman el bloque:

  - **Unidad 7: Manipulación del DOM.** Unidad en la que aprenderemos a modificar todos los elementos HTML, tanto sus propiedades, como a aplicar métodos para transformarlos, reubicarlos, eliminarlos, crearlos, etc.
  - **Unidad 8: Gestión de eventos.** Es la segunda parte del DOM y la base de la programación orientada a eventos y asincrona, tan importante en JavaScript. Al finalizar el bloque se debería ser capaz de realizar casi cualquier aplicación web

- **BLOQUE 4: Programación avanzada en JavaScript.** La idea de este bloque es llegar a la creación de aplicaciones que sean capaces de comunicarse con servidores, que manejen APIs avanzadas y que usen las mejoras que se han producido en el lenguaje JavAScript.

- **Unidad 9: Conceptos avanzados.** Control de errores y excepciones, romesas, instrucciones **async/await,** carga dinámica de módulos y otros elementos de JavaScript de alto nivel para crear aplicaciones más mantenibles y poder usar las últimas mejoras del lenguaje.
- **Unidad 10: AJAX.** Esta unidad completa el currículo de esta materia gracias a poder realizar peticiones http y recoger respuestas de forma dinámica mediante la técnica conocida como AJAX. Se explica en detalle la mejor manera de realizar esta labor con ayuda de la API Fetch.
- **Unidad 11: Uso de APls de JavaScript.** Las librerías de JavaScript, APls, permiten hacer todo tipo de tareas. En esta unidad se explican algunas de las más importantes: dibujo en **canvas,** geolocalización, notificaciones, **WebStorage** o la APi de manejo de medios audiovisuales.

El problema con los contenidos, especialmente para los docentes de la materia, es si hay tiempo disponible para impartir todos los contenidos. No es la pretensión del libro en que se siga al pie de la letra. Es decir, es imprescindible en esta materia que el manejo del DOM, Eventos y AJAX se puedan trabajar en el aula, pero puede ahogar, dependiendo del grupo de alumnos, el intentar ver las primeras unidades completas dedicadas a las bases del lenguaje antes de llegar a esas unidades cuyos contenidos son más motivantes.

Yo mismo como docente, utilizo conceptos y prácticas de las últimas unidades para explicar conceptos de las unidades más teóricas. Es decir, el libro no pretende ser rígido, pretende ser una herramienta de trabajo.

Durante el proceso de escritura de este libro tuve siempre la tentación de si utilizar JavaScript puro (lo que se conoce como **Vanilla JS)** o **jQuery,** ya que conozco a muchos docentes que usan siempre jQuery en todo el jQuery que escriben. Yo mismo he sido de esos docentes, pero lo cierto es que las razones para utilizar jQuery se diluyen con las mejoras que ha tenido el lenguaje JavaScript (especialmente en el manejo del DOM) en estos años y he creído conveniente ceñirme al estándar ya que está totalmente implantado tanto en los navegadores **Chromium** como **Google Chrome, Opera** y **Microsoft Edge,** como en el navegador **Mozilla Firefox.**

Pasar de JavaScript puro a jQuery (u otras librerías y frameworks) es sencillo, pero trabajar bien en JavaScript si siempre se ha usado jQuery, es más complejo y acaba suponiendo una tara en el aprendizaje de esta materia. Ese es mi punto de vista.

También he pretendido que los contenidos incluyeran los últimos de los estándares JavaScript para motivar a la renovación, especialmente de los docentes, que tanto sufrimos el desfase de nuestras materias.

Finalmente decir que, desde la URL http://www.jorgesanchez.net/librodwec, se puede encontrar material extra relacionado con el libro, así como la posibilidad de descargar el código de las prácticas. En esa misma dirección se pueden encontrar formas de contacto conmigo, así como otros textos, ejercicios y presentaciones de otras materias y referencias a otros libros escritos.

Jorge Sánchez Asenjo
@jorgesancheznet

ÍNDICE GENERAL

UNIDAD 1. INTRODUCCIÓN AL DESARROLLO WEB EN EL LADO DEL CLIENTE....1

1.1 CREACIÓN DE APLICACIONES WEB....2

1.1.1 ¿QUÉ ES UNA APLICACIÓN WEB?....2

1.1.2 PROCESO DE DESARROLLO DE APLICACIONES WEB....2

1.1.3 FRONT-END Y BACK-END....4

1.2 TECNOLOGÍAS DEL LADO DEL CLIENTE....6

1.2.1 BREVE RECORRIDO POR LAS TECNOLOGÍAS FRONT-END....6

1.2.2 ELTRIUNFO DEJAVASCRIPT....7

1.3 VERSIONES DE JAVASCRIPT....10

1.3.1 EL PROBLEMA DE LA COMPATIBILIDAD....10

1.3.2 LENGUAJES PREPROCESADOS....10

1.3.3 ESTÁNDARES A PARTIR DELAÑO 2015....11

1.3.4 SITUACIÓN ACTUAL DE JAVASCRIPT....12

1.4 RESUMEN DE LA UNIDAD....13

1.5 TEST DE REPASO....14

UNIDAD 2. PREPARACIÓN DEL ENTORNO DE TRABAJO....15

2.1 LA CAJA DE HERRAMIENTAS DEL DESARROLLADOR FRONT-END....16

2.2 NAVEGADORES....17

2.3 EDITORES DE CÓDIGO....19

2.3.1 AYUDAS DE LOS EDITORES DE CÓDIGO....19

2.3.2 TIPOS DE EDITORES DE CÓDIGO....20

2.3.3 INSTALACIÓN DE VISUAL STUDIO CODE....22

2.3.4 USO DE VISUAL STUDIO CODE....22

2.3.5 CONFIGURACIÓN DE VISUAL STUDIO CODE....25

2.4 NODE.JS YNPM....26

2.4.1 ¿QUÉ ES NODE.JS?....26

2.4.2 INSTALAR NODE.JS....27

2.4.3 NVM....28

2.4.4 NPM....29

2.5 GIT....30

2.5.1 SISTEMAS DE CONTROL DE VERSIONES Y GIT....30

2.5.2 INSTALACIÓN DE GIT....31

2.5.3 FUNCIONAMIENTO BÁSICO DE GIT....31

2.5.4 GITHUB. GIT EN REMOTO....32

2.6 PRÁCTICAS RESUELTAS....34

2.7 RESUMEN DE LA UNIDAD....61

2.8 TEST DE REPASO....62

UNIDAD 3. PRINCIPIOS DE PROGRAMACIÓN EN JAVASCRIPT63

3.1 EJECUTAR CÓDIGO JAVASCRIPT64
3.1.1 EJECUCIÓN DEJAVASCRIPT EN EL NAVEGADOR64
3.1.2 EJECUCIÓN DE JAVASCRIPT FUERA DEL NAVEGADOR67

3.2 FUNDAMENTOS BÁSICOS68
3.2.1 CONCEPTOS FUNDAMENTALES68

3.3 VARIABLES Y TIPOS DE DATOS71
3.3.1 VALORES71
3.3.2 DECLARACIÓN DE VARIABLES72
3.3.3 TIPOS DE DATOS PRIMITIVOS74

3.4 OPERADORES79
3.4.1 ARITMÉTICOS79
3.4.2 RELACIONALES79
3.4.3 OPERADOR DE ENCADENAMIENTO81
3.4.4 OPERADORES LÓGICOS82
3.4.5 OPERADORES DE BIT83
3.4.6 OPERADORES DE ASIGNACIÓN85

3.5 CONVERSIÓN DE TIPOS87
3.5.1 CONVERSIÓN AUTOMÁTICA87
3.5.2 FORZAR CONVERSIONES88
3.5.3 FUNCIÓN ISNAN89

3.6 CUADROS DE DIÁLOGO DEL NAVEGADOR.....89
3.6.1 MENSAJES DE ALERTA 89
3.6.2 CUADROS DE CONFIRMACIÓN 90
3.6.3 CUADROS DE ENTRADA DE TEXTO 91

3.7 CONTROL DEL FLUJO DEL PROGRAMA92
3.7.1 INTRODUCCIÓN92
3.7.2 NÚMEROS ALEATORIOS92
3.7.3 INSTRUCCIÓN **IF93**
3.7.4 INSTRUCCIÓN **WHILE96**
3.7.5 BUCLE **DO...WHILE100**
3.7.6 BUCLE **FOR101**
3.7.7 ABANDONAR UN BUCLE102
3.7.8 BUCLES ANIDADOS103

3.8 PRÁCTICAS RESUELTAS 105
3.9 PRÁCTICAS PROPUESTAS113
3.10 RESUMEN DE LA UNIDAD115
3.11 TEST DE REPASO117

UNIDAD 4. ESTRUCTURAS BÁSICAS DE DATOS 119

4.1 STRINGS120
4.1.1 USO DE STRINGS120
4.1.2 COMPARACIÓN DE STRINGS120
4.1.3 MÉTODOS DE LOS STRINGS121

4.2 ARRAYS126
4.2.1 ¿QUÉ ES UN ARRAY?126
4.2.2 CREACIÓN DE ARRAYS127

4.3 RECORRER ARRAYS..... 131
4.3.1 USO DEL BUCLE **FOR** PARA RECORRER ARRAYS..... 131
4.3.2 BUCLE **FOR...IN**..... **132**
4.3.3 BUCLE **FOR..OF**..... **133**
4.4 MÉTODOS DE LOS ARRAYS..... 134
4.4.1 TAMAÑO DEL ARRAY..... 134
4.4.2 SABER SI UN ELEMENTO ES UN ARRAY..... 134
4.4.3 MÉTODOS PARA AÑADIR ELEMENTOS..... 134
4.4.4 MEZCLAR ARRAYS..... 135
4.4.5 OBTENER Y AÑADIR SUBARRAYS..... 136
4.4.6 CONVERTIR ARRAY EN STRING..... 137
4.4.7 BÚSQUEDA DE ELEMENTOS EN UN ARRAY..... 138
4.4.8 MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY..... 139
4.4.9 DESESTRUCTURACIÓN DE ARRAYS..... 140
4.5 ESTRUCTURAS DE TIPO SET..... 141
4.5.1 INTRODUCCIÓN..... 141
4.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS..... 141
4.5.3 MÉTODOS DE LOS CONJUNTOS..... 142
4.5.4 CONVERTIR CONJUNTOS EN ARRAYS..... 143
4.5.5 RECORRER CONJUNTOS..... 143
4.6 MAPAS..... 144
4.6.1 ¿QUÉ SON LOS MAPAS?..... 144
4.6.2 DECLARAR MAPAS..... 144
4.6.3 ASIGNAR VALORES A MAPAS..... 144
4.6.4 OPERACIONES SOBRE MAPAS..... 145
4.6.5 CONVERTIR MAPAS EN ARRAYS..... 147
4.6.6 RECORRER MAPAS..... 147
4.7 PRÁCTICAS RESUELTAS..... 149
4.8 PRÁCTICAS RECOMENDADAS..... 157
4.9 RESUMEN DE LA UNIDAD..... 160
4.10 TEST DE REPASO..... 161
UNIDAD 5. FUNCIONES..... 163
5.1 INTRODUCCIÓN A LAS FUNCIONES..... 164
5.2 CREACIÓN DE FUNCIONES..... 164
5.2.1 ELEMENTOS DE UNA FUNCIÓN..... 164
5.2.2 DECLARAR E INVOCAR FUNCIONES..... 165
5.2.3 ASIGNAR FUNCIONES A VARIABLES..... 167
5.2.4 FUNCIONES FLECHA..... 168
5.3 DETALLES SOBRE VARIABLES Y PARÁMETROS..... 170
5.3.1 ÁMBITO DE LAS VARIABLES..... 170
5.3.2 PASO POR VALOR Y PASO POR REFERENCIA..... 171
5.3.3 ARGUMENTOS CON VALORES POR DEFECTO..... 173
5.3.4 NÚMERO VARIABLE DE PARÁMETROS..... 174
5.4 USO AVANZADO DE FUNCIONES..... 176
5.4.1 LA PILA DE FUNCIONES..... 176
5.4.2 RECURSIVIDAD..... 180
5.4.3 FUNCIONES CALLBACK..... 183

5.4.4 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS185
5.5 PRACTICAS RESUELTAS.....190
5.6 PRÁCTICAS RECOMENDADAS.....197
5.7 RESUMEN DE LA UNIDAD.....199
5.8 TEST DE REPASO..... 200
UNIDAD 6. PROGRAMACIÓN DE OBJETOS EN JAVASCRIPT.....203
6.1 JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS..... 204
6.1.1 ¿QUÉ ES LA POO?..... 204
6.1.2 CARACTERÍSTICAS DE LA POO..... 204
6.1.3 JAVASCRIPT COMO LENGUAJE ORIENTADO A OBJETOS.....206
6.2 USO DE OBJETOS..... 206
6.2.1 ACCESO A PROPIEDADES Y MÉTODOS..... 206
6.2.2 OBJETOS LITERALES..... 207
6.2.3 OBJETO **THIS**.....**208**
6.2.4 RECORRER LAS PROPIEDADES DE UN OBJETO..... 210
6.2.5 BORRAR PROPIEDADES DE OBJETOS.....210
6.3 USO AVANZADO DE OBJETOS.....211
6.3.1 CREAR OBJETOS A TRAVÉS DE CONSTRUCTORES.....211
6.3.2 OPERADOR INSTANCEOF.....212
6.3.3 PROTOTIPOS.....213
6.3.4 NOTACIÓN JSON.....215
6.4 OBJETOS PREDEFINIDOS..... 217
6.4.1 OBJETO MATH..... 217
6.4.2 OBJETO DATE..... 218
6.4.3 EXPRESIONES REGULARES..... 222
6.5 PRÁCTICAS RESUELTAS.....231
6.6 PRÁCTICAS RECOMENDADAS.....242
6.7 RESUMEN DE LA UNIDAD.....245
6.8 TEST DE REPASO.....247
UNIDAD 7. MANIPULACIÓN DEL MODELO DE OBJETOS DEL DOCUMENTO249
7.1 EL OBJETO WINDOW.....250
7.1.1 BOM, MODELO DE OBJETOS DEL NAVEGADOR.....250
7.1.2 OBJETO NAVIGATOR..... 251
7.1.3 OBJETO SCREEN..... 252
7.1.4 OBJETO LOCATION.....253
7.1.5 OBJETO HISTORY..... 254
7.1.6 OTRAS PROPIEDADES Y MÉTODOS DE WINDOW.....255
7.2 ¿QUÉ ES EL DOM?.....257
7.2.1 EL MODELO DE OBJETOS DEL DOCUMENTO.....257
7.2.2 EL OBJETO DOCUMENT..... 259
7.2.3 TIPOS DE NODOS.....259
7.3 SELECCIÓN DE ELEMENTOS DEL DOM..... 260
7.3.1 SELECCIÓN POR EL IDENTIFICADOR.....261
7.3.2 POR ETIQUETA.....261
7.3.3 OBJETOS NODELIST.....262

7.3.4 POR CLASE....262
7.3.5 POR SELECTOR CSS....263
7.4 OBTENER Y MODIFICAR DOM....263
7.4.1 MANIPULACIÓN DE ATRIBUTOS....263
7.4.2 MANIPULACIÓN DEL CONTENIDO DE LOS ELEMENTOS....265
7.4.3 MODIFICAR CSS....266
7.4.4 OBTENER ATRIBUTOS DATA....270
7.4.5 NAVEGAR POR EL DOM....270
7.4.6 AÑADIR ELEMENTOS....273
7.4.7 REEMPLAZAR ELEMENTOS....275
7.4.8 ELIMINAR ELEMENTOS....276
7.4.9 COLECCIONES VIVAS....276
7.4.10 OTRAS PROPIEDADES Y MÉTODOS DE LOS ELEMENTOS....278
7.5 TEMPORIZADORES....279
7.5.1 ¿QUÉ SON LOS TEMPORIZADORES?....279
7.5.2 SETTIMEOUT....280
7.5.3 SETINTERVAL....280
7.6 COOKIES....281
7.6.1 ¿QUÉ SON LAS COOKIES?....281
7.6.2 LECTURA Y GRABACIÓN DE COOKIES....281
7.6.3 FECHAS DE EXPIRACIÓN....282
7.6.4 RUTA Y DOMINIO DE LAS COOKIES....283
7.6.5 BORRAR COOKIES....283
7.7 PRÁCTICAS RESUELTAS....284
7.8 PRÁCTICAS RECOMENDADAS....292
7.9 RESUMEN DE LA UNIDAD....294
7.10 TEST DE REPASO....296
UNIDAD 8. GESTIÓN DE EVENTOS....299
8.1 INTRODUCCIÓN A LOS EVENTOS....300
8.2 CAPTURA DE EVENTOS....300
8.2.1 MÉTODOS CLÁSICOS Y OBSOLETOS....300
8.2.2 MÉTODO ACONSEJABLE....301
8.2.3 PROPAGACIÓN DE EVENTOS....302
8.2.4 ANULAR EVENTOS....304
8.2.5 CAPTURA DE EVENTOS EN ELEMENTOS DINÁMICOS....304
8.3 OBJETO DE EVENTO....305
8.3.1 UTILIDAD Y USO DEL OBJETO DE EVENTO....305
8.3.2 OBTENER COORDENADAS DEL EVENTO....306
8.3.3 OBTENER LA TECLA PULSADA....307
8.3.4 ANULAR COMPORTAMIENTOS PREDETERMINADOS....309
8.3.5 CANCELAR PROPAGACIÓN....309
8.3.6 LANZAR EVENTOS....310
8.4 LISTA DE TIPOS DE EVENTOS....311
8.4.1 EVENTOS DE RATÓN....311
8.4.2 EVENTOS DE TECLADO....312
8.4.3 EVENTOS DE MOVIMIENTO EN LA VENTANA....313
8.4.4 EVENTOS SOBRE CARGA Y DESCARGA DE ELEMENTOS....313

8.4.5 OTROS EVENTOS.... 314
8.5 FORMULARIOS.... 316
8.5.1 EL FORMULARIO COMO OBJETO DEL DOM.... 316
8.5.2 EVENTOS DE FORMULARIO.... 318
8.5.3 PROPIEDADES DE LOS CONTROLES.... 319
8.5.4 MÉTODOS DE LOS CONTROLES.... 321
8.6 PRÁCTICAS SOLUCIONADAS.... 322
8.7 PRÁCTICAS RECOMENDADAS.... 339
8.8 RESUMEN DE LA UNIDAD.... 343
8.9 TEST DE REPASO.... 344
UNIDAD 9. CONCEPTOS AVANZADOS.... 345
9.1 JAVASCRIPT AVANZADO.... 346
9.2 CONTROL DE ERRORES.... 346
9.2.1 INTRODUCCIÓN AL CONTROL DE ERRORES.... 346
9.2.2 ERRORES, EXCEPCIONES Y AVISOS.... 347
9.2.3 JAVASCRIPT MÁS ESTRICTO.... 347
9.2.4 CREAR Y LANZAR ERRORES PROPIOS.... 348
9.2.5 GESTIONAR LAS EXCEPCIONES. BLOQUE TRY...CATCH.... 349
9.3 MÓDULOS.... 351
9.3.1 USO DE MÓDULOS Y PAQUETES.... 351
9.3.2 CARGA Y CREACIÓN DE MÓDULOS.... 353
9.4 PROGRAMACIÓN ASÍNCRONA.... 355
9.4.1 PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA.... 355
9.4.2 CALLBACK HELL.... 356
9.4.3 PROMESAS.... 357
9.4.4 FUNCIONES **ASYNC**.... 364
9.5 PRÁCTICAS RESUELTAS.... 367
9.6 RESUMEN DE LA UNIDAD.... 371
9.7 TEST DE REPASO.... 373
UNIDAD 10. AJAX.... 375
10.1 COMUNICACIÓN CLIENTE/SERVIDOR.... 376
10.2 INTRODUCCIÓN A AJAX.... 377
10.2.1 ¿QUÉ ES AJAX?.... 377
10.2.2 VENTAJAS DE AJAX.... 378
10.2.3 DESVENTAJAS DE AJAX.... 379
10.3 PETICIONES AJAX.... 379
10.3.1 USO DE UNA API PARA ACCEDER A UN SERVICIO WEB.... 379
10.3.2 CORS.... 380
10.3.3 APIS EN JAVASCRIPT PARA EL USO DE AJAX.... 380
10.4 REALIZAR PETICIONES MEDIANTE FETCH.... 381
10.5 MANIPULAR LA RESPUESTA. OBJETO RESPONSE.... 382
10.5.1 PROPIEDADES Y MÉTODOS DEL OBJETO DE RESPUESTA.... 382
10.5.2 DATOS DE LA RESPUESTA.... 383
10.5.3 PROCESAMIENTO DE LAS RESPUESTAS.... 384

10.6 PERSONALIZAR LA PETICIÓN. OBJETO REQUEST **389**
10.6.2 ESTABLECER LA CABECERA. OBJETO HEADERS 389
10.6.1 PROPIEDADES Y MÉTODOS DE REQUEST 391
10.7 ENVIAR DATOS CON LA PETICIÓN **392**
10.7.1 ENVÍO DE DATOS USANDO GET392
10.7.2 ENVÍO DE DATOS DE FORMULARIO394
10.7.3 ENVÍO DE PARÁMETROS EN FORMATO JSON397
10.8 USO DE AWAIT/ASYNC CON FETCH 398
10.9 PRÁCTICAS SOLUCIONADAS 400
10.10 PRÁCTICAS RECOMENDADAS **417**
10.11 RESUMEN DE LA UNIDAD **422**
10.12 TEST DE REPASO **423**

UNIDAD 11. USO DE APIS DE JAVASCRIPT425
11.1 ¿QUÉ SON LAS APIS?426
11.2 APIS DE HTML5426
11.3 API PARA CANVAS **428**
11.3.1 ¿QUÉ ES CANVAS?428
11.3.2 EMPEZAR A TRABAJAR CON CANVAS428
11.3.3 DIBUJO DE FORMAS 429
11.3.4 DIBUJO DE TEXTO 432
11.3.5 DIBUJAR IMÁGENES433
11.3.6 BORRAR433
11.3.7 CONCLUSIONES SOBRE CANVAS 434
11.4 WEBSTORAGE434
11.4.1 INTRODUCCIÓN A WEB STORAGE434
11.4.2 GRABAR Y LEER DATOS435
11.4.3 BORRAR ITEMS436
11.4.4 CONCLUSIÓN SOBRE WEB STORAGE436
11.5 GEOLOCATION436
11.5.1 OBTENER LA POSICIÓN437
11.5.2 OBTENER POSICIÓN CONTINUAMENTE438
11.6 API PARA MULTIMEDIA **439**
11.7 NOTIFICACIONES441
11.7.1 FUNCIONAMIENTO DE LA API DE NOTIFICACIONES441
11.7.2 CONCLUSIONES SOBRE LAS NOTIFICACIONES444
11.8 PRÁCTICAS SOLUCIONADAS 445
11.9 RESUMEN DE LA UNIDAD455
11.10 TEST DE REPASO456

UNIDAD 1

INTRODUCCIÓN AL DESARROLLO WEB EN EL LADO DEL CLIENTE

OBJETIVOS

- Determinar los modelos de ejecución de código en las aplicaciones web
- Distinguir las capas presentes en la ejecución de código en el lado del cliente y en el del servidor
- Reconocer las tecnologías presentes en el lado del cliente
- Identificar las funcionalidades que el lenguaje JavaScript aporta a las aplicaciones web
- Reconocer las razones del éxito del lenguaje JavaScript para programar en el lado del cliente
- Reconocer la forma de normalizar JavaScript y su situación actual

CONTENIDOS

1.1 CREACIÓN DE APLICACIONES WE B
- 1.1.1 ¿QUÉ ES UNA APLICACIÓN WEB?
- 1.1.2 PROCESO DE DESARROLLO DE APLICACIONES WEB
- 1.1.3 FRONT-END Y BACK-END

1.2 TECNOLOGÍAS DEL LADO DEL CLIENTE
- 1.2.1 BREVE RECORRIDO POR LAS TECNOLOGÍAS FRONT-END
- 1.2.2 EL TRIUNFO DE JAVASCRIPT

1.3 VERSIONES DE JAVASCRIPT
- 1.3.1 EL PROBLEMA DE LA COMPATIBILIDAD
- 1.3.2 LENGUAJES PREPROCESADOS
- 1.3.3 ESTÁNDARES A PARTIR DEL AÑO 2015
- 1.3.4 SITUACIÓN ACTUAL DE JAVASCRIPT

1.4 RESUMEN DE LA UNIDAD

1.5 TEST DE REPASO

1.1 CREACIÓN DE APLICACIONES WEB

1.1.1 ¿QUÉ ES UNA APLICACIÓN WEB?

En informática, una aplicación es un software que permite al usuario realizar una determinada tarea o servicio. Las aplicaciones clásicas se crean en lenguajes como C o C++ que ponen a disposición del programador todas las capacidades de la computadora. Actualmente a este tipo de aplicaciones se las conoce como **aplicaciones de escritorio.**

Una aplicación web, simplemente, es una aplicación creada para ser ejecutada por un software especial conocido como **navegador.** Con lo cual, los lenguajes en los que se pueden programar estas aplicaciones son aquellos que un navegador es capaz de traducir.

Hoy en día las aplicaciones web son el tipo de aplicaciones más populares ya que, independientemente del sistema, todos los usuarios y usuarias tienen instalado en sus equipos algún navegador ***(Google Chrome, Internet Explorer, Mozilla Firefox...)*** Los navegadores están presentes incluso en dispositivos más pequeños como tabletas o smartphones.

1.1.2 PROCESO DE DESARROLLO DE APLICACIONES WEB

A la hora de crear aplicaciones web se pueden utilizar diversos lenguajes y tecnologías. Pero hay dos tipos de tecnologías principales a tener en cuenta.

- **Lenguajes y tecnologías que se ejecutan en el lado del cliente.** Se trata de los elementos que se incorporan junto al código HTML de una aplicación web y que necesitan ser interpretados por el navegador del usuario. Este código puede ser, incluso, examinado por el propio usuario.

- **Lenguajes y tecnologías en el lado del servidor.** Es la programación de la aplicación escrita para ser interpretada por el servidor web. El código escrito, en este caso, queda oculto al usuario ya que se ejecuta de forma externa.

Una aplicación web típica utiliza ambas tecnologías.
Las aplicaciones web actuales utilizan lo que se conoce como arquitectura de tres niveles (en inglés ***three-tier architecture),*** a veces incluso se habla de más capas. Estas capas son:

- **Capa de presentación.** También se la conoce como **capa del cliente** o **capa del navegador.** Esta capa controla la parte de la aplicación web que llega al navegador del usuario. Es decir, se encarga de la forma de presentar la información al usuario. Esta capa es la relacionada con la programación del lado del cliente o ***front-end*** y, actualmente, se programa en HTML, CSS y JavaScript. El código que procede de esta capa lo envía el servidor web y lo entrega al navegador, esa comunicación se realiza por medio del protocolo http.

- **Capa lógica,** llamada también **capa del servidor** y **capa de aplicación.** Es la encargada de gestionar el funcionamiento interno de la aplicación. La capa se programa en lenguajes

del lado del servidor como: **PHP, ASP, Python, Java, JavaScript** (hay JavaScript del lado del servidor) y otros. El código de esta capa se traduce por un **servidor de aplicaciones** que, normalmente, es un servidor web con módulos de traducción de esos lenguajes.

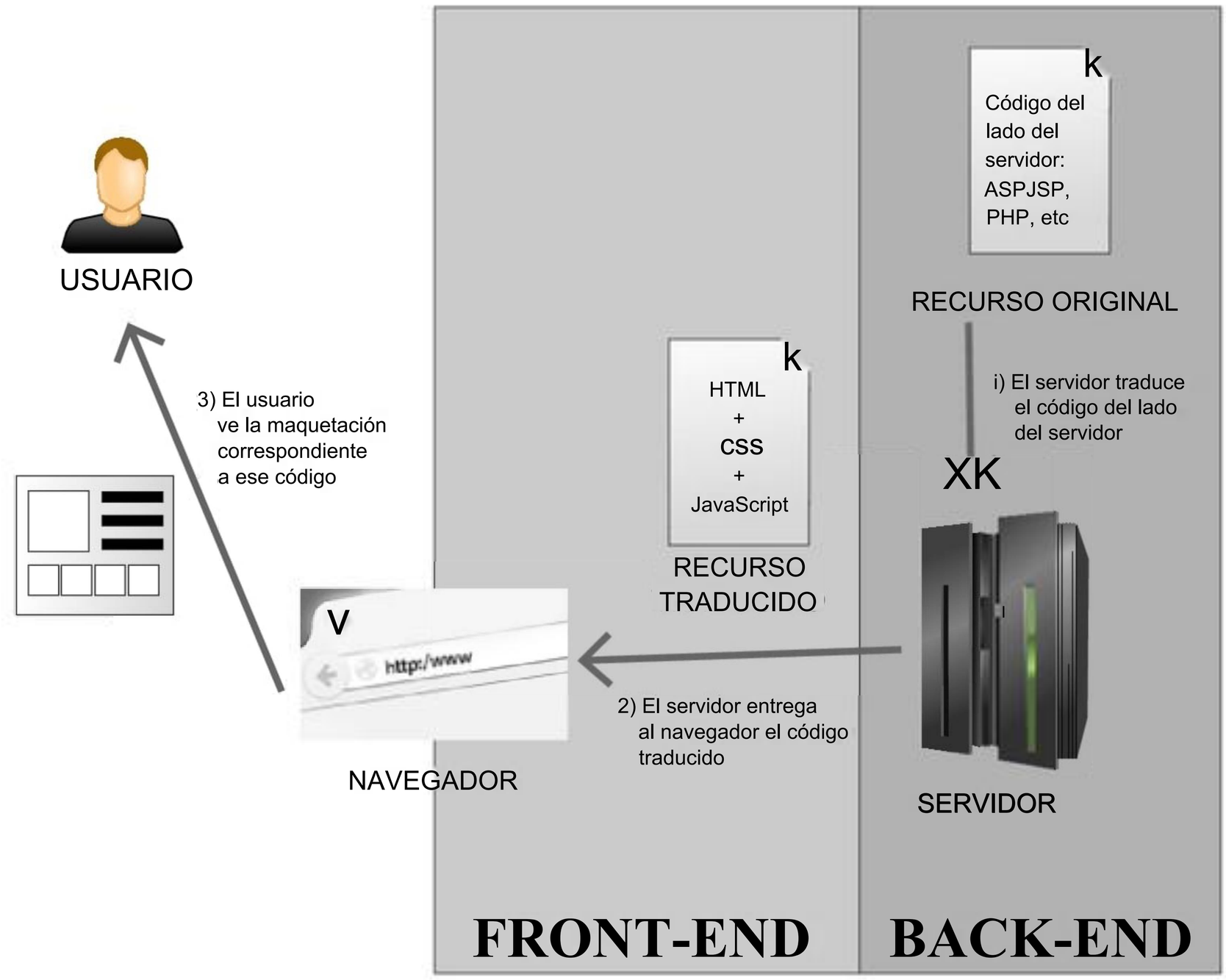


Figura 1.1: El ***front*** y el ***back*** en el desarrollo de aplicaciones web

El servidor de aplicaciones traduce ese código cuyo resultado será código ya compatible con la capa de presentación. Por ejemplo, si hemos escrito esta capa usando el lenguaje PHP, el servidor de aplicaciones traducirá el código PHP produciendo código HTML, CSS ***y*** JavaScript que es el que se entregará al navegador. Al navegador no le llega el código PHP original.

- Capa de negocio. Llamada también **capa de datos.** En esta capa se almacena la información empresarial. Esta información siempre tiene un requerimiento básico: que su información quede oculta a cualquier persona sin autorización. En esta capa, habitualmente, se encuentra un **sistema gestor de bases de datos (SGBD),** que es un software que sirve para administrar la información de la empresa. En esta capa también puede haber otro tipo de servidores que sirvan para proporcionar recursos empresariales tales como vídeo, audio, certificados, correo, etc.

  La capa lógica se comunica con esta capa para obtener datos que luego procesa y entrega, ya convertido en un formato interpretable, al navegador. De modo que el proceso de acceso a estos recursos queda oculto totalmente al navegador, lo que añade una mayor seguridad al proceso.

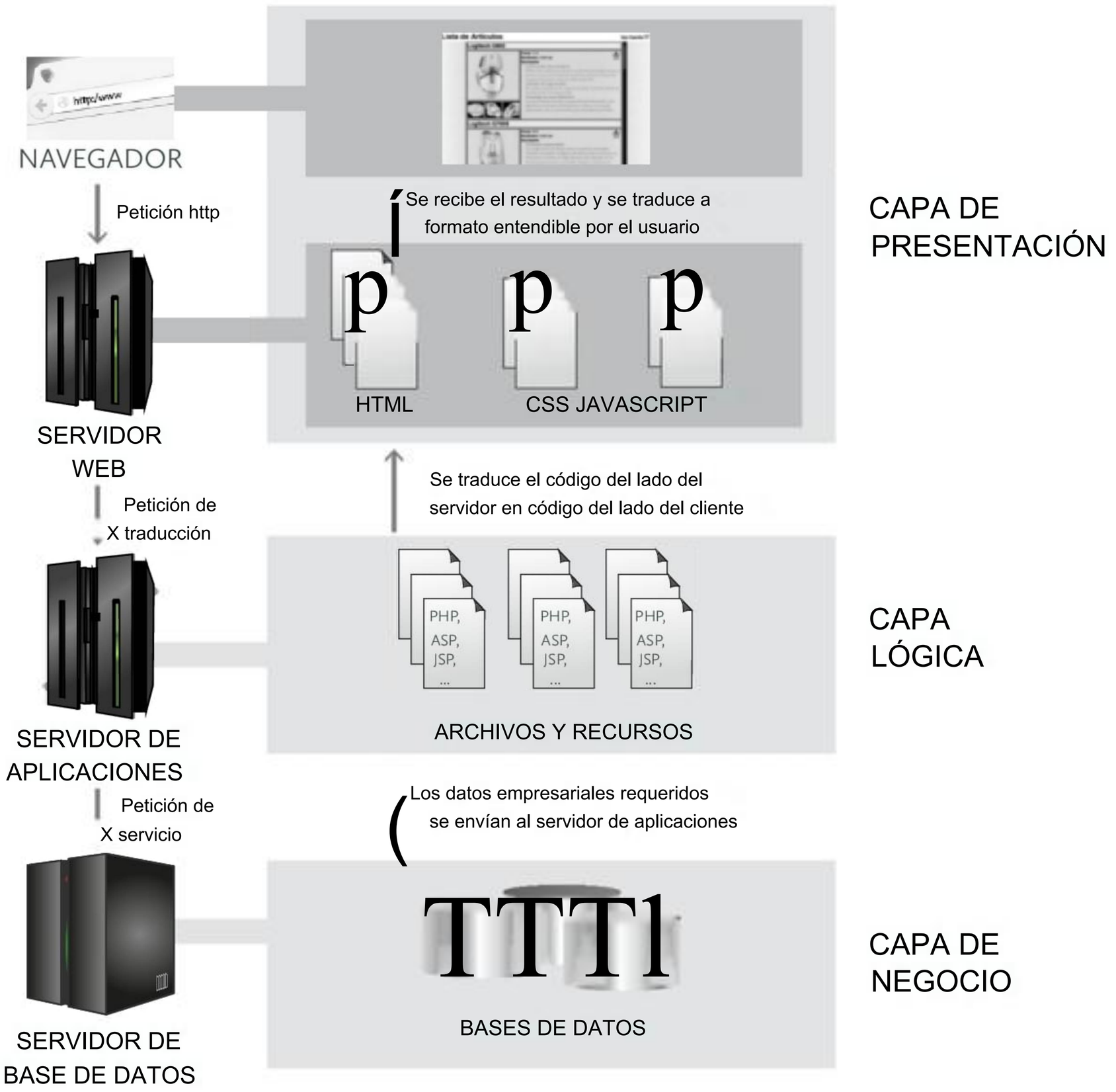


Figura 1.2: Funcionamiento de las 3 capas

1.1.3 FRONT-END Y BACK-END

La división en capas del funcionamiento de las aplicaciones web ha propiciado dos tipos fundamentales de desarrolladores de aplicaciones: los programadores **front-end** y los programadores **back-end:**

- **Front-end.** Se refiere a la parte del desarrollo encargado de producir la apariencia final de la aplicación que verá el usuario. Está relacionada con la capa de presentación del modelo de tres capas. En el front-end trabajan los diseñadores de la aplicación, encargados de determinar lo que se conoce como experiencia de usuario (conocida por las siglas UX) y el diseño visual de la aplicación y los programadores de los lenguajes HTML, CSS y javaScript que realmente construyen el funcionamiento final de la aplicación. El objetivo es optimizar la presentación y conseguir una interfaz final lo más agradable y funcional posible para los usuarios.

- **Back-end.** Un desarrollador back-end se encarga de realizar la parte de la aplicación que queda oculta al usuario. El funcionamiento de la capa lógica y de negocio sería el back-end, por lo que en esta zona se usan los lenguajes del lado del servidor y los de base de datos. Esos lenguajes son muy variados y mucho más especializados.

  Los objetivos del back-end son: un acceso rápido a los datos, una comunicación eficiente con la base de datos y el navegador, control de la seguridad, etc.

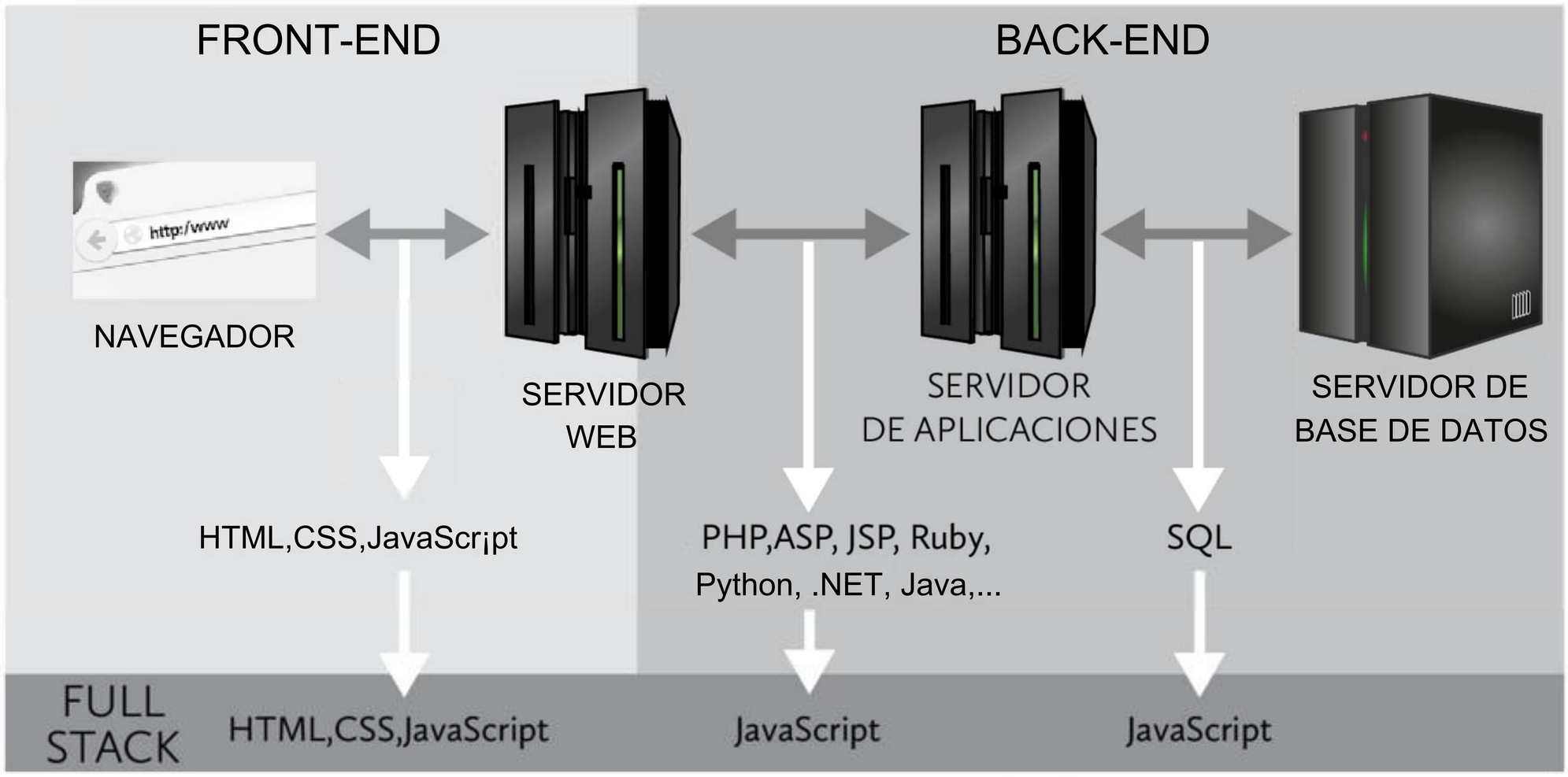


Figura 1.3: Funcionamiento front-end y back-end

No hay una zona mejor que otra, ambas necesitan del concurso de la otra. Al final una aplicación típica puede utilizar numerosos lenguajes y tecnologías y eso impide que un programador front-end tenga difícil pasar al back-end y viceversa. Para mitigar este problema, hay desarrollos que utilizan el mismo lenguaje en las tres cañas (que siempre será JavaScript) y de ese modo el cambio es más intuitivo. Se conoce a esta estrategia como ***Full Stack Development.***

Parecía que esta estrategia iba a tener un gran resultado, y en parte lo ha tenido, pero lo cierto es que la especialización de profesionales en la creación de aplicaciones web sigue siendo muy importante e incluso cada vez se especializa más. Así, sigue siendo normal, a la hora de optar a un puesto profesional de esta rama, que la oferta se refiera a un programador front-end o a un programador back-end. Es, todavía .extraño pedir un programador ***full-stack.***

Incluso, muchas veces, se requieren profesionales más específicos como diseñadores, administradores de bases de datos, especialistas en **DevOps**[1]**,** desarrolladores de juegos, científicos de datos o especialistas en inteligencia artificial.

---

i Práctica de los ingenieros informáticos que aglutina conocimientos tanto de administración de sistemas como de desarrollo web. Permite integrar todos los procesos, e implicáis crear, probar e implementar el software en un entorno de producción.

1.2 TECNOLOGÍAS DEL LADO DEL CLIENTE

1.2.1 BREVE RECORRIDO POR LAS TECNOLOGÍAS FRONT-END

Los lenguajes actuales que los navegadores son capaces de interpretar son:

- **HTML.** No es un lenguaje de programación, sino un formato de documentos de texto que incluye etiquetas especiales para dar significado al contenido. Se trata, pues, de un lenguaje de marcado. Es la base de las aplicaciones web y se encarga de la parte semántica de la aplicación. Es decir, de dar estructura conceptual al contenido de la página web.

- **CSS.** Es un lenguaje de diseño encargado de determinar el formato y la maquetación de los elementos de un documento HTML. Si HTML se encarga de dar semántica, CSS se encarga de la apariencia y la presentación visual.

- **JavaScript.** Es un lenguaje de programación que, en cuanto a su función en una aplicación web, se encarga de dar dinamismo a la página web. Su potencia se ha multiplicado en estos últimos años, convirtiendo a los navegadores en verdaderas máquinas virtuales que traducen este lenguaje a toda velocidad. Indudablemente es la base de la creación de aplicaciones web

Estas son las tres tecnologías principales y obligadas en toda creación de aplicaciones web en la actualidad. A lo largo del tiempo, sin embargo, han existido y existen otras tecnologías:

- **XML.** Hace unos años se consideró a XML como el lenguaje de marcado que acabaría con el propio HTML. No fue así, pero sigue siendo una tecnología a tener en cuenta. Se trata de un lenguaje de marcado que permite definir formatos documentales a voluntad, y que se nutre de una serie de tecnologías anexas capaces de dar formato, dinamismo y otras capacidades a los documentos XML. En la actualidad, en cuanto a la creación de aplicaciones, se utiliza principalmente como formato de intercambio de datos entre aplicaciones y servicios. La mayoría de navegadores comprenden este lenguaje.

- **JSON.** Es un formato documental procedente del lenguaje JavaScript (en este mismo libro se detalla su funcionamiento en temas posteriores) y que, en gran medida, está reemplazando a XML como formato de intercambio de datos. Los navegadores no comprenden este formato de manera nativa, pero existen numerosos ***plugins*** (software que se añade al navegador para otorgarle funcionalidades extra) que permiten visualizar adecuadamente este tipo de documentos en el navegador.

- **SVG.** Es un lenguaje basado en XML que permite mostrar imágenes vectoriales en los navegadores. Actualmente está implantado y reconocido en todos los navegadores actuales.

- **Flash.** Es una tecnología que se creó para conseguir crear, de forma eficiente y rápida, aplicaciones web enriquecidas. Tuvo un enorme éxito desde finales de los años 90 del siglo pasado, pero actualmente está en claro desuso. Requería la instalación de un plugin en el navegador. Su popularidad hizo que dicho plugin estuviera incorporado de base.

  Actualmente está en claro desuso, solo algunas páginas utilizan esta tecnología, por las quejas sobre su rendimiento y, sobre todo, por las capacidades que actualmente poseen los lenguajes HTML, CSS y JavaScript.

- **Applets.** Es otra tecnología en práctica extinción. Se trataba de componentes creados en lenguaje java que aprovechaban la potencia de este lenguaje para dotar de capacidades avanzadas a las aplicaciones web. Como en el caso de Flash, requiere de un plugin especial. La potencia actual del lenguaje JavaScript ha relegado completamente a esta tecnología, dejando su uso solo para aplicaciones muy concretas (y en muchos casos obsoletas).

Actualmente se habla de aplicaciones simplemente HTML5, porque la última versión de la norma es la cinco, y se da por hecho que HTML5 incluye HTML, CSS y JavaScript. Esto simplifica mucho lo que un programador de aplicaciones web (en el lado del cliente) debe de aprender.

Los lenguajes se van modernizando y adoptando nuevas funcionalidades que los navegadores van incorporando. Se exige a los navegadores que tengan capacidad, por ejemplo, de mostrar imágenes en varios formatos, reproducir vídeo, mostrar notificaciones al usuario e incluso manejar periféricos como cámaras y micrófonos.

En definitiva, el navegador se ha convertido en un software que simula ser toda una máquina de trabajo comparable a las máquinas reales. Por ello es posible crear aplicaciones web totalmente comparables a las aplicaciones clásicas de escritorio.

1.2.2 EL TRIUNFO DE JAVASCRIPT

1.2.2.1 INICIOS DE JAVASCRIPT

El desarrollo de aplicaciones web ha dispuesto de varias tecnologías front-end a lo largo de la historia. Pero lo cierto es que, por encima de todas, ha destacado JavaScript hasta ser el único lenguaje de programación (HTML y CSS no son lenguajes de programación) que los navegadores son capaces de traducir sin necesidad de añadir plugins o extensiones.

lnicialmente JavaScript se ideó para dotar de dinamismo a las páginas web. Las primeras páginas web se hacían en HTML y, aunque se añadió las primeras versiones de CSS, lo que las páginas web podían hacer era todavía muy poco. Eran simplemente páginas estáticas con algunas opciones de formato: colores, tamaños de letra, tablas y pocas cosas más.

Consciente del problema, la empresa **Netscape** decidió crear un lenguaje para que su navegador **Navigator pudiera dar dinamismo a las páginas web.** Inicialmente su nombre fue **LiveScript** (anteriormente se había llamado **Mocha),** pero el éxito que tuvo el lenguaje **Java** de la empresa **Sun Microsystems** creado en esa misma época hizo que se cambiara el nombre por **JavaScript,** aunque realmente sea muy diferente del lenguaje Java.

Por lo tanto, **Java** es un lenguaje que, inicialmente, era propiedad de Sun Microsystems que poseía los derechos de explotación de la marca ***Java.*** Netscape pidió permiso para utilizar la marca JavaScript a la propia Sun Microsystems, la cual se hizo propietaria de la marca JavaScript pero permitió su uso a Netscape. Eso significó que solo Netscape podía usar ese nombre para el lenguaje.

JavaScript tuvo un éxito inmediato y las compañías que fabricaban otros navegadores adoptaron el lenguaje. Microsoft desarrolló su propia versión a la que llamó **JScript** y otras empresas crearon otros (como **ActionScript** creado por **Macromedia).** El hecho de que los navegadores Microsoft Internet Explorer y Netscape Navigator coparan el mercado hizo que ambas versiones, que tenían bastantes diferencias, compitieran entre sí creando un problema a los desarrolladores de aplicaciones web.

1.2.2.2 NORMALIZACIÓN DE JAVASCRIPT

Durante mucho tiempo la tecnología de creación de páginas web se llamó **DHTML** (acrónimo de ***Dynamic HTML)*** porque con JavaScript las paginas eran ya dinámicas y empezaban a reaccionar a las acciones del usuario. Pero en estos años en los que se desarrolló la llamada ***Guerra de navegadores*** entre **Microsoft** y **Netscape** (que finalmente ganó Microsoft) se agravó el problema de las diferentes versiones de JavaScript.

Por ello se empezó a tratar la cuestión de estandarizar el lenguaje. La **World Wide Consortium,** organismo de estándares para las tecnologías relacionadas con HTML y, sobre todo **ECMA,** organismo de estándares para la comunicación e información decidieron estandarizar el lenguaje. ECMA normalizó el lenguaje por primera vez en 1997, llamándole **ECMAScript.**

Actualmente el nombre JavaScript, como marca registrada, es propiedad de **Oracle Corporation** al adquirir a la empresa Sun Microsystems. Netscape se disolvió a principios de siglo y dejó el proyecto como proyecto de código abierto impulsado por la **Fundación Mozilla.** Por ello, esta fundación obtuvo el derecho de usar también el nombre JavaScript.

Dejando de lado este tema del nombre, aunque es interesante conocer su situación legal, lo cierto es que comúnmente el lenguaje se conoce como JavaScript. Se suele hacer referencia a ECMAScript para hablar del estándar actual de JavaScript.

La versión 5 del estándar (ECMAScript 5) fue totalmente adoptada por todos los navegadores en torno al año 2012, lo que mitigó el problema de las diferentes versiones para cada navegador. Como veremos más adelante, la ECMA ha seguido publicando estándares que, poco a poco, la industria va adoptando.

1.2.2.3 APLICACIONES ENRIQUECIDAS

Google desarrolló el navegador **Chrome** y consiguió crear un motor, llamado **V8,** capaz de ejecutar JavaScript a una gran velocidad. El crecimiento continuo de JavaScript hizo que aparecieran aplicaciones web que ofrecían una experiencia de usuario semejante a la de las aplicaciones de escritorio. Se las llamó **RLA** ***(Rich Internet Applications)*** y en su potencia fue fundamental el aumento de las capacidades y la velocidad del lenguaje JavaScript.

Se vieron favorecidas con el desarrollo de HTML5, estándar actual de la tecnología frontend que incluyó la tercera versión de CSS junto con la modernización del lenguaje HTML (en detrimento del designado como sustituto, XHTML), además del impulso definitivo de javaScript como único lenguaje de ejecución en el lado del navegador.

1.2.2.4 JAVASCRIPT SALE DEL NAVEGADOR

Todo ello ha contribuido a desarrollar aun más javaScript y a tener un éxito absoluto como pieza central del desarrollo de aplicaciones web. Así en el año 2009, Ryan Dahl desarrollo un software llamado **Node.js** o simplemente **Node.**

Puesto que el motor V8 de Google que servía para traducir JavaScript en Chrome era un proyecto de código abierto, Dahl decidió utilizar el código para crear un entorno independiente del navegador, capaz de interpretar código JavaScript. Esto ha permitido, entre otras posibilidades, crear aplicaciones JavaScript en el lado del servidor.

Fue otro éxito inmediato que ha contribuido a ver a JavaScript como un lenguaje completo capaz de crear aplicaciones de todo tipo y que podía rivalizar con los lenguajes clásicos de programación.

El impulso dado por Node.js con ayuda de las múltiples librerías y frameworks de las que se puede dotar ha permitido a JavaScript ser un lenguaje capaz de:

Crear aplicaciones de escritorio con ayuda de frameworks como **Electrón** o **NW.js.**

Programar dispositivos hardware con ayuda de frameworks como **Johnny-Five, que** tiene mucho éxito para programar placas **Arduino, Cylon.js** o **Node-Red.**

Generar aplicaciones móviles nativas. Las aplicaciones móviles se crean usando lenguajes específicos: por ejemplo, Java, en dispositivos **Android,** y **Swift** u **Objective C** en dispositivos **iOS.** Hoy en día es posible programar una aplicación HTML5 usando frameworks de JavaScript que permiten manipular el hardware del dispositivo. Algunos frameworks populares de este tipo son: **PhoneGap/Cordova, lonic, Flutter** o **React Native.**

- Ser el lenguaje de manipulación de Sistemas de Bases de Datos. Como ocurre por ejemplo con los sistemas **MongoDB.**

En definitiva, el salto de JavaScript para ser un lenguaje multipropósito, más allá de ser el lenguaje fundamental de creación de aplicaciones web, es total.

1.2.2.5 JSON

JSON es el acrónimo de ***JavaScript Object Notation,*** Notación de Objetos de JavaScript. Se trata de un formato de documento, procedente de la forma en la que se definen los objetos en JavaScript. JavaScript es un lenguaje capaz (como veremos en temas posteriores) de manipular objetos y se diseñó en el lenguaje una forma de definir objetos.

La popularidad de ese formato ha sido tal que actualmente ha traspasado al propio lenguaje y se le considera un formato de intercambio de datos independiente de JavaScript. Es decir, es una

forma de definir datos que ha resultado tan potente y eficiente que ahora muchos lenguajes son capaces de leer y manipular datos en formato JSON de manera nativa.

Es más, en la actualidad está arrinconando completamente al lenguaje XML como formato preferente de intercambio de datos.

1.3 VERSIONES DE JAVASCRIPT

1.3.1 EL PROBLEMA DE LA COMPATIBILIDAD

Hemos visto que los desarrolladores, desde el primer momento, han tenido problemas con el tema de la compatibilidad. Por otro lado, a medida que ha crecido el uso del lenguaje por parte de los desarrolladores de aplicaciones web, se le han ido exigiendo más funcionalidades y, sobre todo, más capacidades para ser usado como lenguaje completo de trabajo.

Ocurrió que las mejoras al lenguaje las incorporaban las empresas sin consensuar en absoluto dichas mejoras. La razón era que los fabricantes de navegadores dispusieran de un lenguaje mejor que el de sus rivales.

Sin embargo, la estandarización del lenguaje comenzó desde el primero momento ya que Netscape envió el borrador de JavaScript a ECMA para la estandarización. La norma **ECMA-262** es la que se relaciona con la estandarización del lenguaje. El lenguaje estandarizado se acordó que se llamara **ECMAScript.**

Pero, aunque Microsoft y Netscape se comprometieron con el estándar, lo cierto es que **JavaScript** y **JScript** continuaron añadiendo especificaciones al lenguaje.

Todo cambio con la versión 5 del lenguaje ECMAScript que apareció en el año 2009 y con la aceptación de HTML 5 como estándar. Esta versión y en especial su sucesora, la versión 5.1 del año 2011, fue adoptándose por todos los navegadores y finalmente Microsoft respetó este estándar en las últimas versiones de Internet Explorer y más aún con la aparición de Edge.

Cuando en 2014 la W3C publicó el estándar HTML 5, también JavaScript en su versión 5.1 fue adoptado masivamente. El impulso definitivo como estándar lo dio (también en el 2011) la agencia mundial de estándares **1SO** junto con **1EC** ***(Comisión de Electrotecnia Internacional)*** normalizó el lenguaje en la norma **1SO/IEC 16262:2011.**

Dejando de lado las cuestiones formales, se dispuso de un estándar ampliamente adoptado y respaldado por los principales organismos.

Eso podría haber significado el final de la historia, pero los lenguajes de programación están vivos y se siguen mejorando.

1.3.2 LENGUAJES PREPROCESADOS

Las necesidades de los programadores empezaron a influir en empresas y otras entidades para crear nuevos lenguajes que mejoraran el lenguaje original.

El primero en conseguir popularidad fue **CoffeeScript. Permitió dotar a JavaScript de una sintaxis** inspirada en el lenguaje **Python** para poder escribir JavaScript de forma más rápida y eficaz. Actualmente está en claro retroceso, pero durante muchos años fue utilizado por miles de programadores.

CoffeScript no respeta los estándares porque, de hecho, es un lenguaje nuevo. Eso significa que ningún navegador, de forma nativa, es capaz de interpretar código CoffeeScript. Por ello el código debe ser convertido a JavaScript.



Figura 1.4: Funcionamiento de un preprocesador JavaScript

Esta es la idea de un lenguaje preprocesado, se escribe código en un lenguaje y se convierte con ayuda de un software especial (conocido simplemente como **preprocesador)** a JavaScript estándar.

Bajo esa misma idea, en el año 2012 apareció **TypeScript,** lenguaje creado por Microsoft que amplía JavaScript para que reconozca, entre otras muchas cosas, tipos de datos avanzados y las ventajas de los lenguajes fuertemente tipados. Ha tenido mucho éxito y es todavía muy utilizado, más aun al ser el lenguaje base del exitoso framework **Angular** de Google.

La misma idea está detrás del lenguaje **Dart** desarrollado por Google para modernizar JavaScript, Elm que permite programar en un lenguaje puramente funcional o los intentos de **ClojureScript** o **Scala.js** para convertir código en los lenguajes **Clojure** y **Scala,** respectivamente, en código JavaScript.

1.3.3 ESTÁNDARES A PARTIR DEL AÑO 2015

La influencia de los lenguajes preprocesados para JavaScript y las claras necesidades de mejorar el lenguaje, dio lugar a nuevos estándares.

Así en el año 2015 apareció la norma **ECMAScript 6** (conocida como **ES6** o **ECMAScript 2015)** que mejoró mucho el lenguaje JavaScript al dotarlo de elementos avanzados de otros lenguajes.

Han aparecido, hasta el momento de escribir este texto, las versiones siete **(ES7** o **ECMAScript 2016),** ocho **(ES8** o **ECMAScript 2017)** y nueve **(ES9** o **ECMAScript 2018).** Y se esperan nuevas versiones en los años venideros.

La idea, al igual que ocurre con la propia norma HTML 5, es que los estándares sean vivos y sigan incorporando mejoras al lenguaje de forma veloz.

Actualmente, en el momento de escribir estas líneas, todos los navegadores (que estén actualizados) trabajan perfectamente con todo el estándar ES2015 y con gran parte del estándar ES2016. Los siguientes estándares aportan menos mejoras y por ello serán rápidamente adoptadas por los navegadores.

1.3.4 SITUACIÓN ACTUAL DE JAVASCRIPT

Ante tanta variedad de lenguajes y normas, qué lenguaje aprender. Esta es la gran duda. Puede que la respuesta la disponga la página http://cainuse.com



Figura 1.5: Implantación del estándar JavaScript ES2015, según la página ***Canlüse***

Esta página nos permite saber hasta qué punto un aspecto de JavaScript (o de CSS) está implantado en los navegadores. Hoy en día es exigible que los usuarios tengan el navegador actualizado, pero debemos saber qué navegadores son compatibles con nuestro código y esa página es una clara ayuda.

La otra duda está en si utilizar o no lenguajes preprocesados u otras tecnologías facilitadoras, como **jQuery.** Lo aconsejable es aprender el lenguaje estándar para conseguir los mejores fundamentos posibles. Cuando se consigan estas bases, es verdad que hoy en día un programador front-end debe adquirir más habilidades que le permitan trabajar de forma rápida, eficiente y mantenible. Parte de estas técnicas se explican en este mismo libro en los capítulos finales.

1.4 RESUMEN DE LA UNIDAD

- Existen dos tipos tecnologías en el desarrollo de aplicaciones web: tecnologías front-end y tecnologías back-end
- Los lenguajes front-end se relacionan con la capa de presentación de la aplicación web. Su código es visible por los usuarios ya que se envía a sus navegadores. El código back-end, el del lado del servidor, se interpreta fuera del ordenador del usuario: queda fuera de su visión.
- HTML (semántica), CSS (formato) y JavaScript (interactividad) son los tres lenguajes front-end que todos los navegadores son capaces de traducir para presentar la apariencia final al usuario. Solo JavaScript es realmente un lenguaje de creación de aplicaciones.
- El lenguaje JavaScript ha quedado como el único lenguaje para programar el front-end.
- El éxito de JavaScript entre los desarrolladores le ha permitido ser utilizado en todo tipo de aplicaciones (no solo aplicaciones web).
- La estandarización ha sido problemática, pero en la actualidad los estándares de la ECMA (cuyo lenguaje de forma estándar se llama ECMAScritp) son ampliamente adoptados.

1.5 TEST DE REPASO

1.- **Cuál de estas tecnologías no es front-end:**

a) HTML
b) PHP
c) JavaScript
d) CSS

2.- ¿Qué es ES6?

a) Un lenguaje del lado del servidor
b) Una versión no estándar de JavaScript
c) El lenguaje estándar JavaScript en su versión 6
d) Un servicio (en sexta versión) que permite verificar si un navegador realmente es compatible con JavaScript.

3.- **Cuál de esta afirmaciones no es correcta**

a) El código del back-end es visible para el usuario
b) El código back-end debe de ser interpretado por un servidor de aplicaciones
c) El código back-end no puede ser traducido por el navegador
d PHP es un lenguaje back-end

4.- ¿Qué es Node.js?

a) Un navegador estándar
b) Un servicio en la nube para alojar código
c) Un verificador de la situación actual del estándar JavaScript
d) Un entorno de ejecución de código JavaScript

5.- **Cuál de estas tecnologías no es front-end:**

a) Applets de Java
b) Flash
c) ASP.Net
d) SVG

6.- **¿Qué diferencia hay entre JavaScript y ECMAScript?**

a) JavaScript es asincrono y ECMAScript síncrono
b) JavaScript es estándar y ECMAScript no
c) ECMAScript es el nombre del estándar y JavaScript no lo es
d) JavaScript se basa en Java y ECMScript no

7.- **¿Qué hace falta para convertir código TypeScript en código JavaScript?**

a) Un preprocesador TypeScript
b) Nada, TypeScript es sinónimo de JavaScript
c) Un navegador actualizado
d) Instalar Node.js

8.- **¿Qué tipo de aplicaciones se pueden programar en JavaScript?**

a) Solo aplicaciones web front-end
b) Aplicaciones web tanto front-end como back-end
c) Aplicaciones web tanto front-end como back-end y también apps móviles
d) Todo tipo de aplicaciones

9.- **¿A qué empresa pertenece la marca registrada JavaScript?**

a) Microsoft
b) Apple
c) Google
d) Oracle

10.- **En qué capa se encuentran los servidores de bases de datos**

a) Capa de negocio
b) Capa lógica
c) Capa de presentación
d) Pueden estar en cualquier capa

UNIDAD 2

PREPARACIÓN DEL ENTORNO DE TRABAJO

OBJETIVOS

— Reconocer las principales herramientas necesarias para programar aplicaciones web en entorno cliente

— Identificar las ventajas de utilizar herramientas de trabajo adecuadas

— Implementar un entorno de trabajo adecuado para crear aplicaciones web en entorno cliente

CONTENIDOS

2.1 LA CAJA DE HERRAMIENTAS DEL DESARROLLADOR FRONT-END

2.2 NAVEGADORES

2.3 EDITORES DE CÓDIGO
- 2.3.1 AYUDAS DE LOS EDITORES DE CÓDIGO
- 2.3.2 TIPOS DE EDITORES DE CÓDIGO
- 2.3.3 INSTALACIÓN DE VISUAL STUDIO CODE
- 2.3.4 USO DE VISUAL STUDIO CODE
- 2.3.5 CONFIGURACIÓN DE VISUAL STUDIO CODE

2.4 NODE.JS Y NPM
- 2.4.1 ¿QUÉ ES NODE.JS?
- 2.4.2 INSTALAR NODE.JS
- 2.4.3 NVM
- 2.4.4 NPM

2.5 GIT
- 2.5.1 SISTEMAS DE CONTROL DE VERSIONES Y GIT
- 2.5.2 INSTALACIÓN DE GIT
- 2.5.3 FUNCIONAMIENTO BÁSICO DE GIT
- 2.5.4 GITHUB. GIT EN REMOTO

2.6 PRÁCTICAS RESUELTAS

2.7 RESUMEN DE LA UNIDAD

2.8 TEST DE REPASO

2.1 LA CAJA DE HERRAMIENTAS DEL DESARROLLADOR FRONT-END

En esta unidad vamos a realizar un recorrido por herramientas y elementos que se requieren para programar aplicaciones web en el lado del cliente. Se trata de una unidad previa al conocimiento de los lenguajes y componentes que permiten programar aplicaciones. La intención es conseguir un entorno de trabajo que nos permita desarrollar aplicaciones de forma cómoda. Pero es cierto, que alguna de las herramientas que detallamos en esta unidad no vamos a poder utilizarlas en su completa dimensión hasta que no conozcamos más a fondo el lenguaje JavaScript.

Por lo comentado antes, nuestro consejo es volver a esta unidad más adelante, cuando hayamos recorrido el lenguaje JavaScript. Por otro lado, puede ser que el lector de este libro ya tenga un entorno de trabajo a su medida, entonces tampoco hay problema en pasar directamente a la siguiente unidad.

En general un desarrollador de aplicaciones web front-end necesitará:

- Varios **navegadores.** Si indicamos ***varios*** es porque no debería bastarnos con probar nuestro código en un solo navegador.
- Un **editor de código.** Es la primera herramienta que necesita un desarrollador. La mayor parte de horas de trabajo las pasaremos en la pantalla que proporcione el editor. Elegir un buen editor de código es la primera gran decisión. Junto con el navegador, es lo mínimo que necesitamos para trabajar. Nuestra elección en este libro es el editor gratuito más utilizado que es **Visual Studio Code.**
- Un **intérprete independiente** de JavaScript. Al decir ***independiente*** queremos decir que permite ejecutar JavaScript fuera del navegador. Eso permite abordar y aprender las bases del lenguaje de forma más rápida. **Node.js** es la opción lógica en este aspecto, además aporta el sistema de gestión de paquetes **npm.**
- Gestor de paquetes para instalar herramientas y componentes. El uso de frameworks y componentes como **jQuery, React, Vue** o **Angular** entre muchos otros, así como herramientas de gestión de tareas como **Gulp** o **Grunt,** precompiladores como **Babel, Sass** o **TypeScript** y otros muchos componentes y elementos, se suele gestionar gracias a la ayuda de los gestores de paquetes; **npm** es el más popular porque está integrado en **Node.js** pero hay otras alternativas como, por ejemplo, **yarn.**
- **Sistema de control de versiones.** Se trata de software integrado, normalmente, en la línea de comandos del sistema. Permite poder tener diferentes versiones de nuestro código para poder acudir a ellas cuando deseemos. Actualmente, **Git** es el sistema más popular.
- Repositorio, con **sistema de control de versiones integrado, en la nube.** Se trata de un servicio que permita alojar nuestro código manteniendo las diferentes versiones que están siendo gestionadas por nuestro sistema de control de versiones. Indudablemente, **GitHub** es el servicio más popular.

2.2 NAVEGADORES

Todo desarrollador necesita instalar varios navegadores. En el momento de escribir estas páginas, **Google Chrome** es claramente el navegador más utilizado en todo el mundo. La probabilidad de que un usuario utilice Google Chrome como navegador es muy alta (más de un 60%) y hay que tener además en cuenta, que muchos otros navegadores (como **Microsoft Edge** u **Opera)** se basan en el proyecto Chromium de Google. Por lo tanto, es vital que probemos nuestras aplicaciones en ese navegador.

Pero hay un porcentaje importante de personas que utilizan otros navegadores como **Mozilla Firefox** o **Apple Safari.** Cuantos más navegadores instalemos para probar nuestra aplicación, más seguros estaremos de que nuestra aplicación funciona correctamente.

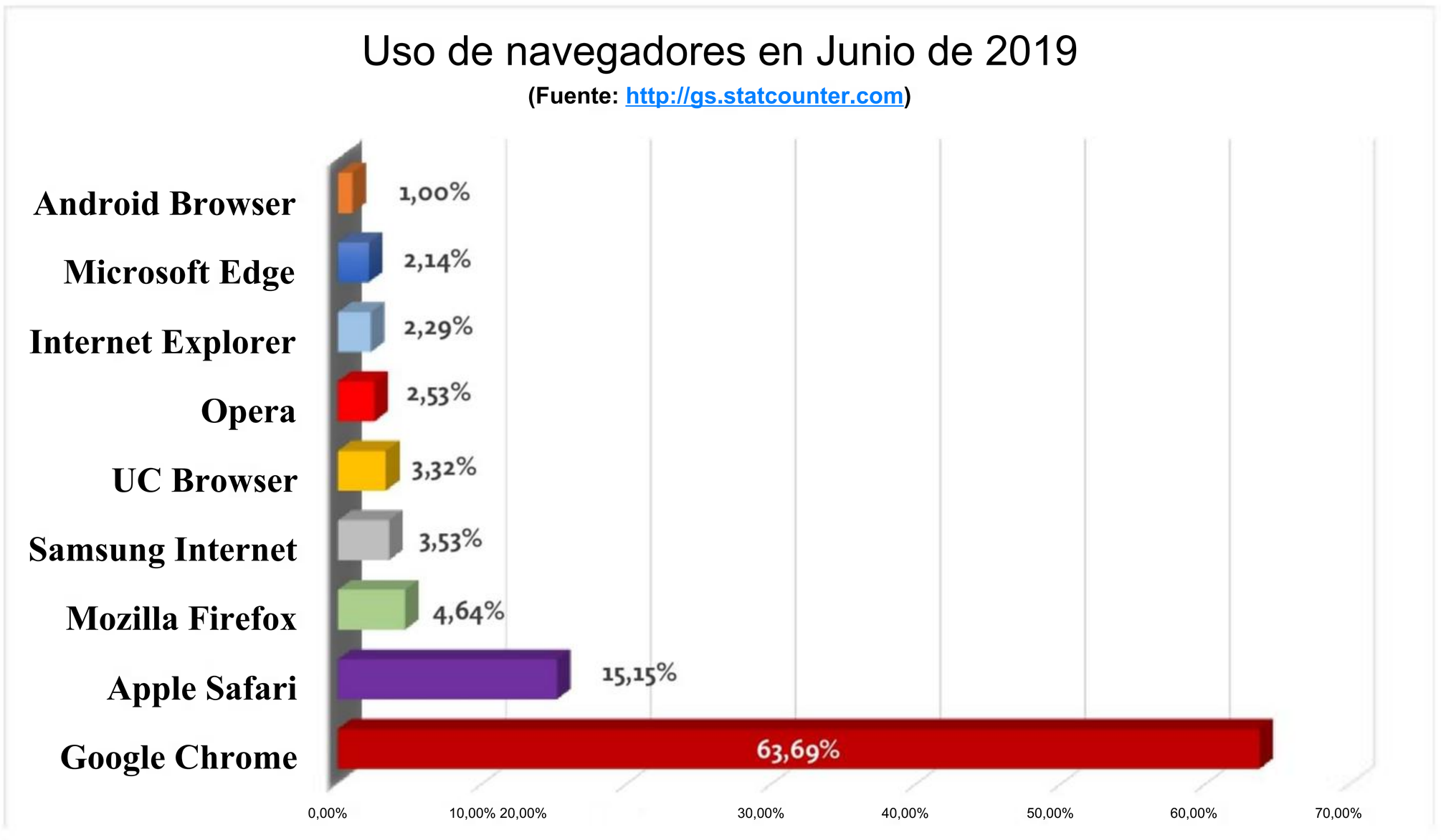


Figura 2.1: Cuotas de uso de navegadores en Junio de 2019 según la página ***gstatcounter***

Es complicado tener instalados todos los navegadores. Además hay que tener en cuenta que hay navegadores que usan el mismo motor **(Edge** de **Microsoft** y **Opera,** por ejemplo, se basan en el proyecto **Chromium** al igual que **Google Chrome)** y que, en nuestro sistema, no podremos instalar todos los navegadores; por ejemplo, **Samsung Internet** es el navegador de los Smart TV de la marca Samsung, **Safari** solo funciona en sistemas **Apple** y **UC Browser** no está disponible en sistemas de escritorio. Al final hay que pensar que el estándar del lenguaje, poco a poco es aceptado por todas las marcas. Nuestra recomendación mínima en cuanto a navegadores es instalar los siguientes:

- **Google Chrome.** Al ser el más utilizado es fundamental que la aplicación funcione principalmente en este navegador.

- **Mozilla Firefox.** Es el segundo navegador más utilizado y además el único rival real de Chrome al tener un motor diferente, tanto para interpretar JavaScript como para renderizar HTML y CSS.
- **Internet Explorer,** para saber cómo ven las páginas los usuarios que no tengan un sistema al día. Es desaconsejable utilizar este navegador hoy en día, pero hay muchos usuarios todavía que lo utilizan y por ello hay que probar, al menos, cómo verán la aplicación estos usuarios para saber si debemos avisar o no de que nuestra aplicación no funciona bien en este navegador.
- **Apple Safari,** si es posible, ya que solo funciona en dispositivos de la marca **Apple.**
- Navegadores específicos de dispositivos, si es posible, como Smart TV, móviles diferentes, etc.

Muchos fabricantes ofrecen versiones de navegadores en pruebas. Estos navegadores especiales se usan para probar características recientes que todavía no están implementadas en los navegadores que utilizan los usuarios. La idea general es la siguiente:

- Google proporciona los siguientes navegadores (aparte del Chrome definitivo):
  - **Canary.** Versión inestable para pruebas. Es el primer navegador en el que Google incorpora las novedades que vayan apareciendo en HTML, CSS y JavaScript.
  - **Dev.** Versión para programadores, es una versión más estable que la anterior. Las novedades que aún son muy inestables, no se implementan aquí.
  - **Beta.** Versión casi definitiva, se suele lanzar 15 días antes del Chrome final.
- Firefox utiliza la misma idea:
  - **Nightly.** Versión para probar novedades. Es en esta versión donde primero aparecen las características últimas de los lenguajes.
  - **Developer Edition.** Más estable y con capacidades especiales para desarrolladores. Sirve para probar las novedades sin los problemas de posible inestabilidad de la anterior
  - **Beta.** Versión previa a la definitiva para confirmar la buena implementación de nuevas características.
- Microsoft Edge tiene versiones equivalentes a las de **Chrome: Canary, Dev** y **Beta.**
- Apple Safari tiene dos versiones: **Technology Preview** (equivalente a Canary) y **Beta.**

Puede ser conveniente instalar alguno de estos navegadores (por ejemplo, **Google Chrome Dev** y **Firefox Developer Edition)** si queremos ir probando nuevas características para que, cuando se aprueben realmente en el navegador definitivo) podamos incorporarlas a nuestros proyectos.

2.3 EDITORES DE CÓDIGO

2.3.1 AYUDAS DE LOS EDITORES DE CÓDIGO

El editor de código es el primer software que todo desarrollador debe instalar para trabajar en el desarrollo de aplicaciones web. El código que se escribe para crear aplicaciones web se almacena en ficheros de texto plano. Esto significa que cualquier herramienta capaz de escribir texto es válida, incluso editores tan poco convenientes como el bloc de notas de los sistemas Windows se pueden utilizar para esta tarea.

Pero, evidentemente, un editor como el bloc de notas no es el ideal. Hay editores creados específicamente para crear código que nos van a facilitar mucho más nuestro trabajo. Las facilidades y ayudas que se nos proporcionan, difieren según el tipo de editor que utilicemos. A continuación, resumimos las más habituales y prácticas.

- **Coloreado de código.** Lo cual facilita enormemente la escritura de código porque se distinguen muy bien las palabras claves del texto normal o de los valores que toman las variables. En la mayoría de editores el coloreado es personalizable y también es común disponer de diferentes ***temas*** para elegir colorados del entorno y código que se ajusten a nuestras preferencias.

- **Facilidad para seleccionar texto.** Los procesadores de texto (como **Microsoft Word** por ejemplo) siempre han dispuesto de herramientas y combinaciones de teclado que facilitan la selección de texto. En el caso de los editores de código, estas facilidades se ajustan mejor a las necesidades de los programadores porque seleccionan sabiendo que lo que tenemos es código y no texto normal.

- **Navegación avanzada.** A veces los desarrolladores generan archivos de cientos o miles de líneas, moverse por archivos tan grandes es difícil. Por ello hay ayudas que ofrecen los editores como:

  - Expandir y comprimir bloques de código, para ocultar o mostrar texto y ver mejor el que nos interesa.
  - Mapa que muestra los elementos involucrados en el código para acceder al instante al código de un elemento en concreto.
  - Mini mapas del código que muestran una miniatura del código para, a través de ella, acceder más rápido a la zona que nos interesa.
  - Ir directamente a una línea de código o a una zona concreta de código a través del nombre de una función u objeto.

- **Búsqueda y reemplazo.** Estas son dos operaciones fundamentales. En muchos editores además se permite buscar no solo en el documento actual, sino en cualquier archivo de nuestro **proyecto de trabajo** (que puede estar formado por decenas de archivos). Además, se suelen proporcionar herramientas de reemplazo de nombres inteligentes.

Un ejemplo suele ser el caso de un programador que quiere cambiar el nombre de una variable que aparece continuamente en el código. Hay editores que al modificar el nombre, actualizan todas las apariciones del mismo en el archivo.

- **Autocorrección al escribir.** Todos los programadores cometemos errores continuamente al escribir código. Muchos editores son capaces de marcar errores, con un subrayado rojo, por ejemplo, en cuanto detectan el fallo. Esto ahorra muchísimo tiempo ya que, de esta forma, es más difícil acumular errores.

- **Uso de abreviaturas.** Son **los** llamados **Snippets,** trozos de código que sabemos que usamos a menudo y que asociamos con una abreviatura. Por ejemplo, podemos conseguir que cuando escribamos ***el*** y pulsemos el tabulador, el editor lo cambie por ***console.log*** (un comando muy habitual en JavaScript).

- **Visualización del resultado.** La mayoría de editores tienen integrada en sus menús y opciones la posibilidad de lanzar el resultado de nuestro código en el navegador. Algunos tienen posibilidades avanzadas, como crear en caliente un servidor web que permita poder examinar los cambios en la aplicación a la vez que los vamos escribiendo en el código.

- **Integración de herramientas.** Se trata de que el editor integre, sin salir del editor, herramientas de uso habitual de los programadores como el acceso al terminal del sistema, control de versiones del código, publicación del código, herramientas de test y depuración, etc.

- **Adición de extensiones y/o plugins.** Esto permite añadir funcionalidades que, inicialmente, no están disponibles en el editor y que podemos añadir a voluntad para crear un entorno de trabajo a medida.

23.2 TIPOS DE EDITORES DE CÓDIGO

- **Editores de texto multipropósito.** Son editores de texto especializados en facilitar la escritura de código fuente para diversos lenguajes. Son capaces de adaptarse al tipo código escrito. Por ejemplo, si escribimos código **JavaScript** el coloreado del texto se adapta a este lenguaje; si escribimos código PHP, el coloreado será distinto porque reconoce la sintaxis particular de este lenguaje.

Suelen ser muy rápidos y ligeros. Pero su principal ventaja es que son ideales también para otros fines, aparte de escribir aplicaciones web. Por ejemplo, editar los archivos de configuración de un sistema o escribir texto puro.

Aunque disponen, al menos de base, menos herramientas que los editores más especializados, podemos, a través de plugins y extensiones, incorporar muchas herramientas interesantes.

Editores de este tipo son **Sublime Text, Notepadd++, TextMate y Geany** (editor deL sistema **UbuntujL.**

- **Editores ligeros especializados en desarrollo web.** La frontera es muy difusa con los anteriores y, a veces, es difícil diferenciarlos. Su funcionamiento es parecido: son editores rápidos y ligeros que aumentan sus prestaciones instalando extensiones y plugins. La diferencia es que, por defecto, ya vienen preparados con herramientas y elementos que nos facilitan escribir el código de las aplicaciones web HTML, CSS y javaScript.

  Rivalizan con los anteriores y, claramente, pueden ser igual de versátiles. A esta categoría pertenecen los editores **Visual Studio Code** de **Microsoft, Atom** de **GitHub, Brackets** de **Adobe** y también editores como **Codekit** y **Coda** disponibles solo en entorno Maclntosh.

- **Entornos de desarrollo integrados (1DE).** Es software mucho más grande en tamaño y con opciones que aglutinan todas las necesidades de los programadores. Su ventaja es que no se echa en falta ninguna herramienta, ya que incorporan todo tipo de posibilidades, extras y ayudas.

  A cambio son entornos de trabajo más pesados, tardan mucho en ejecutarse, que requieren máquinas menos ligeras y que, normalmente, son de pago.

  En el ámbito de la programación frontend algunos IDEs populares son **WebStorm, Komodo, Visual Studio** o **Netbeans,** siendo el primero, WebStorm, el más utilizado actualmente.

- **Editores WYSIWYG.** No son los ideales para el desarrollo de aplicaciones web, pero pueden complementar el trabajo del desarrollador. Suelen incluir un editor de código, pero su finalidad es crear aplicaciones web viendo al instante el resultado visual. Es decir, son editores visuales de páginas web.

  Pueden proporcionar un punto de entrada para el diseño inicial de la aplicación y luego el código escribirle en otro entorno o bien en el editor incorporado. El más famoso software de este tipo es **Adobe Dreamweaver.** Otros interesantes son: el vistoso **CoffeeCup,** el ligero y fácil (y gratuito) **Mobirise** o el clásico **SeaMonkey** (de la fundación **Mozilla).**

- **Editores de código online.** Se trata de aplicaciones web que permiten la escritura de código y previsualizan el resultado al instante. Suelen incluir cierto espacio de almacenamiento para guardar el código en la nube y así acceder al mismo desde cualquier máquina. Su ventaja fundamental es que no requieren instalar nada en nuestra máquina para comprobar el funcionamiento del código.

  Los más exitosos de este género de editores son los que están pensados para crear pequeños retazos de código, que pueden incluir código en varios lenguajes como HTML, CSS y JavaScript, para hacer demostración de pequeñas funcionalidades o para acompañar de código a explicaciones y manuales. Esa última función será utilizada en este mismo libro para facilitar el acceso al código de ejemplo por parte del lector.

  JSFiddle, CodePen, JS Bin o **CSS Deck** son algunos de las más populares.

2.3.3 INSTALACIÓN DE VISUAL STUDIO CODE

2.3.3.1 VENTAJAS DE VISUAL STUDIO CODE

Visual Studio Code es, quizá, el editor gratuito más utilizado para escribir código. Se trata de un software creado para crear aplicaciones web, pero que se utiliza actualmente para crear todo tipo de aplicaciones gracias al gran número de extensiones que posee.

Está disponible para ordenadores en entornos Windows, MacOS y también Linux, lo que le permite ser muy utilizado en todo tipo de entornos.

Entre sus principales ventajas están:

- **La tecnología InteliSense.** Se trata de una tecnología de Microsoft creada para sus entornos de desarrollo ya hace bastantes años, en especial ha sido uno de los buques insignias de **Visual Studio** (el entorno profesional de Microsoft, que en realidad tiene otra filosofía de trabajo). Esta tecnología es la que permite que el editor nos ayude a escribir aportando sugerencias a medida que escribimos para facilitar la escritura correcta de nuestro código.

- **Depuración.** Permite depurar el código desde el propio editor. Las herramientas de depuración nos permiten hacer pruebas con el código para facilitar la detección de errores.

- **Control de versiones mediante Git.** El control de versiones, cuya utilidad explicamos más adelante, está integrado en el propio entorno. Tiene una integración total de comandos **Git.**

- **Extensiones.** Visual Studio Code dispone de un gran número de extensiones que permiten hacer que el editor integre todo lo necesario para trabajar cómodos con él. Inicialmente Visual Studio Code tiene preinstaladas pocas extensiones para que nosotros seleccionemos las que nos interesan más.

2.3.3.2 INSTALACIÓN

Desde la página: https://code.visualstudio.com/download podemos seleccionar la versión, dependiendo de nuestro sistema, que deseamos descargar.

Una vez descargado el instalador, seguiremos sus pasos y podremos utilizar el editor para trabajar.

ACTIVIDAD 2.1: INSTALACIÓN DE VISUAL STUDIO CODE

- Realiza la práctica Práctica 2.1 en la página 34.

2.3.4 USO DE VISUAL STUDIO CODE

No pretendemos en este libro publicar un manual de uso del editor, pero al menos hay que destacar algunos aspectos mínimos para empezar a trabajar.

Visual Studio Code posee un menú en la parte superior que contiene la mayoría de opciones de trabajo. El área principal de trabajo la ocupará el archivo en el que estemos trabajando.

En el lado izquierdo aparece un menú vertical con cinco iconos. Estos iconos permiten cambiar la vista principal. Estos iconos son:

ICONO	USO
0	**Vista de explorador.** Permite ver los archivos y carpetas del proyecto de trabajo actual. Desde esta vista podemos abrir los archivos deseados, borrarlos, cambiar el nombre, moverlos, copiarlos, etc.
p	**Buscar.** Sirve para buscar textos por el archivo actual o incluso por otros archivos. Dispone de opciones muy avanzadas para realizar esa búsqueda.
V	**Control de código fuente.** Permite el trabajo con el sistema de control de versiones de forma integrada en el editor. Más adelante hablaremos de estos sistemas (véase 2.5 ***"G1T",*** en la página 30).
	Debug. Acceso a la herramienta integrada de depuración de programas. La depuración facilita la corrección y detección de errores.
	Extensions. Permite añadir y quitar nuevos componentes al editor.

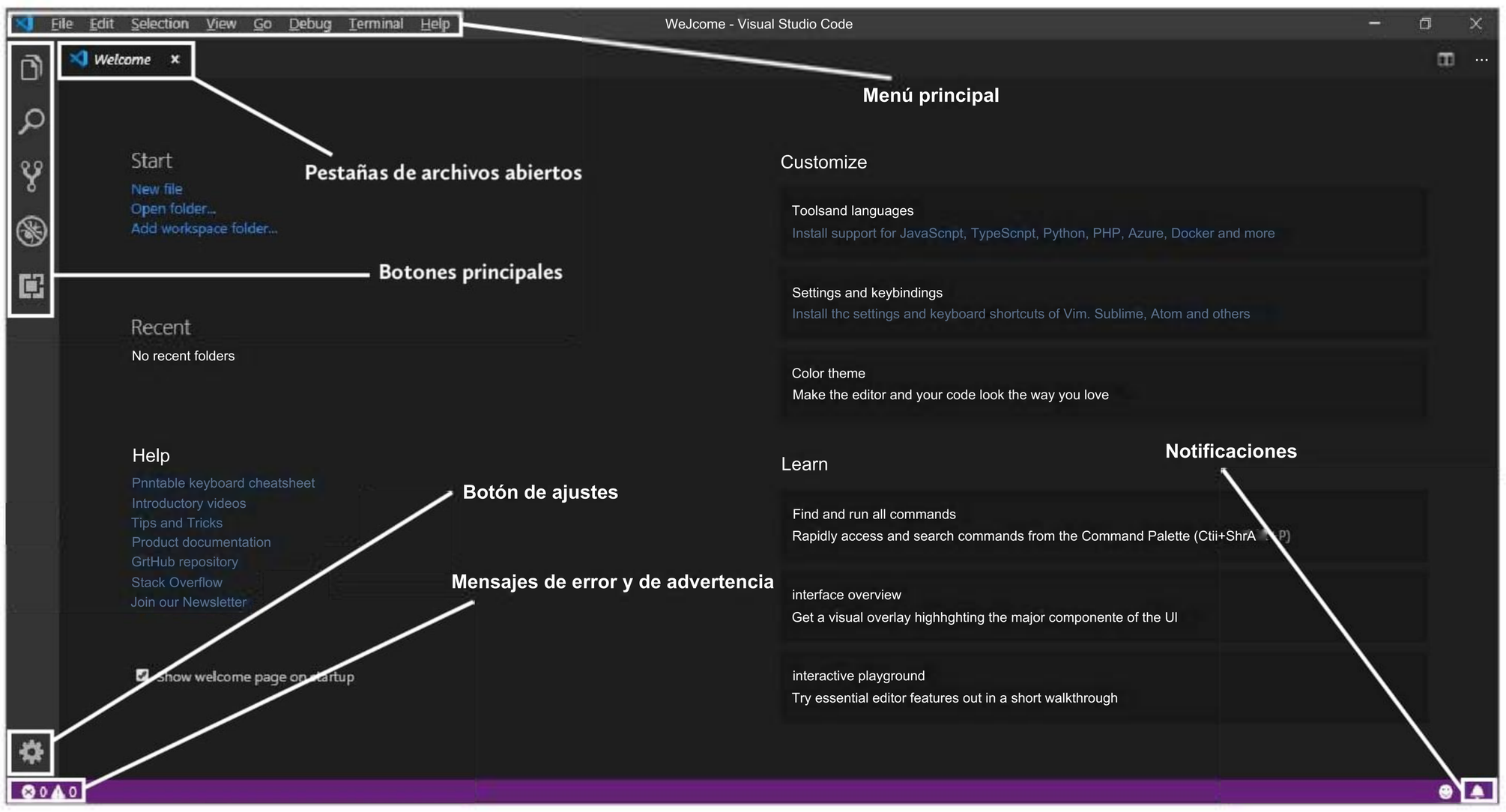


Figura 2.2: Elementos de la pantalla de Visual Studio Code

Cuando se trabaja en una aplicación web, todo el código de la misma se coloca dentro de un mismo directorio. Para que Visual Studio Code trabaje con ese directorio, deberemos elegir

Archivo - Abrir Carpeta ***(File - Open Folder)*** en los menús y después elegir el directorio raíz de nuestro trabajo. Lo habitual es al menos haber creado el directorio de trabajo.

Una vez abierto el directorio podremos crear más archivos y directorios y realizar todas las operaciones que deseemos sobre los archivos. Si estamos en el explorador de archivos (primer botón de la barra lateral) aparecerá nuestro directorio y su contenido (si es que hay contenido), arrimando el botón al nombre del directorio aparecen botones para crear archivos y carpetas.

El área principal lo ocupará el archivo en el que estemos trabajando. Hacer doble clic en el archivo lo abre y se muestra su contenido en el área principal. Podemos abrir más y veremos como cada archivo abierto ocupa una pestaña en la zona superior.

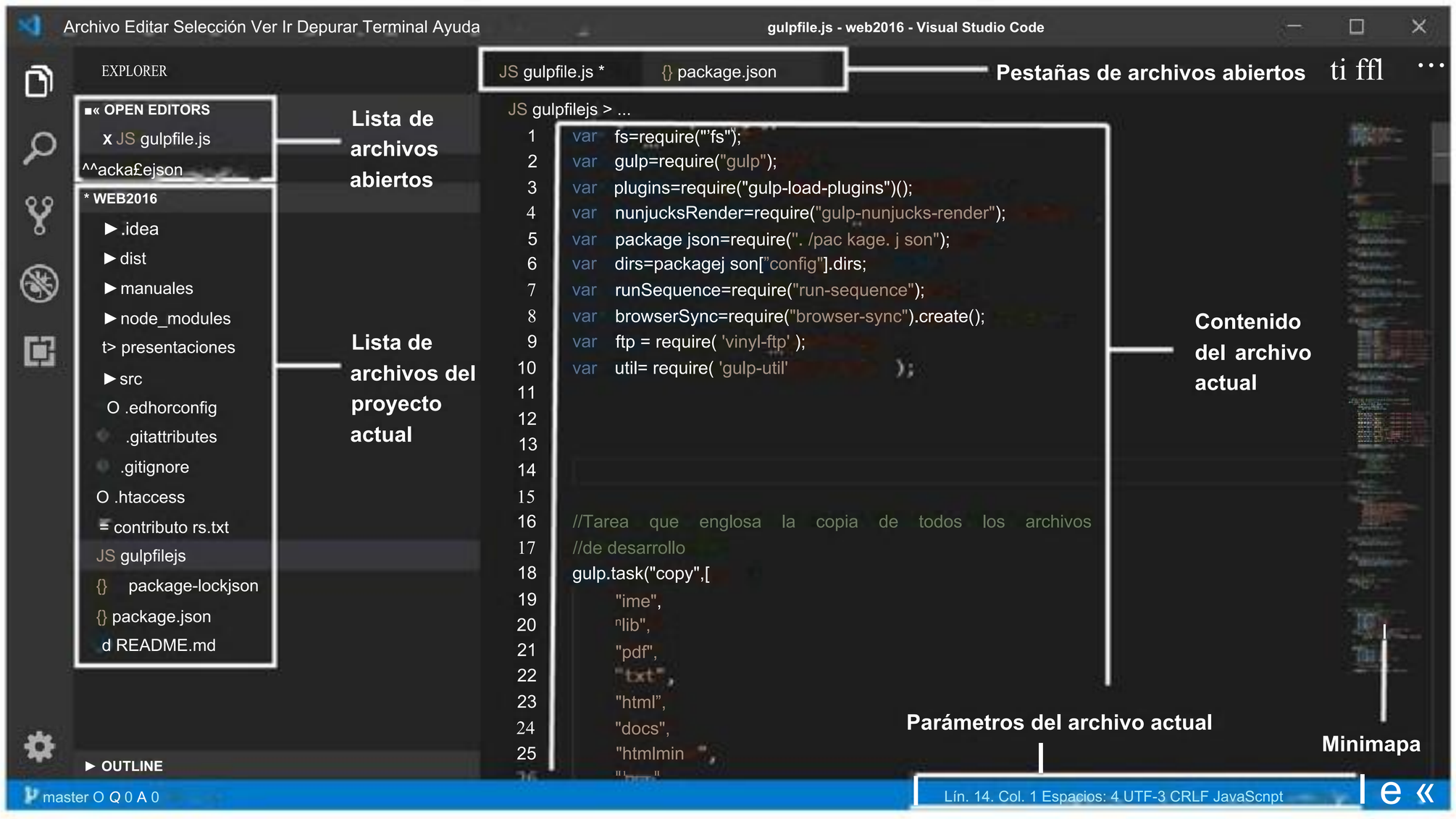


Figura 2.3: Proyecto abierto en Visual Studio Code con sus elementos resaltados

Bastará con escribir el código y guardar los cambios para poder trabajar. Una opción muy interesante es acudir a la paleta de comandos mediante el mentí **Ver - Paleta de comandos** ***(View-Command Pallette)*** o pulsar la combinación de teclas **Ctrl+Mayús+P.** Esto muestra una ventana que nos permite escribir y buscar cualquier operación disponible en Visual Studio Code. En la imagen siguiente estamos buscando el comando relacionado con mostrar el terminal del sistema y, para ello hemos mostrado la paleta de comandos y escrito **View Terminal:**



Figura 2.4: Paleta de comandos de Visual Studio Code

En la imagen se puede apreciar que no hace falta escribir el término entero, Visual Studio Code va buscando el comando apropiado a medida que escribimos. Además, se nos muestra, si es posible, la combinación de teclas equivalente.

Visual Studio Code tiene otras muchas opciones magníficas de trabajo, pero no vamos a extendernos en explicarlas todas dado que éste no es un manual de manejo de Visual Studio Code. En todo caso, podemos acudir a la documentación oficial de Microsoft en la dirección:

https://code.visualstudio.com/docs.

ACTIVIDAD 2.2: INICIÁNDOSE CON VISUAL STUDIO CODE

- Realiza la Práctica 2.2 ***"Iniciándose con Visual Studio Code",*** en la página 38.

2.3.5 CONFIGURACIÓN DE VISUAL STUDIO CODE

2.3.5.1 CONFIGURACIÓN DEL ENTORNO

Como hemos visto, la instalación de Visual Studio Code permite empezar a trabajar con el editor directamente, sin embargo, es conveniente personalizar el entorno para poder trabajar de forma más cómoda y eficiente. Aunque no vamos a explicar el funcionamiento completo del programa, sí es conveniente dar las pistas mínimas para poder entender cómo se configura el entorno.

La configuración atañe a diversos aspectos como: el tipo de letra y tamaño, a la apariencia general, al funcionamiento de los componentes del editor, a la manera de responder del editor al escribir en cada lenguaje, a decidir cuánta ayuda queremos que nos preste el editor, etc.

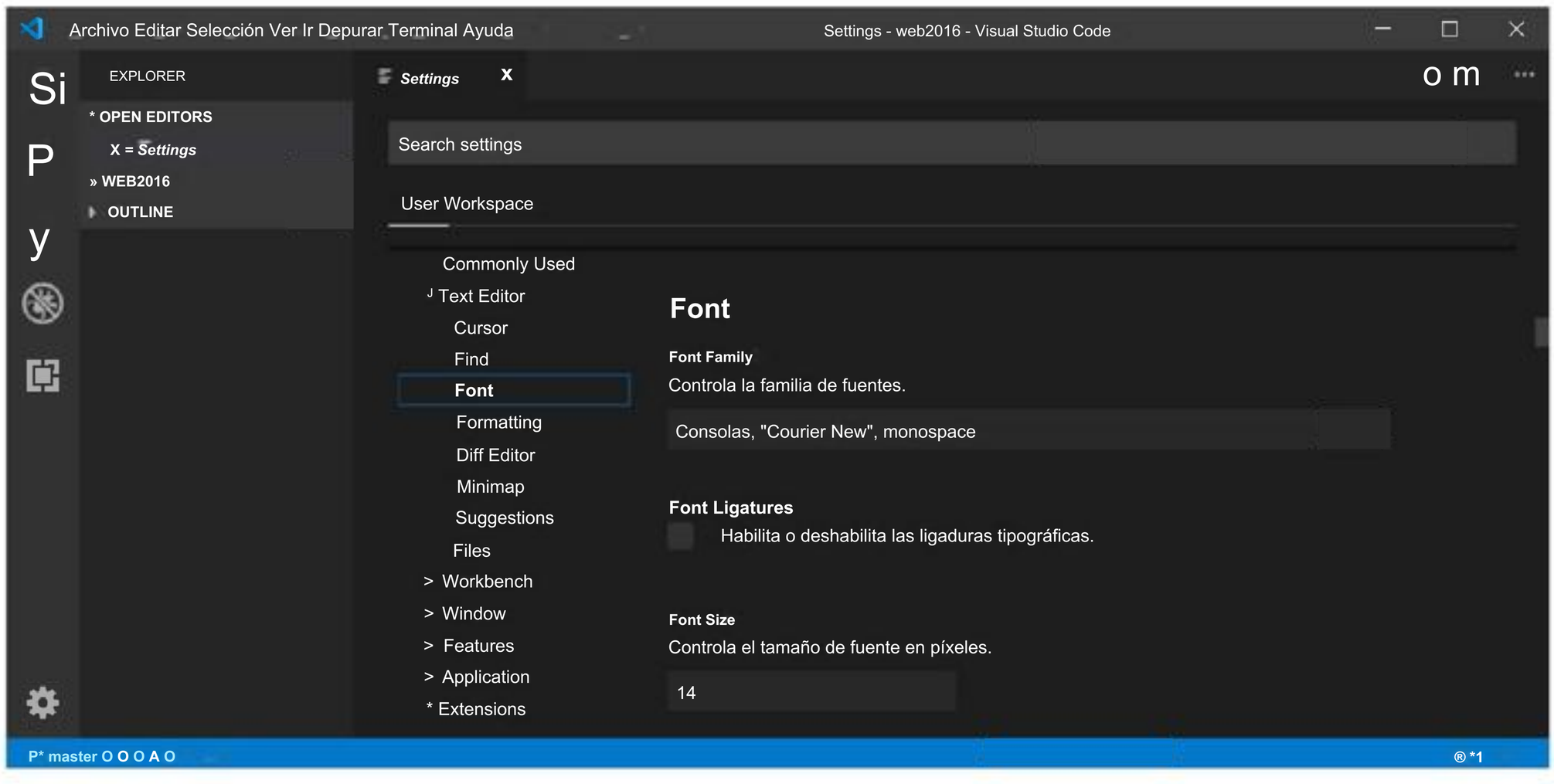


Figura 2.5: Ajustes de Visual Studio Code mostrando las opciones para modificar la letra

Realmente hay dos tipos de ajustes:

- **Ajustes de usuario *(user settings).*** Afectan a todo el funcionamiento de Visual Studio Code independientemente del proyecto en el que estemos trabajando.
- **Ajustes del espacio de trabajo *(workspace settings).*** Afectan solo al proyecto (espacio de trabajo) actual. Estos ajustes tienen preferencia sobre los del usuario.

Para llegar a los ajustes deberemos elegir los menús: **Archivo-Preferencias-Ajustes *(File-Preferences-Settings)***

La configuración realmente se graba en un archivo en formato JSON, si son ajustes de usuario se graban en el directorio de configuración de usuario de nuestro sistema y si son de espacio de trabajo se graban en el directorio **.vscode** que se creará en el directorio raíz de nuestro proyecto.

23.5.2 EXTENSIONES

El panel de extensiones de Visual Studio Code permite agregar nuevos elementos al editor para mejorar sus prestaciones. También permite quitar aquellos que no nos interesen. Cuantas más extensiones instalemos, más posibilidades tendrá el editor, pero también, más lento se ejecutará. Las extensiones permiten personalizar el editor a nuestra forma de trabajar.

Hay extensiones que permiten usar las combinaciones de teclas de otros editores con los que estemos más familiarizados, otras permiten integrar nuevos componentes o herramientas, otras manejar nuevos lenguajes, etc. En realidad hay extensiones para mejorar o modificar casi cada aspecto de Visual Studio Code.

Para gestionar las extensiones de nuestro editor deberemos pulsar el botón de extensiones (último botón del panel de botones izquierdo). Aparece un nuevo panel que nos permitirá ver y modificar las extensiones instaladas y también instalar nuevas a través de un panel de búsqueda que nos permite buscarlas por Internet.

ACTIVIDAD 2.3: EJEMPLO DE CONFIGURACIÓN

• Realiza la Práctica 2.3 ***"Configurar Visual Studio Code",*** en la página 41.

2.4 NODE.JSYNPM

2.4.1 ¿QUÉ ES NODE.JS?

Como ya se ha comentado en apartados precedentes, **node.js** es un software que nos permite interpretar código JavaScript. Es muy interesante su instalación porque nos permite poder utilizar JavaScript fuera del navegador ***y*** así programar otro tipo de aplicaciones.

Además, es fundamental para utilizar el gestor de paquetes **npm,** que permite instalar numerosas utilidades en el sistema que nos van a ayudar en nuestro trabajo de desarrolladores.

Node.js permite practicar los fundamentos de JavaScript sin necesidad de crear una página web. La compatibilidad de node.js con los estándares es altísima y eso hace que aprender a trabajar con estructuras de datos y otros elementos avanzados del lenguaje JavaScript, sea más cómoda.

Aunque no es el objetivo de este libro, node.js tiene capacidad para crear código JavaScript back-end, además de capacitar a nuestra máquina para funcionar como un servidor web. Es más, node.js es uno de los servidores web más populares actualmente.

En todo caso, de cara a la pretensión de este libro, que es desarrollar aplicaciones en el lado del cliente, si es interesante node.js es por dos razones:

- Disponer del gestor de paquetes npm para instalar utilidades y herramientas interesantes.
- Poder ejecutar código JavaScript directamente, sin pasar por el navegador, para aprender las bases del lenguaje JavaScript.

2.4.2 INSTALAR NODE.JS

La instalación de Node.js depende del entorno en el que estemos, pero, en general, basta con ir a la página oficial de node.js (https://nodeis.org) y descargar el instalador apropiado (la propia página detecta nuestro sistema).

Una vez descargado el instalador hay que ejecutarlo como administradores y seguir los pasos.

ACTIVIDAD 2.4: INSTALACIÓN DE NODE.JS

• Realiza la Práctica 2.4 *"Instalación de node.js",* en la página 44.

2.4.2.1 EL PROBLEMA DE LAS VERSIONES DE NODE

Como ocurre en mucho otro software (especialmente si es de código abierto), Node.js tiene versiones con diferentes tipos de soporte. Actualmente se distinguen dos versiones principales:

- **LTS.** ***Long Term Support,*** Soporte de Largo Plazo. Se trata de la versión de la que se asegura ya estabilidad y que, además, dispondrá de mucho tiempo de ciclo de vida. Es decir, habrá actualizaciones durante mucho tiempo desde su descarga. Es una versión ya final a la que no se añadirán nuevas características de alto nivel. Es la recomendada, en general, para ser instalada por los desarrolladores.
- **Current.** Versión actual. Incorpora las últimas novedades del lenguaje, pero no se considera estable. Puede seguir renovándose e incorporando nuevas funcionalidades. Suele ser descargada para poder probar y aprender el uso de las nuevas capacidades de JavaScript que se han añadido a Node.js.

Como hemos comentado, la versión LTS es la que se debe descargar.

Ahora bien, cada cierto tiempo aparecen nuevas versiones. Node.js pasa del estado ***Current*** al estado ***Active*** (Activo) cada versión en un ciclo de desarrollo predecible y que se puede consultar en https://nodejs.org/es/about/releases/

Cuando una versión pasa al estado ***Mantenance*** (Mantenimiento) significa que ya no se renueva y que se da soporte solo cuando se detecten problemas y ***bugs*** en esa versión.

Suele ser un problema el actualizar node.js, porque implica desinstalar la versión anterior e instalar la nueva.



Versión	Estado	Nombre Clave	Lanzamiento Inicial	Inicio LTS Activo	Inicio Mantenimiento LTS	Fin-de-vida
v8	Mantenimiento LTS	Carbón	2017-05-30	2017-10-31	2019-01-01	2019-12-31
vIO	LTS Activo	Dubnium	2018-04-24	2018-10-30	2020-04-01	2021-04-01
vl2	Actual		2019-04-23	2019-10-22	2021-04-01	2022-04-01
vl3	Pendiente		2019-10-22		2020-04-20	2020-06-01
Vl4	Pendiente		2020-04-21	2020-10-20	2022-04-01	2023-04-01

Figura 2.6: Calendario de versiones de node.js indicando la previsión de versiones activas, en mantenimiento e inestables

2.4.3 NVM

Existe una utilidad que permite administrar de forma cómoda la gestión de las versiones de node.js. Se trata de nvm.

Cuando esa utilidad está instalada, disponemos del comando **nvm** en el terminal del sistema (independientemente de qué terminal estemos usando). Con este comando podremos instalar varias versiones de Node.js (para hacer pruebas o para probar la compatibilidad de nuestras aplicaciones) a la vez.

La instalación de **nvm** en un sistema Linux o MacOS es muy sencilla, basta con usar nuestro gestor de paquetes habitual e instalar el paquete nvm de la misma forma que se instala cualquier otro paquete.

En el caso de Windows, la instalación es un poco más compleja. Deberemos acudir al repositorio disponible en https://github.com/coreybutler/nvm-windows donde se nos informa de la instalación en Windows. Además en https://github.com/coreybutler/nvm-windows/releases podemos descargar el instalador de nvm que deseemos.

Realmente hay matices en el funcionamiento de nvm para Windows y el nvm normal (Linux/Mac), pero, en general, funciona muy parecido.

Debemos desinstalar cualquier versión de node.js para poder aprovechar toda la potencia de nvm a la hora de gestionar las diferentes versiones de node.

Los comandos más interesantes de nvm son:

- **nvm install *versión.*** Ínstala la versión de node.js que indiquemos. Si la versión la cambiamos por la palabra **latest,** se instala la última estable.
- **nvm ls.** Muestra las versiones de node.js instaladas. Coloca un asterisco en la versión que se utiliza actualmente como versión por defecto.
- **nvm ls available.** Muestra las versiones de node.js disponibles para instalar. En ambiente Linux este comando es **nvm ls-remote.**
- **nvm use *versión.*** Hace que la versión indicada sea la que funcione como versión por defecto.
- **nvm uninstall *versión.*** Desinstala la versión de node.js indicada.
- **nvm root.** Muestra la ruta del sistema en la que se almacenan físicamente las diferentes versiones de node.js.
- **nvm -v.** Muestra la versión instalada del propio nvm.

ACTIVIDAD 2.5: INSTALACIÓN DE NVM

• Realiza la Práctica 2.5 *"Instalación de nvm",* en la página 46.

2.4.4 NPM

Es fácil confundir npm con nvm. Ambos son comandos en el sistema relacionados con node.js. Mientras que nvm es una utilidad para gestionar diferentes versiones de node.js en el mismo sistema, npm es una utilidad de instalación de paquetes software.

El administrador de paquetes npm permite instalar todo tipo de utilidades que facilitan mucho el trabajo del desarrollador front-end (también el de otro tipo de desarrolladores). Los paquetes que instalemos los podemos poner a disposición de todo el sistema si usamos el modificar -g. De otro modo, el paquete se instala para el directorio en el que estemos. Este segundo método se usa cuando queremos instalar paquetes específicamente para un proyecto.

En nuestro caso la mayoría de paquetes que necesitemos se instalarán con el modificador -g, porque queremos que sean utilidades disponibles en cualquier parte del sistema.

Así, algunos comandos a conocer de npm son (se usa en todos ellos el modificador -g para referirnos a modificadores globales):

- **npm ls -g.** Muestra los paquetes globales instalados.
- **npm install *paquete* -g.** Ínstala un paquete en el sistema. Se puede indicar más de un paquete si los separamos con espacios.
- **npm uninstall *paquete* -g.** Desinstala el paquete indicado.
- **npm root.** Muestra la ruta del sistema en la que se almacenan los paquetes descargados,
- **npm search *texto.*** Busca en los repositorios de npm en Internet nombres de paquetes que contengan el texto indicado.
- **npm -v.** Muestra la versión instalada de npm.

ACTIVIDAD 2.6: INSTALACIÓN DE PAQUETES GLOBALES CON NPM

• Realiza la Práctica 2.6 *"Instalación de paquetes globales con npm",* en la página 50.

2.5 GIT

2.5.1 SISTEMAS DE CONTROL DE VERSIONES Y GIT

Hay una situación temida por los programadores. Se trata de disponer de un software que funciona correctamente, pero al que tenemos que añadir una funcionalidad o modificar parte de su comportamiento. Cuando intentamos hacer las modificaciones resulta que nuestro programa ya no funciona como antes y tenemos problemas para detectar cuál es el problema. Ante esta situación lo ideal sería volver a la situación anterior.

Este es uno de los ejemplos en los que un sistema de control de versiones nos proporciona su utilidad. Otro típico, y muy habitual, ocurre en el caso de que varios programadores estén colaborando para crear un determinado proyecto y trabajan en paralelo *y* desean mezclar las modificaciones de ambos para que el proyecto refleje ambos trabajos.

En definitiva, un sistema de control de versiones nos permite poder hacer instantáneas de la situación de un proyecto en un momento dado, para regresar a dicha instantánea cuando lo consideremos necesario.

La cuestión del trabajo en equipo se soluciona mediante ramas en las que el desarrollo del proyecto sigue distintos caminos (cada uno con sus versiones o instantáneas) que podremos mezclar a voluntad.

Hay varios posibles sistemas de control de versiones que podemos utilizar. Actualmente el más exitoso es **Git.**

Git fue creado por **Linus Torvals** (conocido por ser la persona que desarrolló Linux) en el año 2005. Git tiene numerosas posibilidades, pero en este libro nos centraremos solo en la capacidad de poder grabar versiones de nuestro código a las que podremos acudir cuando queramos.

2.5.2 INSTALACIÓN DE GIT

Git se puede descargar de https://git-scm.com/downloads donde dispondremos de instaladores para todas las versiones.

Una vez instalado dispondremos del comando git en nuestro terminal del sistema. Lo primero que deberemos hacer es indicar nuestro nombre con el que se grabará nuestro código, que dará autoría al código que gestionemos con git:

```
> git coriñg —global user.ñame "Jorge Sánchez"
```

También podemos indicar nuestro email:

```
> git config —global user.email "jorgesanchez@jorgesanchez.net"
```

Estos son parámetros globales que se asociarán a todos nuestros proyectos.

ACTIVIDAD 2.7: INSTALAR GIT

- Realiza la Práctica 2.7 *"Instalación de Git",* en la página 51.

2.5.3 FUNCIONAMIENTO BÁSICO DE GIT

Explicar en detalle el funcionamiento de Git daría para un libro completo. No obstante, explicaremos las bases fundamentales que nos permitirán trabajar con Git.

Normalmente, un proyecto de trabajo se basa en una carpeta o directorio raíz que contendrá todos los archivos del proyecto. Para que dispongamos de control de versiones basado en Git sobre esa carpeta, deberemos acudir al terminal del sistema y situarnos en dicha carpeta. Ahí escribiremos:

```
> git init
```

Ese comando creará un directorio oculto llamado .git en el que se almacenarán los archivos que el sistema git necesita para controlar las versiones de nuestro proyecto. El comando solo se necesitará escribir una vez.

A partir de ese momento nuestro proyecto puede tener tres estados:

- ***Working Directory.*** Estado en el que hemos hecho modificaciones a los archivos, pero no las hemos marcado de ninguna manera para ser tenidos en cuenta.
- ***Staging.*** Estado de puesta en escena. Se han hecho cambios y se han marcado los que queremos que se tengan en cuenta para la siguiente confirmación. El comando que permite pasar a este estado es **git add,** al que hay que indicar qué archivos se tendrán en cuenta. Si estamos en la raíz de nuestro proyecto y escribimos:

```
> git add .
```

Entonces todos los cambios realizados en cualquier archivo se tendrán en cuenta

- ***Repositorio.*** Con él, los cambios marcados ya están grabados en una instantánea. El comando que hace posible este estado es **git commit:**

```
> git commit -m "Realizada área de login"
```

Tras ese comando se graba una versión de nuestro código a la cual se asigna una clave única e irrepetible. Esa clave es la que nos permitirá acudir a la versión cuando deseemos.

El estado actual del proyecto lo podemos obtener mediante el comando:

```
> git status
```

El listado de las versiones que hemos guardado está disponible con este comando:

```
> git log
```

Volver a una versión concreta de un archivo se puede hacer con el comando

```
> git checkout 43e6f index.html
```

Este comando devuelve el archivo index.html al estado que tenía tras la versión asociado al número que comienza con el hexadecimal 43e6f. Para que se mantengan esos cambios, basta con ejecutar un nuevo ***commit.***

Sin indicar archivo alguno, entonces devolveremos nuestro repositorio a ese estado. Sería necesario utilizar ramas para separar esa nueva versión del proyecto. Pero el uso de ramas no será explicado en este libro para no extendernos demasiado.

Sí conviene conocer un comando que devuelve nuestro proyecto a un estado anterior, pero eliminando todos los ***commits*** a partir de ese momento. Lógicamente hay que estar muy seguros de que es eso lo que queremos hacer. El comando es:

```
> git reset -hard 43e6f
```

ACTIVIDAD 2.8: CREAR VERSIONES CON GIT

- Realiza la Práctica 2.8 *"Uso de Git para crear versiones",* en la página 54.

2.5.4 GITHUB. GIT EN REMOTO

Los sistemas de control de versiones se hacen todavía más interesantes cuando se añade la posibilidad de almacenar el código en Internet. En este sentido hay que hablar de GitHub. Se trata de un servicio que permite almacenar repositorios Git en la nube. GitHub contiene miles de proyectos (la mayoría de código abierto) pensados para que se desarrollen de forma colaborativa. Almacenar nuestros propios proyectos en GitHub es gratuito y solo requiere darnos de alta.

Una vez que nos demos de alta bastará con crear un repositorio en GitHub con el nombre que deseemos. Inicialmente en las cuentas gratuitas solo podíamos crear repositorios públicos, pero

desde el año 2019 es posible crear de forma gratuita repositorios privados (con, como mucho, tres colaboradores).

Hay otros servicios similares a GitHub, como **Bitbucket** o **GitLab,** pero indudablemente GitHub (que actualmente pertenece a Microsoft) es el más popular. En todo caso el uso de repositorios en la nube requiere de estos comandos:

git clone *URL.* Permite descargar un repositorio en la nube en el directorio en el que estemos situados. Se descargarán también todas las versiones del mismo. Si el repositorio es privado se nos requerirá introducir nuestro usuario y contraseña.

git remote. Permite mostrar los servicios remotos asociados a nuestro proyecto actual,

git remote *alias URL.* Establece un servicio remoto asociado a nuestro proyecto. La dirección del repositorio remoto es la indicada con su URL. El ***alias*** es el nombre familiar que le damos al repositorio remoto.

git remote rm *alias.* Quita la asociación con el repositorio remoto indicado. No borra realmente el repositorio, lo que hace es dejar de asociar ese repositorio remoto a nuestro proyecto.

git push *alias rama.* Publica el repositorio actual en el remoto indicado por su alias. Índica también la rama que subimos, si no hemos usado ramas, la rama habitual se llama **master.**

git pulí *alias rama.* Descarga el contenido del repositorio remoto y lo mezcla con el repositorio actual.

ACTIVIDAD 2.9: SUBIR NUESTRO PROYECTO A GITHUB

- Realiza la Práctica 2.9 *"Subir nuestro código a GitHub",* en la página 58.

2.6 PRÁCTICAS RESUELTAS

Práctica 2.1: Instalación de Visual Studio Code

- Realiza la instalación de Visual Studio Code en Windows. Puedes realizar la instalación también en otros sistemas ya que los pasos son similares

SOLUCIÓN: PRÁCTICA 2.1

[1] Ir a la dirección: https://code.visualstudio.com/ desde esa página podemos descargar el instalar del programa apropiado para nuestro sistema. El botón de descarga que aparece nos permite descargar el instalador para el sistema que ha detectado la propia página, pero podemos elegir otro instalador usando el botón con una flecha hacia abajo.

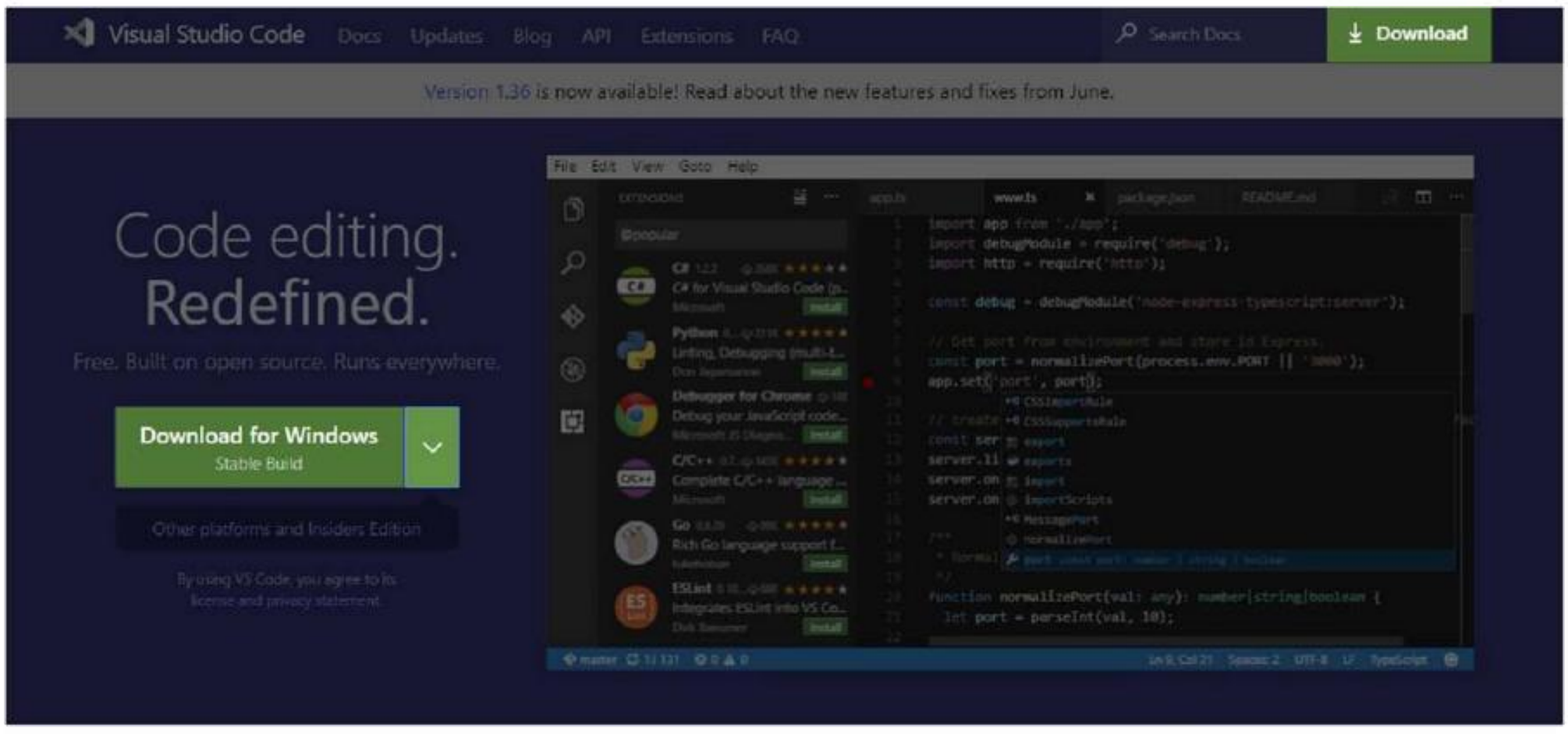


Figura 2.7: Página de descarga de **Visual Studio Code** con los botones de descarga resaltados

[2] Tras hacer clic en el botón de descarga, dispondremos del instalador. Su nombre será **VSCodeSetup** seguido del nombre del sistema (por ejemplo **x64** para Windows de 64 bits) y de la versión de Visual Studio Code. Hacemos doble clic en el archivo para lanzar la instalación:

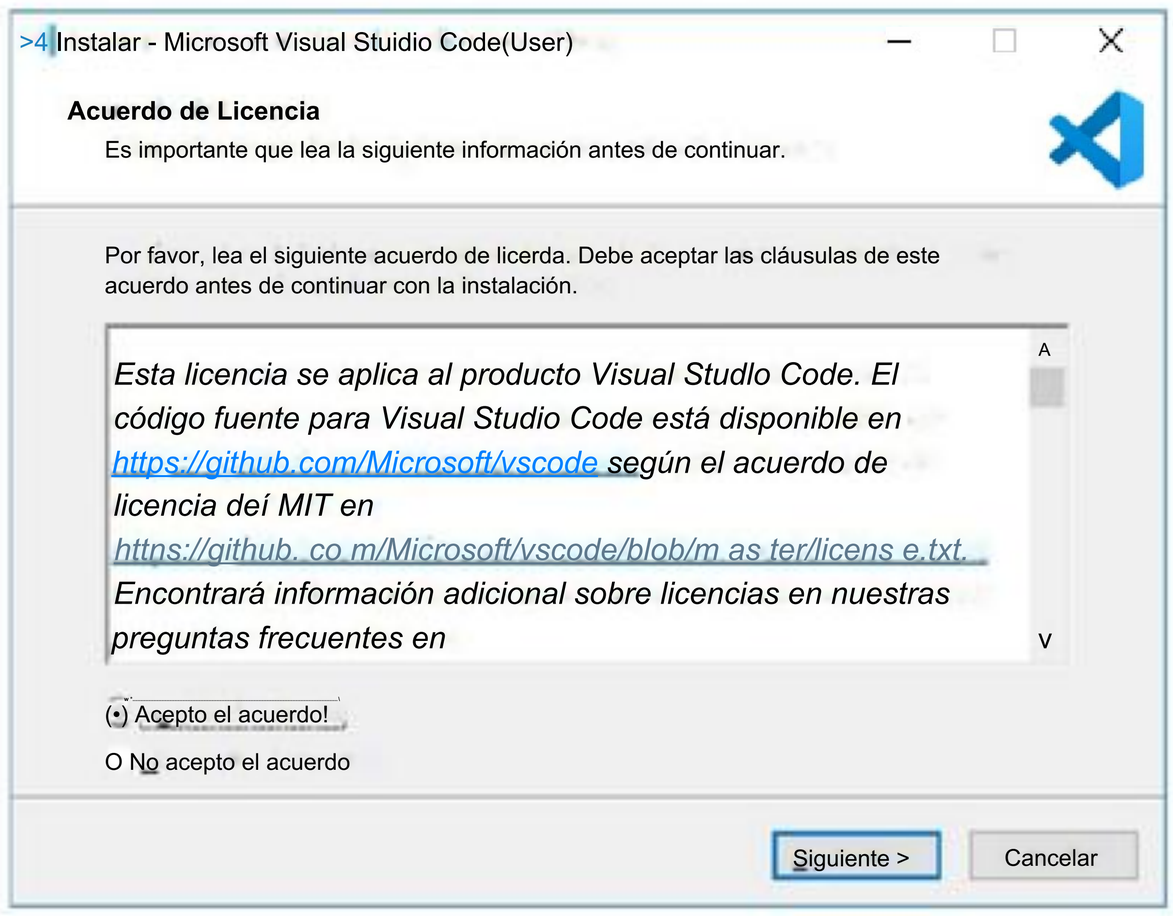


Figura 2.8: Pantalla de acuerdo de licencia del instalador de Visual Studio Code

[3] Tras leer la licencia y aceptarla, si estamos de acuerdo, continua el proceso de instalación. Ahora seleccionaremos el directorio de instalación. Por defecto instala en el directorio del usuario actual en el disco en el que esté instalado el sistema.

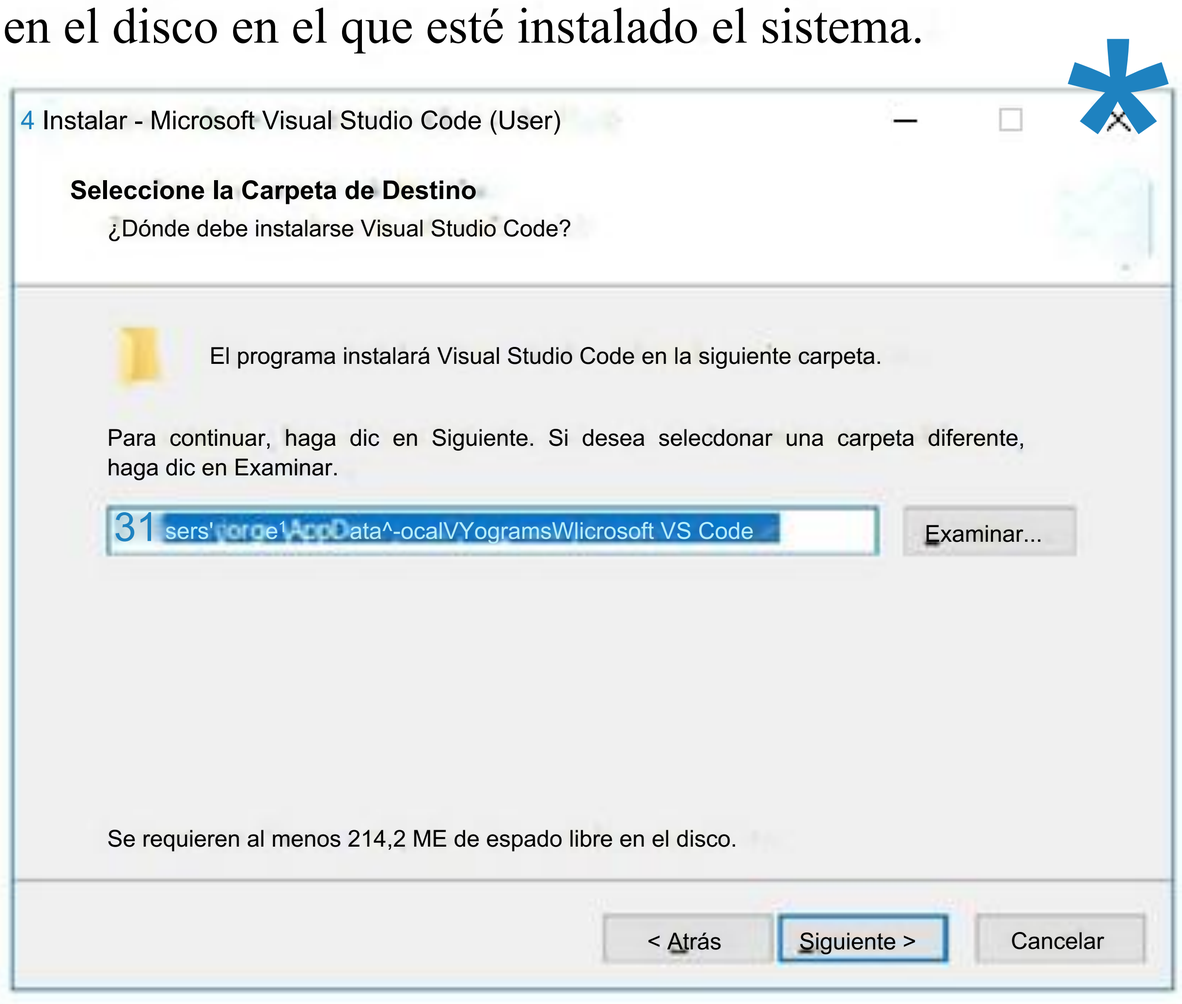


Figura 2.9: Pantalla de acuerdo de licencia del instalador de Visual Studio Code

[4] En el siguiente paso nos pide el nombre del grupo con el que el software aparece en el menú de inicio de Windows

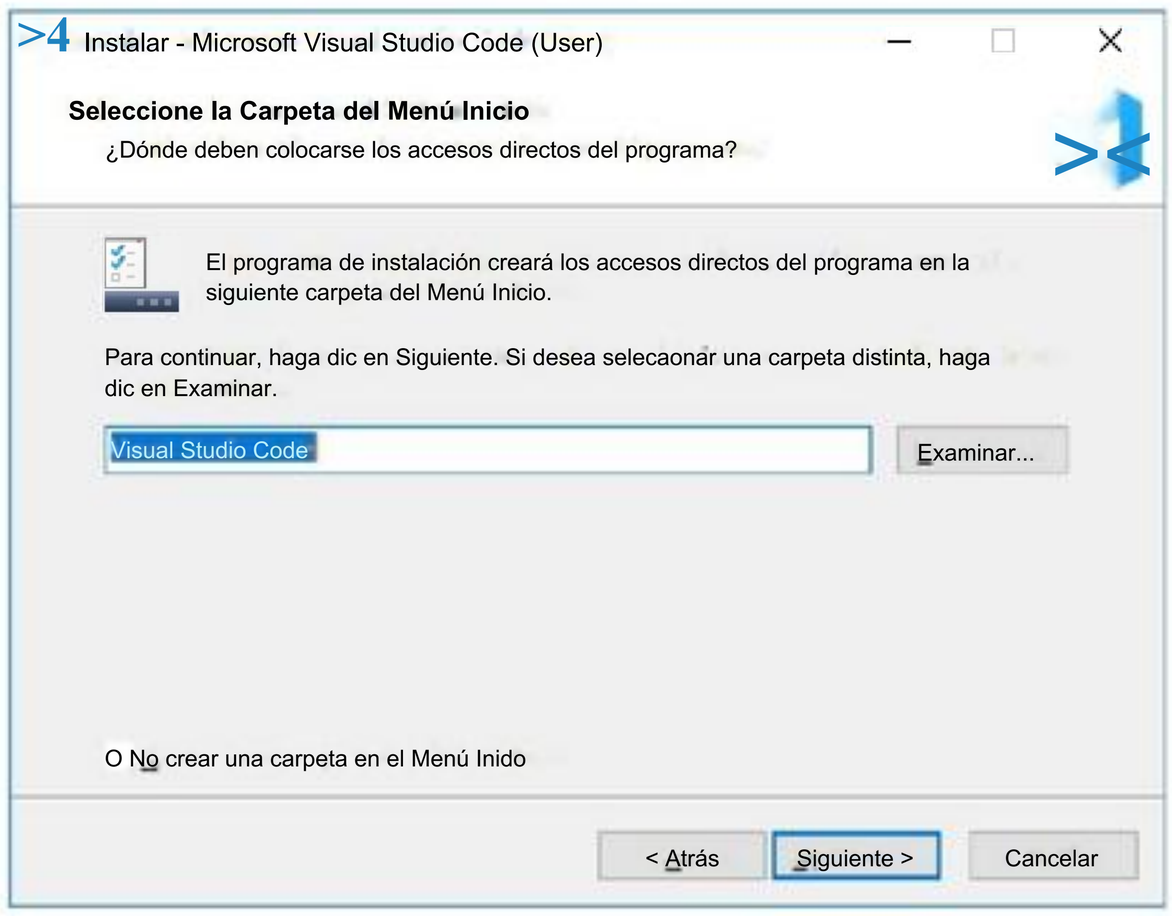


Figura 2.10: Nombre del grupo de programas en el menú de inicio

[5] El siguiente paso tiene opciones importantes. Son varias casillas que nos permiten indicar aspectos de cómo deseamos trabajar con Visual Studio Code:

- La casilla de ***Crear un icono en el escritorio*** permite acceder al programa desde el propio escritorio. Puede ser buena idea activarla.
- La casilla con el texto: ***Agregar la acción Abrir con Code en el menú contextual de archivo...*** es muy interesante, ya que permite, cuando estemos recorriendo carpetas desde el explorador de Windows, hacer clic encima de un archivo con el botón secundario del ratón y usar la opción ***Abrir con Code*** que nos permitirá manipular y examinar el archivo usando Visual Studio Code.
- La casilla siguiente: ***Agregar la acción Abrir con Code en el menú contextual de directorio....*** es muy parecida. En este caso, permite abrir, con la misma opción de ***Abrir con Code,*** el contenido del directorio sobre el que hemos pulsado el botón secundario para que sea el directorio de trabajo en Visual Studio Code.
- La casilla: ***Registrar Code como editor de tipos de archivo admitidos*** hace que Visual Studio Code se registre como editor predeterminado de todos los tipos de archivo que este software sea capaz de abrir. Son más interesantes, desde nuestro punto de vista, las opciones anteriores, pero hay muchos usuarios que activan esta casilla para no tener siquiera que hacer uso del botón secundario. Si no nos parece del todo recomendable es porque abre tipos de archivo donde este editor quizá no sea la opción predeterminada más idónea.
- La última casilla: ***Agregar a PATH*** requiere reinicio. Eso permite que la ruta a Visual Studio Code esté disponible en el PATH del sistema. Gracias a ello, desde el terminal del sistema, el comando code lanzará el editor. Es muy recomendable activar esta casilla.

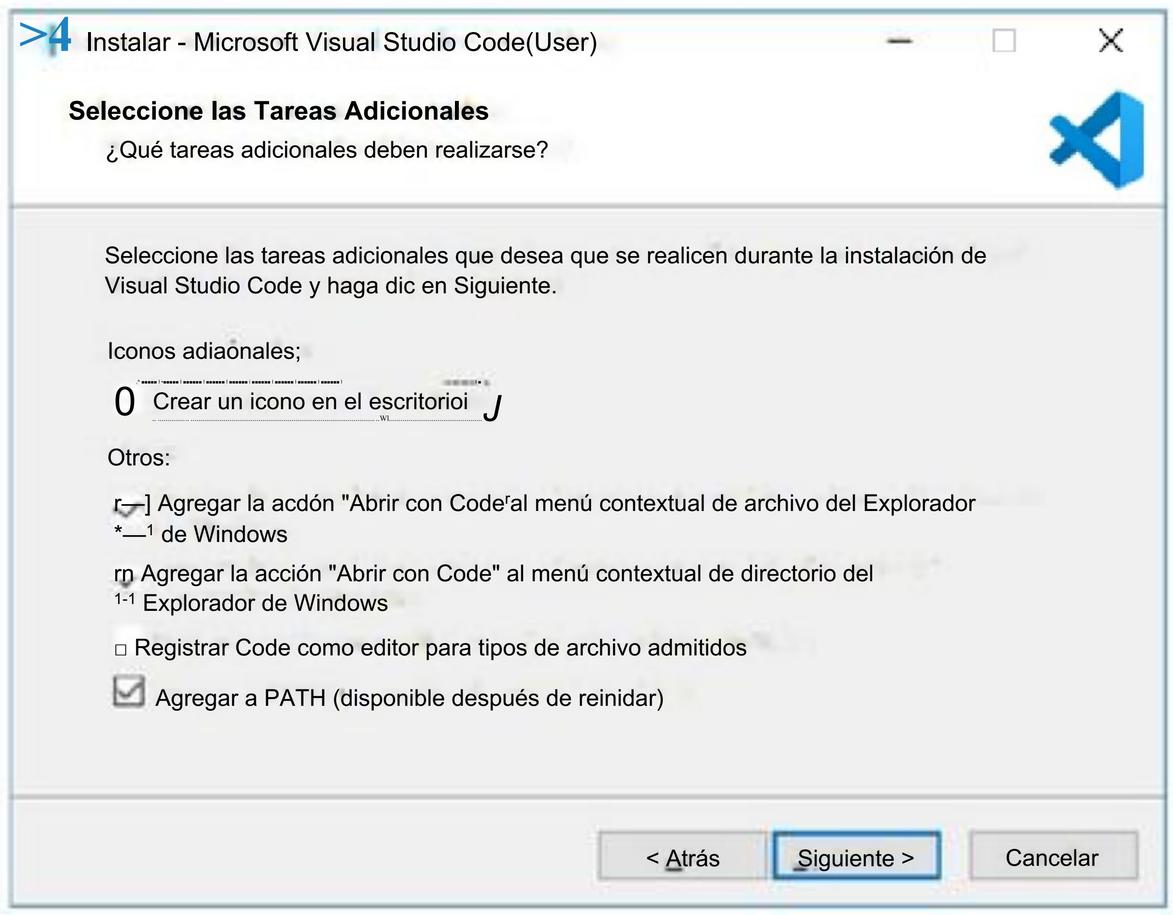


Figura 2.11: Opciones de instalación recomendadas para Visual Studio Code

[6] Tras examinar, en la pantalla siguiente, el resumen de la instalación elegida, podremos pulsar el botón ***Instalar.*** La instalación es rápida e inmediatamente podremos ya probar el programa:



Figura 2.12: Finalización de la instalación

Práctica 2.2: Iniciándose con Visual Studio Code

- Crea un nuevo proyecto con Visual Studio Code
- Haz que simplemente el proyecto sea una página web que tenga un título que diga "Mi primera página con Visual Studio Code"
- Haz que se use un archivo externo con CSS que modifique la letra para sea sans-serif y que el texto Visual Studio Code sea de color violeta

SOLUCIÓN: PRÁCTICA 2.2

[1] Desde el sistema operativo crearemos una carpeta o directorio que se llame **Práctica02-02** en la ubicación que deseemos. Podemos usar otro nombre si lo deseamos.

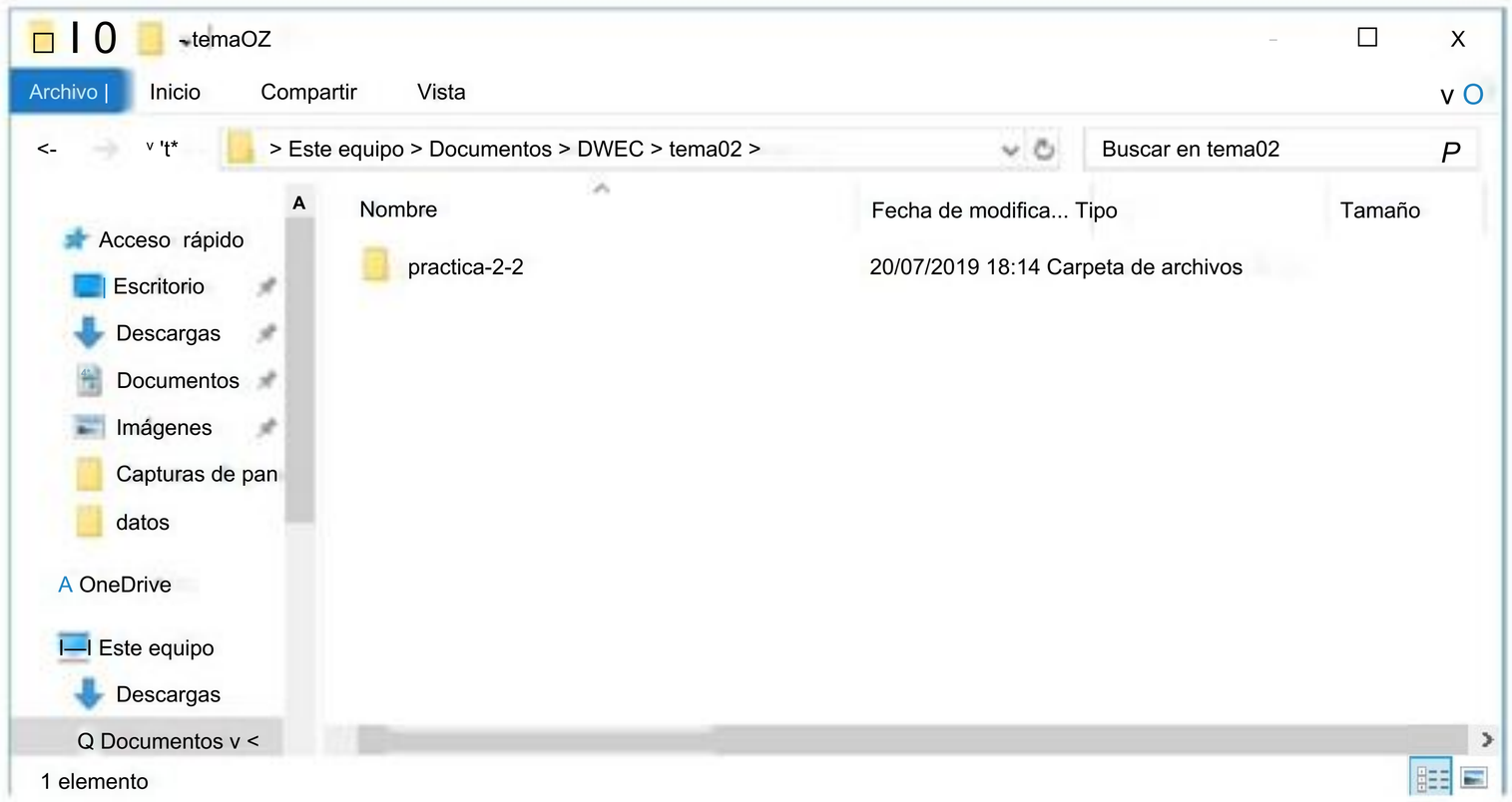


Figura 2.13: Ejemplo de ubicación para la nueva carpeta en Windows

[2] Abrimos esa carpeta desde Visual Studio Code. Podemos usar **File-Open Folder** o desde el menú contextual en el explorador del sistema eligiendo **Abrir con Code.** Para ello en la instalación, tenemos que haber permitido esa posibilidad. Véase Figura 2.14.

[3] Cerramos el archivo de bienvenida haciendo clic en la equis (X) de la pestaña con el texto **We/come.**

[4] Creamos un nuevo archivo mediante el menú **File-New File** o pulsando la combinación de teclas **Ctrl+N.**

[5] Guardamos ya el archivo para indicar su nombre y extensión y que así, Visual Studio Code, sepa qué lenguaje vamos a utilizar. Para ello elegimos **File-Save** o bien pulsamos **Ctrl+S.** Después escribimos el nombre del archivo, será **index.html** (es importante escribir la extensión **.html)**

[6] Visual Studio Code tiene instalada la extensión **Emmet** por defecto, gracias a ella nos será más fácil escribir el código. Simplemente escribiremos el cierre de admiración (!) y pulsaremos tabulador. Se escribirá el código base de la página.

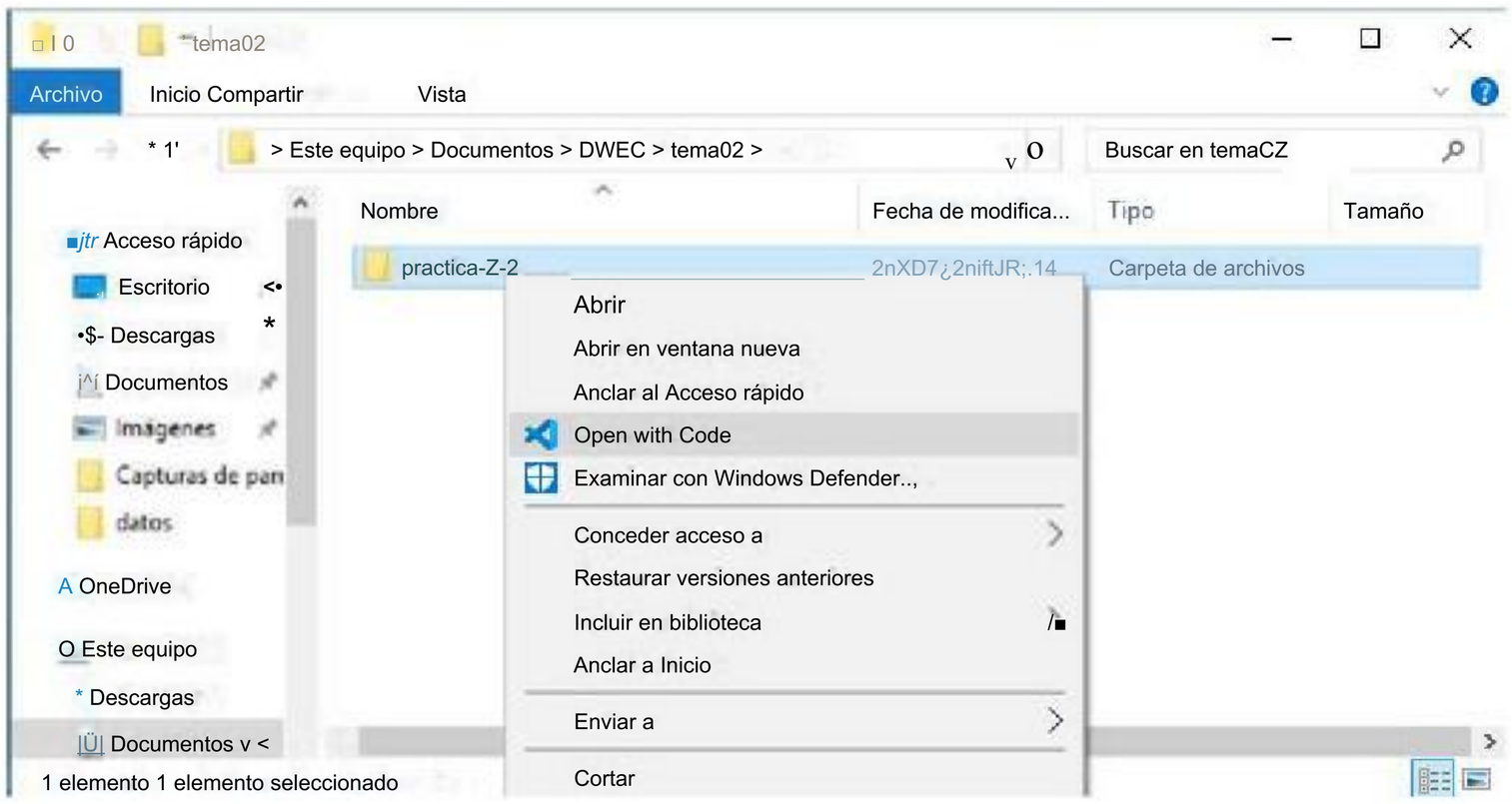


Figura 2.14: Usar la carpeta como raíz del proyecto usando la opción *Abrir con code*

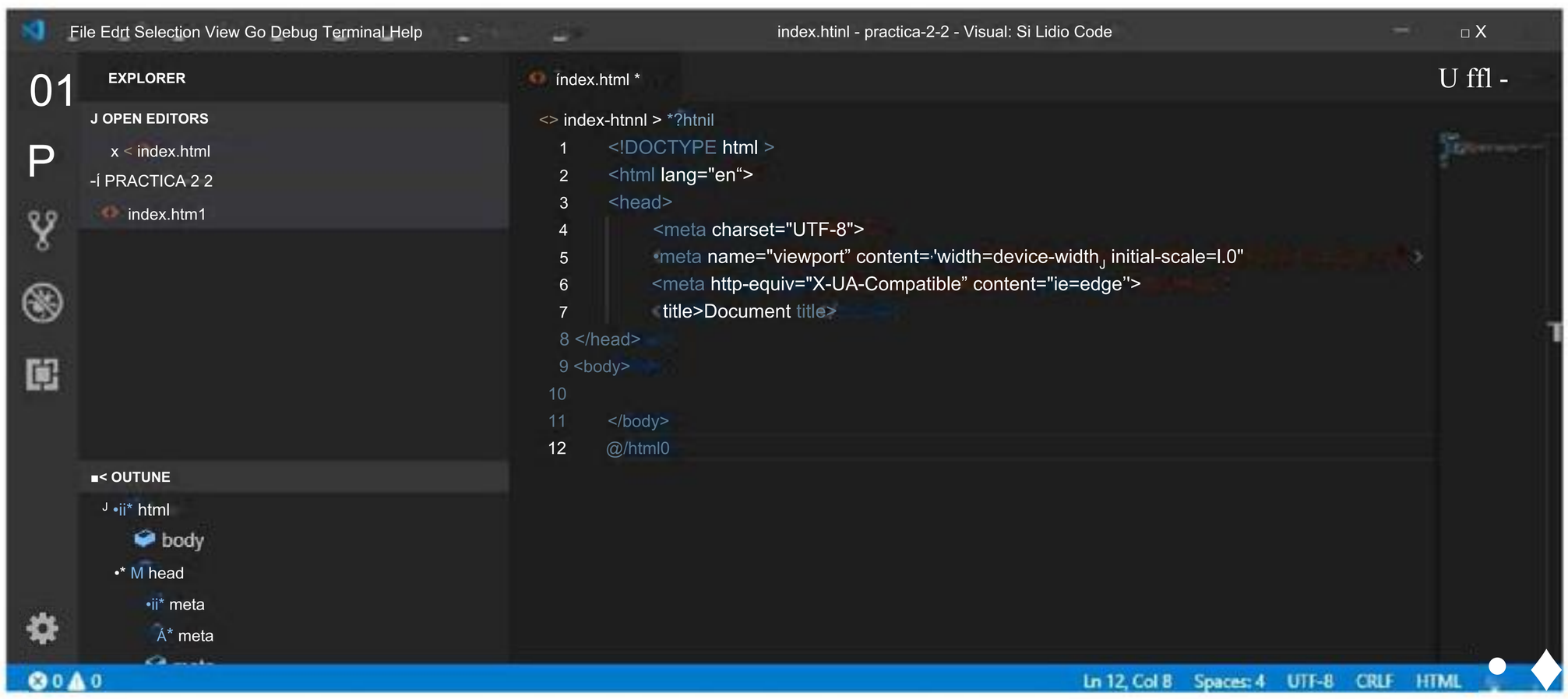


Figura 2.15: Resultado en la ventana de Visual Studio Code de las acciones hasta el punto 6

[7] Modificamos el código base para que quede de esta forma:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Mi primera página</title>
</head>
<body>
        <hl>Mi primera página con <strong>Visual Studio Code</strong>
        </hl>
</body>
</html>
```

[8] Creamos una carpeta dentro de la raíz del proyecto. Para ello arrimamos el ratón al nombre del proyecto en el panel explorador ***(Explorer)*** dentro de Visual Studio Code ***y*** hacemos clic en el botón de nueva carpeta ***(New Folder).***



Figura 2.16: Botones del panel Explorador que aparecen al arrimar el ratón sobre la carpeta de proyecto. Se ha resaltado el botón para crear una nueva carpeta

[9] Escribimos **css** como nombre de la nueva carpeta.

[10] Creamos un nuevo archivo en esa carpeta. Podemos simplemente pulsar el botón secundario del ratón en la carpeta css dentro del Explorador en Visual Studio Code y elegir **New File.** Escribiremos como nombre del archivo: **estilos.css.**

[11] Añadiremos el siguiente código:

```
hl{
        font-family: sans-serif
}

strong{
        color:blue;
}
```

[12] Guardamos el archivo pulsando la combinación de teclas **Ctrl+S.**

[13] Volvemos al archivo **index.html** (bastará con hacer clic en su pestaña en la parte superior del editor). Añadiremos, dentro de la sección **head** la etiqueta **link** que nos permite enlazar el archivo CSS anterior.

[14] Guardamos los cambios, desde el sistema operativo podremos abrir el archivo index.html en el navegador para ver el resultado.

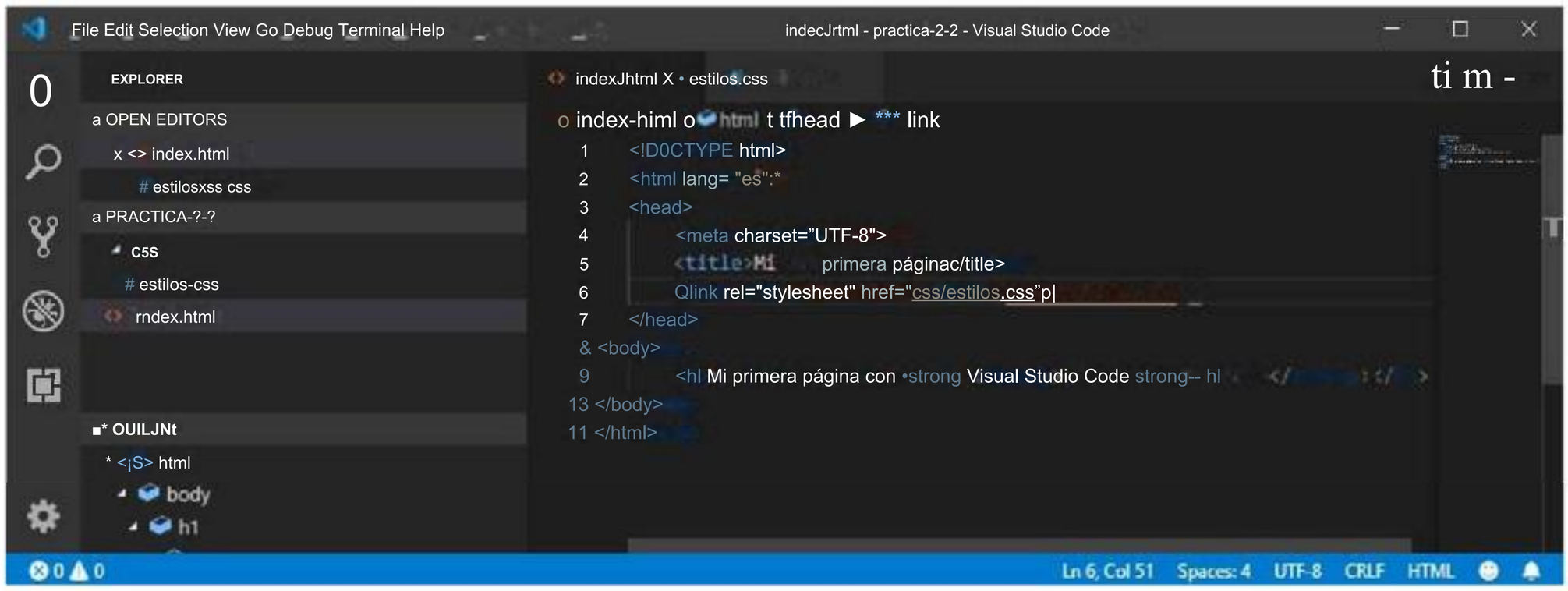


Figura 2.17: Aspecto de Visual Studio Code al finalizar la práctica

Práctica 2.3: Configurar Visual Studio Code

• Realiza estas configuraciones en Visual Studio Code:

- Conseguir que el lenguaje de los menús sea el español
- Modificar el tipo y tamaño de letra
- Cambiar el tema de colores
- Añade las extensiones: **Auto Rename Tag, Open in Browser** y **Live Server**

SOLUCIÓN: PRÁCTICA 2.3

[1] Comenzamos añadiendo la extensión para el idioma español. Hay que tener en cuenta que, aunque añadamos esa extensión, no todo el editor modifica el idioma en todos sus elementos. Por ello hay muchos desarrolladores que prefieren no modificar el idioma. Para ello hacemos clic en el botón de **Extensiones** en el que podemos manipular las extensiones de Visual Studio Code.

[2] En el cuadro de texto de búsqueda de extensiones escribimos **Spanish.** Se buscarán paquetes con ese término. Aparecerá el paquete a instalar: **Spanish Language Pack for Visual Studio Code.** Instalamos el paquete haciendo clic en **Install.**

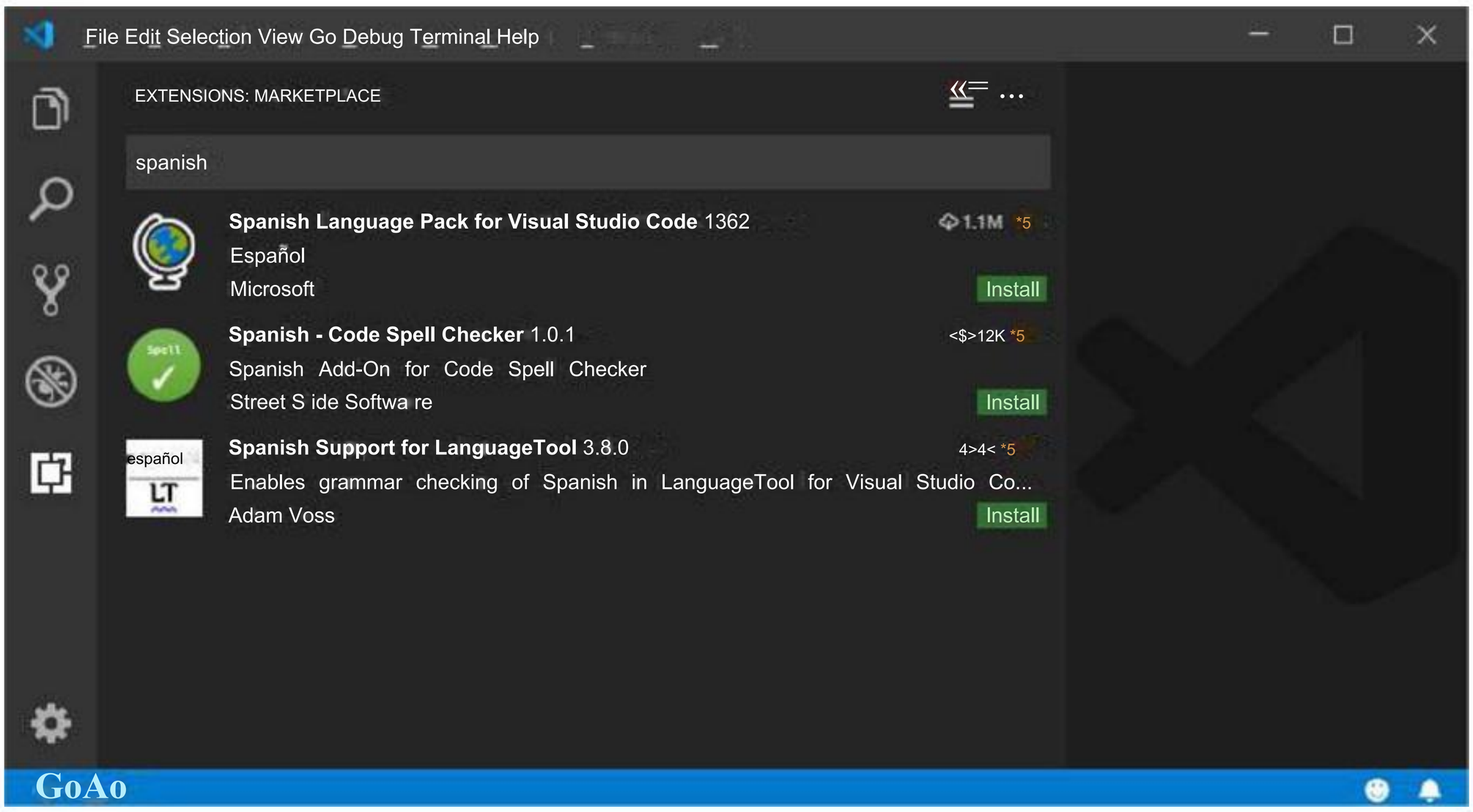


Figura 2.18: Panel de extensiones con el paquete de instalación del lenguaje español en primer lugar

[3] Tendremos que reiniciar Visual Studio Code para que la extensión cambie el idioma a español.

[4] Para cambiar el tipo de letra y tamaño vamos a la configuración desde el menú **Archivo-Preferencias-Configuración** o desde el botón de la rueda dentada que está en la parte inferior izquierda o pulsando las teclas Ctrl+coma (,).

[5] Vamos al apartado Editor de texto-Fuente y modificamos el tamaño (18) y el tipo de letra a **Sourcer Code Pro,** dando opciones a **Courier New** y a la genérica **monospace.** La letra Sopurcer Code Pro, se puede descargar libremente y gratuitamente de Internet. En cuanto editamos la configuración, ya se llevan a cabo los cambios.

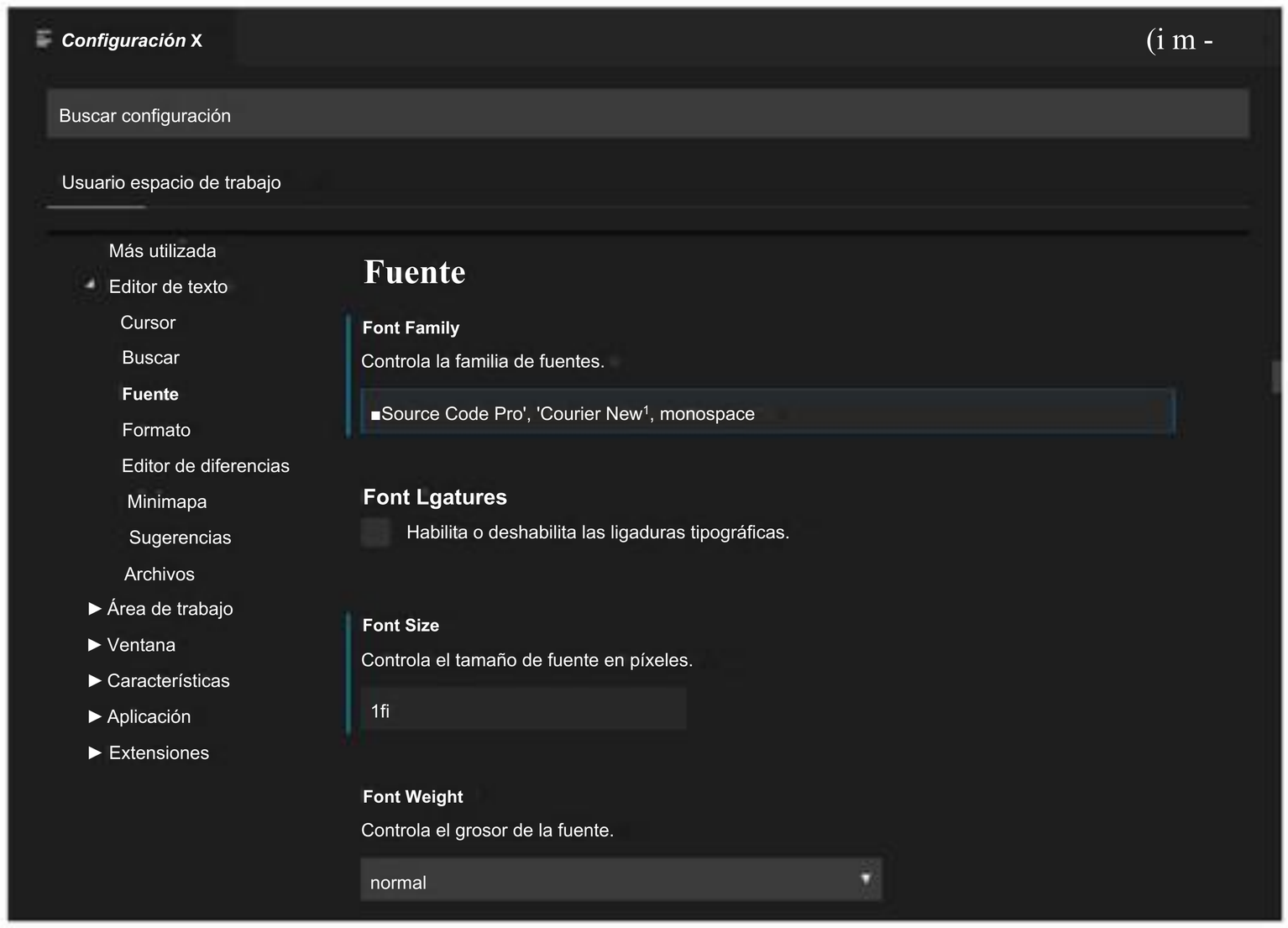


Figura 2.19: Configuración con los cambios en la fuente del sistema

[6] Para cambiar el tema de colores, bastaría con ir a **Archivo-Preferencias-Tema de color.** Aparecerá un panel en el que podemos elegir el deseado. También podemos instalar nuevos temas. Cabe decir que podemos instalar y cambiar también temas de iconos, los cuales modifican los iconos para las miniaturas de los archivos con las que aparecen en el panel de navegación. Véase Figura 2.20.

[7] Para añadir las extensiones, basta con añadir su nombre en el panel de extensiones. Todas son útiles.

[7.1] **Auto Rename Tag.** Cuando modificamos el nombre de una etiqueta HTML (o XML), se modifica también la etiqueta de cierre.

[7.2] **Open in Browser.** Una vez instalada, desde el panel de navegación podemos hacer clic con el botón secundario en cualquier archivo y aparecerán las opciones ***Open in default browser*** (abre el archivo en el navegador predeterminado) y ***Open in other browsers*** (permite elegir el navegador con el que se abrirá el archivo).



Figura 2.20: Panel de cambio del tema de color

[7.3] **Live Server.** Ejemplo de extensión muy potente. Tras la instalación, desde el botón secundario de ratón sobre un archivo tendremos la opción **Open with Live Server.** Veremos cómo se abre nuestro navegador por defecto mostrando el archivo, pero usando una ruta (normalmente: http://127.0.0.1:5500/) que nos indica que se ha creado un servidor local. Lo interesante es que cada vez que hagamos cambios en el archivo y los guardemos, veremos al instante el resultado en el navegador. Incluso podemos copiar esa ruta a otro navegador y veremos cómo también funciona

[8] Podemos ver las extensiones instaladas en el panel de extensiones, apartado ***Habilitado.***

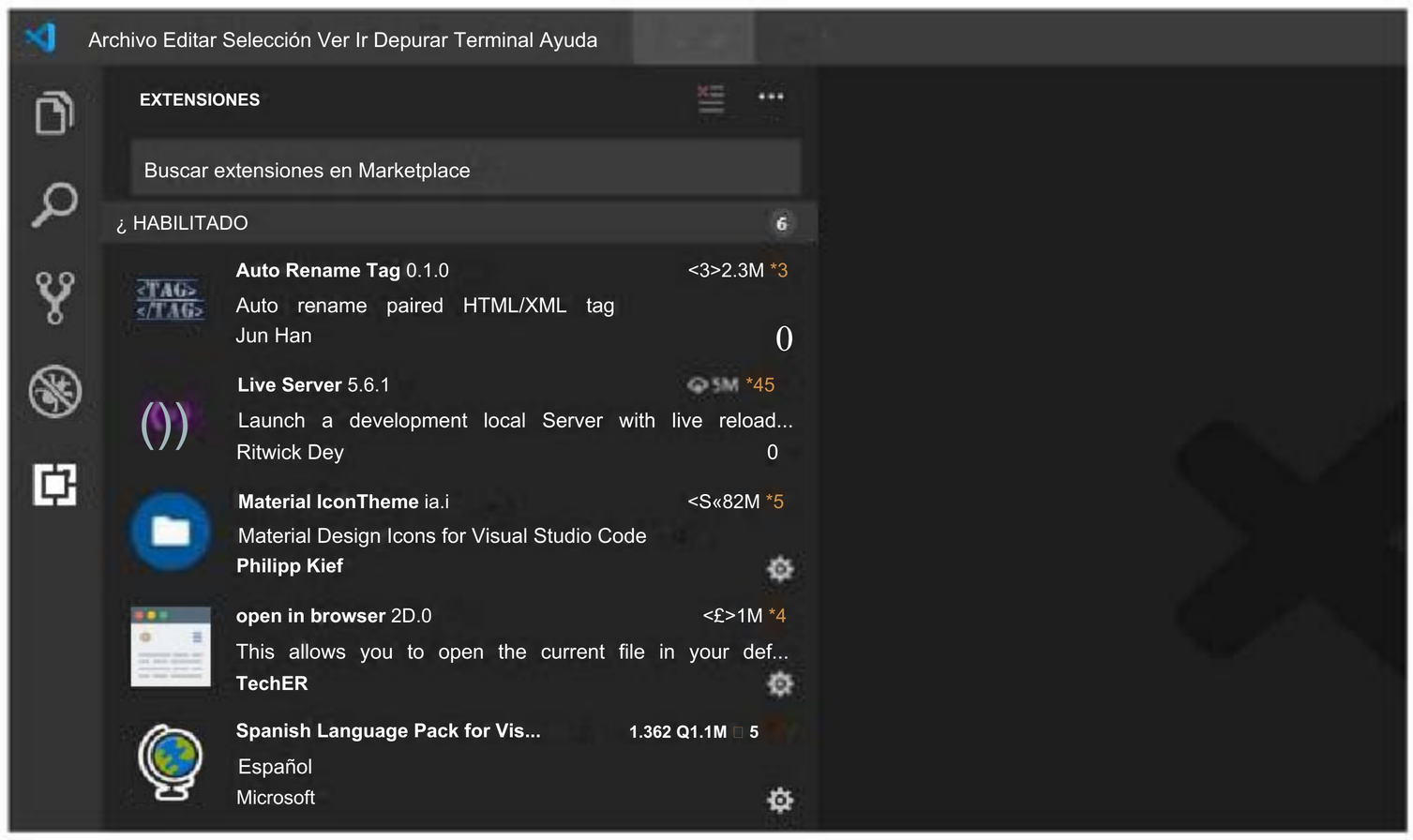


Figura 2.21: Ejemplo de extensiones instaladas en Visual Studio Code

Práctica 2.4: Instalación de node.js

- Instala node.js y comprueba que está funcionando

SOLUCIÓN: PRÁCTICA 2.4

[1] Se comenta la solución para entorno Windows. Ir a la dirección https://nodeis.org/es/

[2] Haz clic en el botón con la versión LTS que se sugiere para la descarga (en la imagen inferior sería el botón con el texto ***10.16.0 LTS).***



Figura 2.22: Página de descarga de node.js

[3] Tras la descarga, ejecutamos el archivo descargado. Se inicia la instalación.

[4] Avanzamos, pulsando en ***Next,*** la primera pantalla y en la segunda aceptamos la licencia. Se nos pregunta en el tercer paso la ruta de instalación:

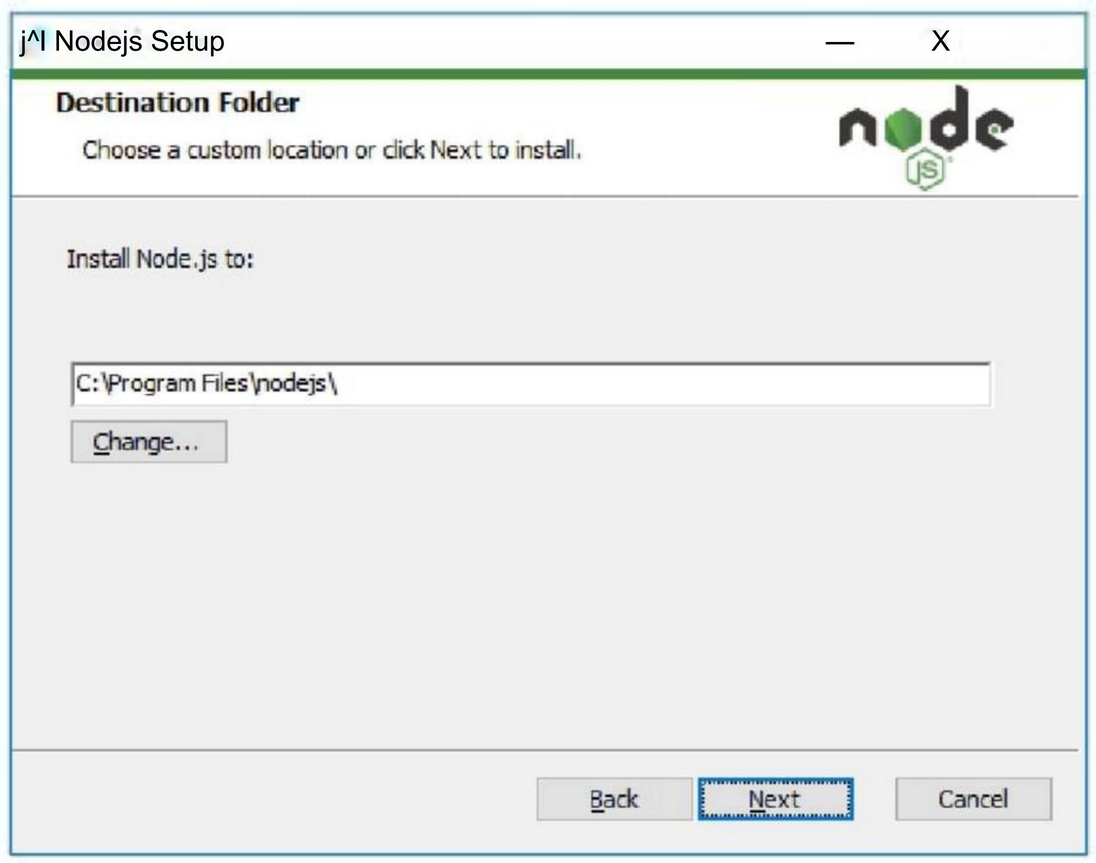


Figura 2.23: Instalador de node.js indicando la ruta sugerida en la que se instalará el software

[5] El siguiente paso nos permite seleccionar lo que deseamos instalar. Lo normal es instalar todo. Es especialmente importante, que la opción que hace referencia a la modificación del PATH del sistema, también se lleve a cabo:

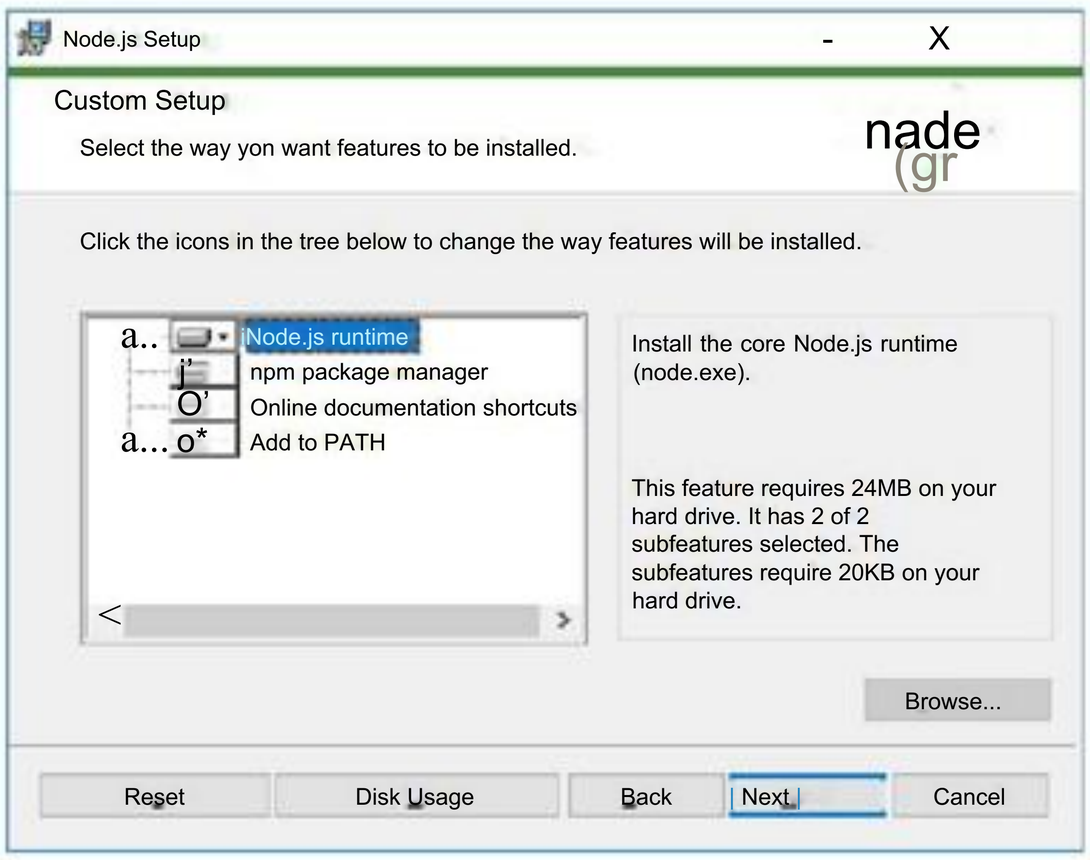


Figura 2.24: Selección del software que se instalará con node.js

[6] Aceptamos el resto de pasos.

[7] Para comprobar que node.js está correctamente instalado, abrimos un terminal del sistema y escribimos el comando **node.** Aparacerá el prompt de node (usa el símbolo >). Podemos escribir código JavaScript, escribiremos consolé. log("Hola");

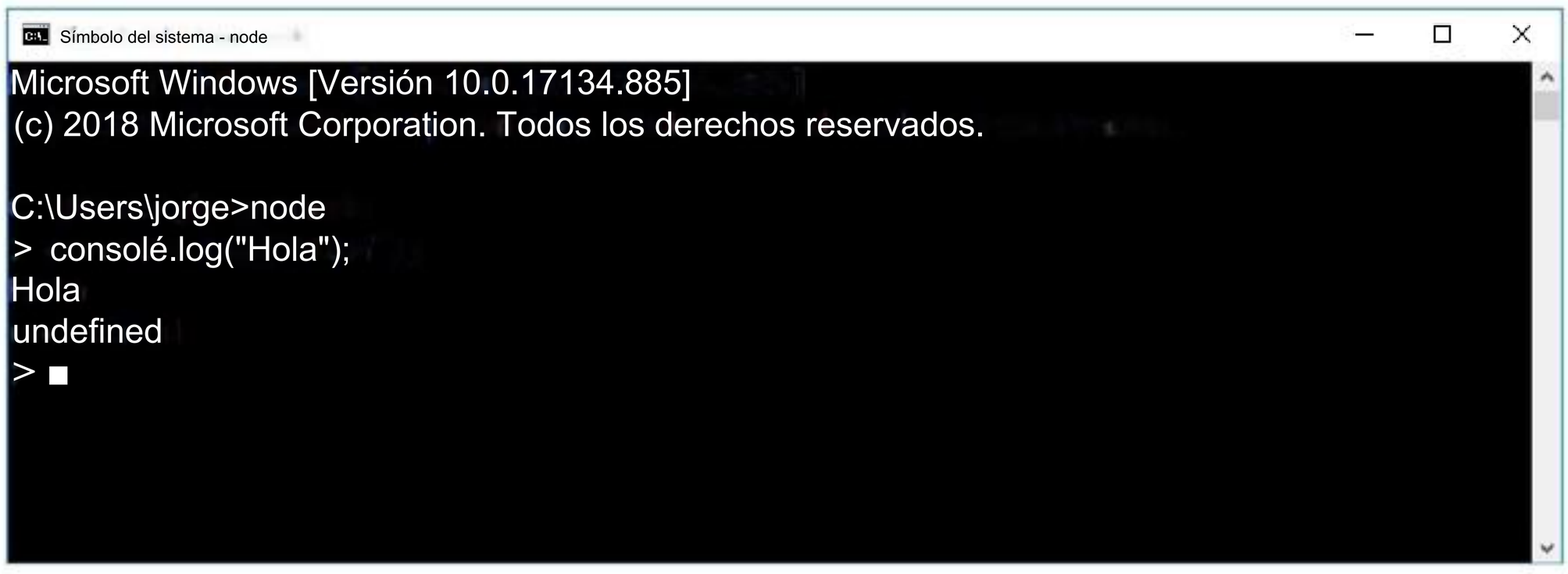


Figura 2.25: Comprobación de que node funciona

[8] Salimos del entorno de ejecución de node pulsando dos veces **Ctrl+C.**

[9] Vamos a ver cómo ejecuta node el código completo JavaScript de un archivo. Para crearlo rápidamente sin salir de la consola, suponemos que Visual Studio Code está correctamente instalado, escribimos y ejecutamos el comando: code pruebaOl. js

[10] Se abrirá Visual Studio Code mostrando en blanco el archivo indicado. Escribimos el código siguiente:

```
let mensaje="Hola de nuevo";
consolé.log(mensaje);
```

[11] Guardamos el archivo con ese contenido y cerramos Visual Studio Code. En el terminal que temámos antes escribimos el comando: node pruebaOl. js

[12] Se mostrará el mensaje ***Hola de nuevo,*** confirmando que hemos realizado correctamente todo el proceso.

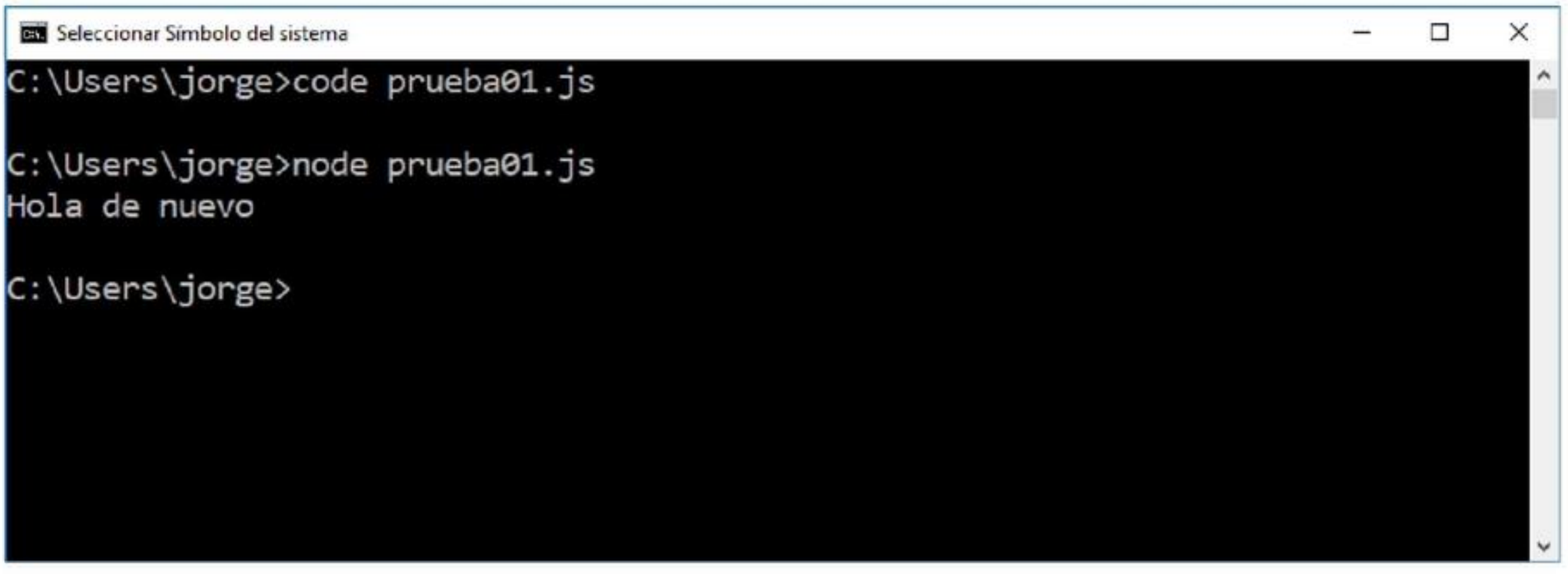


Figura 2.26: Comprobación de que el código del archivo creado funciona correctamente

---

Práctica 2.5: Instalación de nvm

- Ínstala nvm
- Ínstala la versión actual LTS de node.js a través de nvm
- Ínstala también la versión última disponible de node.js y la última versión 4 disponible
- Cambia de versión de node a través de nvm y comprueba cómo compila código la versión más vieja y la versión más nueva

---

SOLUCIÓN: PRÁCTICA 2.5

[1] Desinstalamos todas las versiones node.js que hayas instalado de forma convencional.

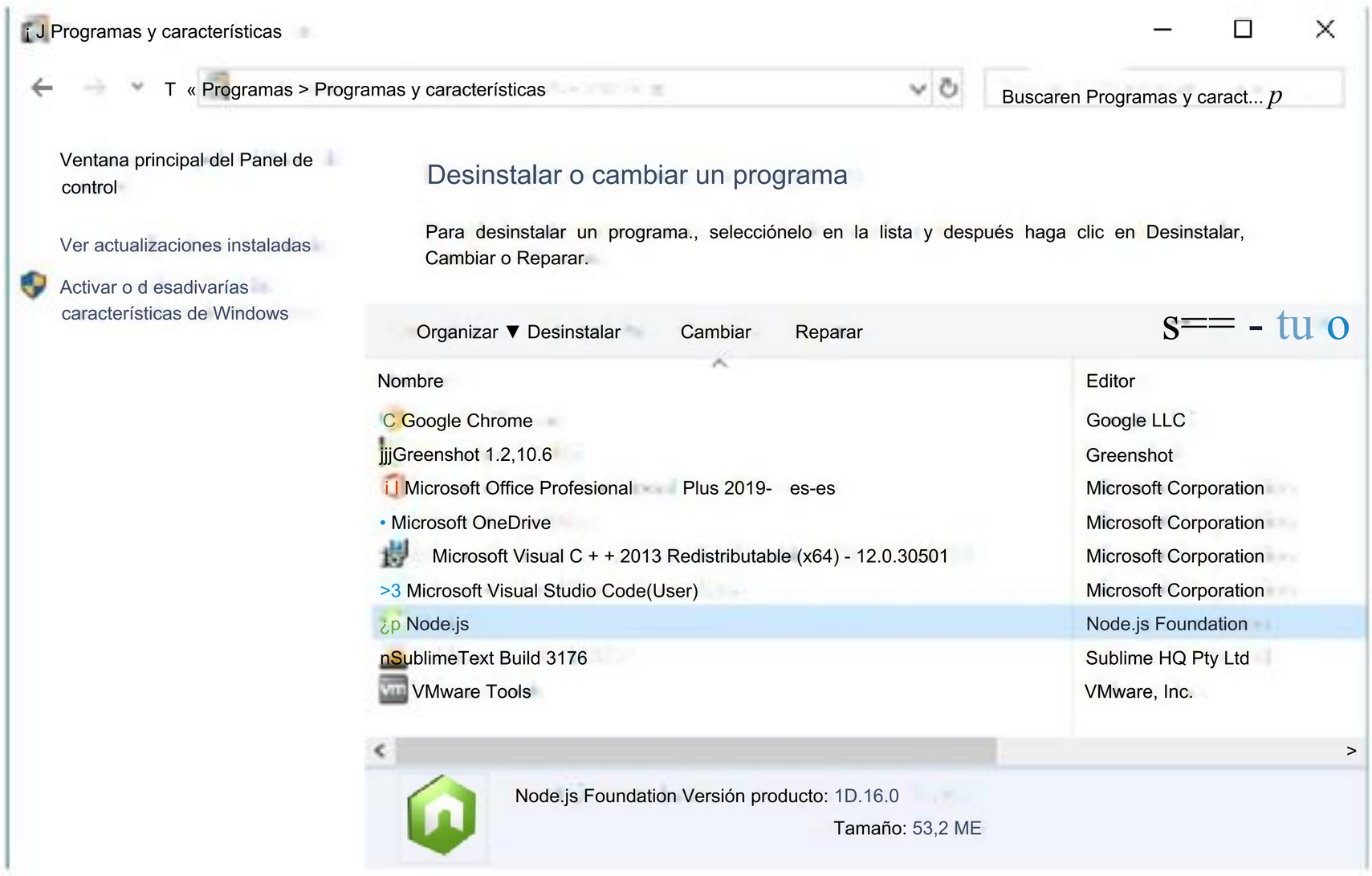


Figura 2.27: Panel de control con Node.js marcado para su desinstalación

[2] Vamos a la dirección https://github.com/coreybutler/nvm-windows/releases y descargamos el archivo **nvm-setup.zip.**



Figura 2.28: Página de descarga de nvm para Windows con el archivo de descarga recomendado resaltado

[3] Descomprimimos el archivo descargado y lanzar el instalador.

[4] Aceptamos la licencia y avanzamos al siguiente paso en el que se nos sugiere la ruta de instalación, que usará el directorio %APPDATA% del usuario actual.

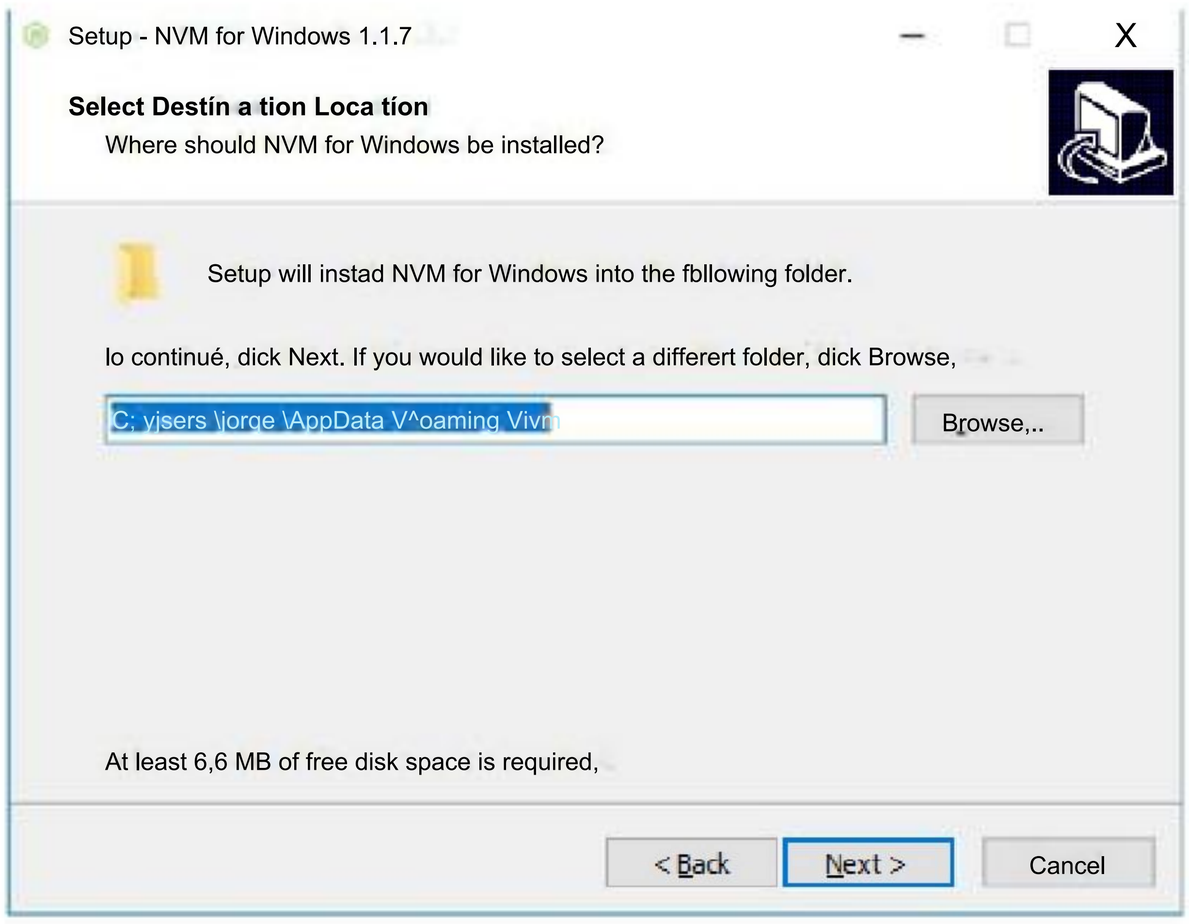


Figura 2.29: Ruta de instalación de nvm en Windows

[5] En el siguiente paso, se nos indica dónde se colocará la ruta simbólica a node.js. Los archivos de todas las versiones de node.js se instalan en la ruta anterior, pero se crea un directorio para que la versión en uso de node.js configurada funcione como si fuera la única instalada.

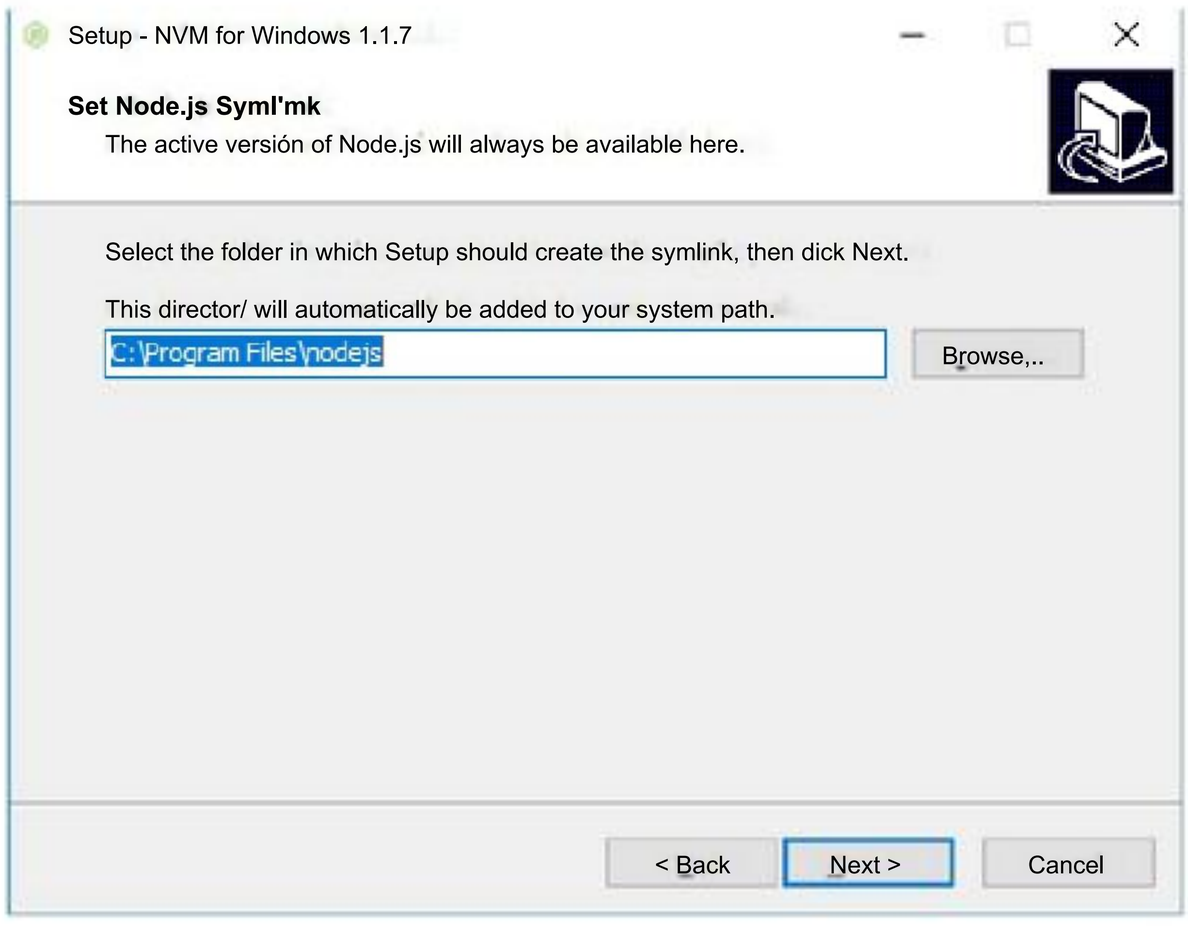


Figura 2.30: Ruta simbólica de instalación de node.js

[6] Comprobamos las versiones disponibles para instalar de node.js con el comando:

```
nvm ls available
```

[7] Aparece la lista completa de versiones

[8] Instalamos la última versión estable:

[9] Instalamos la última versión 4 de node y la versión 10.16.0:

[10] Comprobamos las versiones instaladas:

[11] Forzamos a usar la versión 4.9.1:

[12] Probamos si la versión de node en uso es esa:

[13] Cambiamos a la 10.16.0

[14] Comprobamos de nuevo las versiones instaladas:

```
  * 10.16.0 (Currently using 64-bit executable)
    4.9.1
```

[15] Desinstalamos la versión 4.9.1

```
>nvm uninstall 4.9.1
Uninstalling node v4.9.1... done
```

Práctica 2.6: Instalación de paquetes globales con npm

- Instala el paquete **yeoman** mediante **npm,** de forma global
- Comprueba que el paquete funciona correctamente

SOLUCIÓN: PRÁCTICA 2.6

[1] **Yeoman** es una utilidad potentísima que permite crear la base de aplicaciones web de todo tipo para empezar a trabajar. No nos interesa realmente aprender mucho de **yeoman,** pero es un ejemplo de paquete potente que puede ser instalado con ayuda de npm.

[2] Instalar yeoman de forma global con npm:

```
> npm install -g yo
```

[3] Se descarga el paquete y ya dispondremos del comando **yo** en la línea de comandos.

[4] El paquete yeoman requiere de otros paquetes para generar la carcasa inicial de una aplicación web del tipo que elijamos. Los paquetes de este tipo se llaman **generator-** seguido del tipo de generador que queremos instalar. Nosotros instalaremos el generador **webapp** que crea la carcasa de una aplicación HTML5 con muchas utilidades ya preparadas. Para instalar el paquete ejecutamos en la consola:

```
> npm install -g generator-webapp
```

[5] Tras la instalación, nos ubicaremos desde la consola en un directorio preparado por nosotros, que será en el que crearemos la aplicación web.

[6] Escribiremos el comando:

```
> yo webapp
```

[7] En este caso se nos pregunta por los componentes que queremos instalar. En el ejemplo que se muestra debajo, solo instalaremos **SASS** y **Bootstrap:**

```
     _-----_     ╭──────────────────────────╮
    |       |    |  'Allo 'allo! Out of the |
    |--(o)--|    |     box I include HTML5  |
   `---------´   | Boilerplate, jQuery, and |
    ( _´U`_ )    | a gulpfile to build your |
    /___A___\   /|          app.            |
     |  ~  |     ╰──────────────────────────╯
```

```
   ,__,  .__.  ,__
  " x | o z Y '

? Which additional features would you like to inelude? (Press <space> to
select,
  <a> to toggle all, <i> to invert selection)
>(*) Sass
  (*) Bootstrap
  ( ) Modernizr
  ( ) Google Analytics
```

[8] También se nos pide elegir el tipo de aplicación para testear con el componente Mocha. Elegimos la primera.

[9] Tras un tiempo de instalación, se crearán todos los directorios y archivos de nuestra aplicación ya semipreparada. Podemos examinarlos, veremos una gran cantidad de archivos. La mayoría no tendrán significado para nosotros. A medida que progresemos en este tipo de materia, entenderemos el tipo de aplicación que se ha empezado a preparar gracias a **Yeoman.**

[10] En todo caso, es un buen ejemplo de uso de paquetes npm.

Práctica 2.7: Instalación de Git

- Instala Git en entorno Windows
- Comprueba que disponemos del comando **git** en la línea de comandos

---

SOLUCIÓN: PRÁCTICA 2.7

[1] Nuevamente explicaremos la instalación en Windows. Empezamos yendo a la dirección https://git-scm.com/ ***y*** desde esa página descargamos el software instalador de Git para Windows.

[2] Ejecutamos el archivo descargado.

[3] Leemos la licencia y pulsamos **Next,** si estamos de acuerdo con ella.

[4] En el siguiente paso se nos muestra la ruta de instalación.

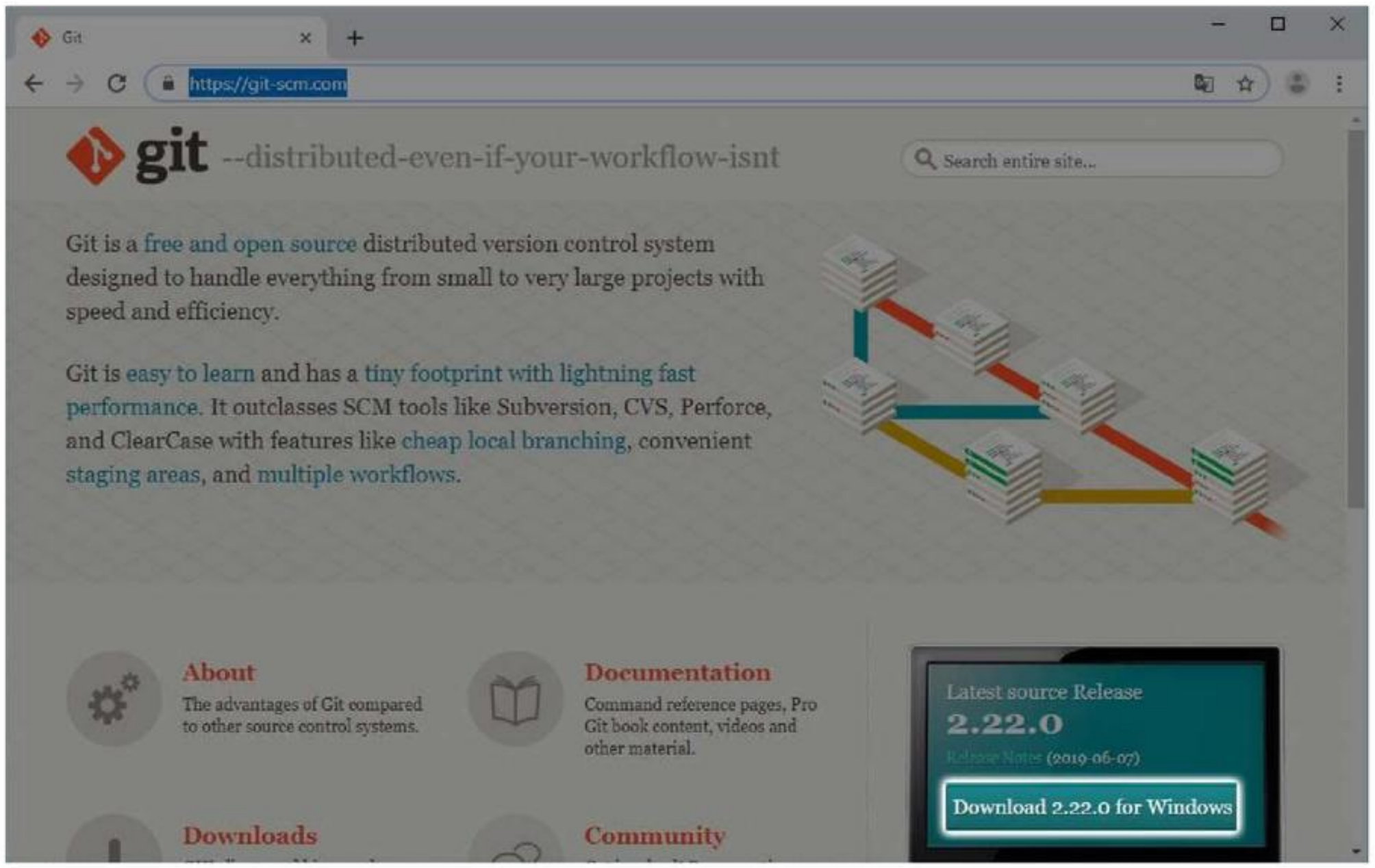


Figura 2.31: Página de descarga de Git para Windows con el botón de descarga resaltado.

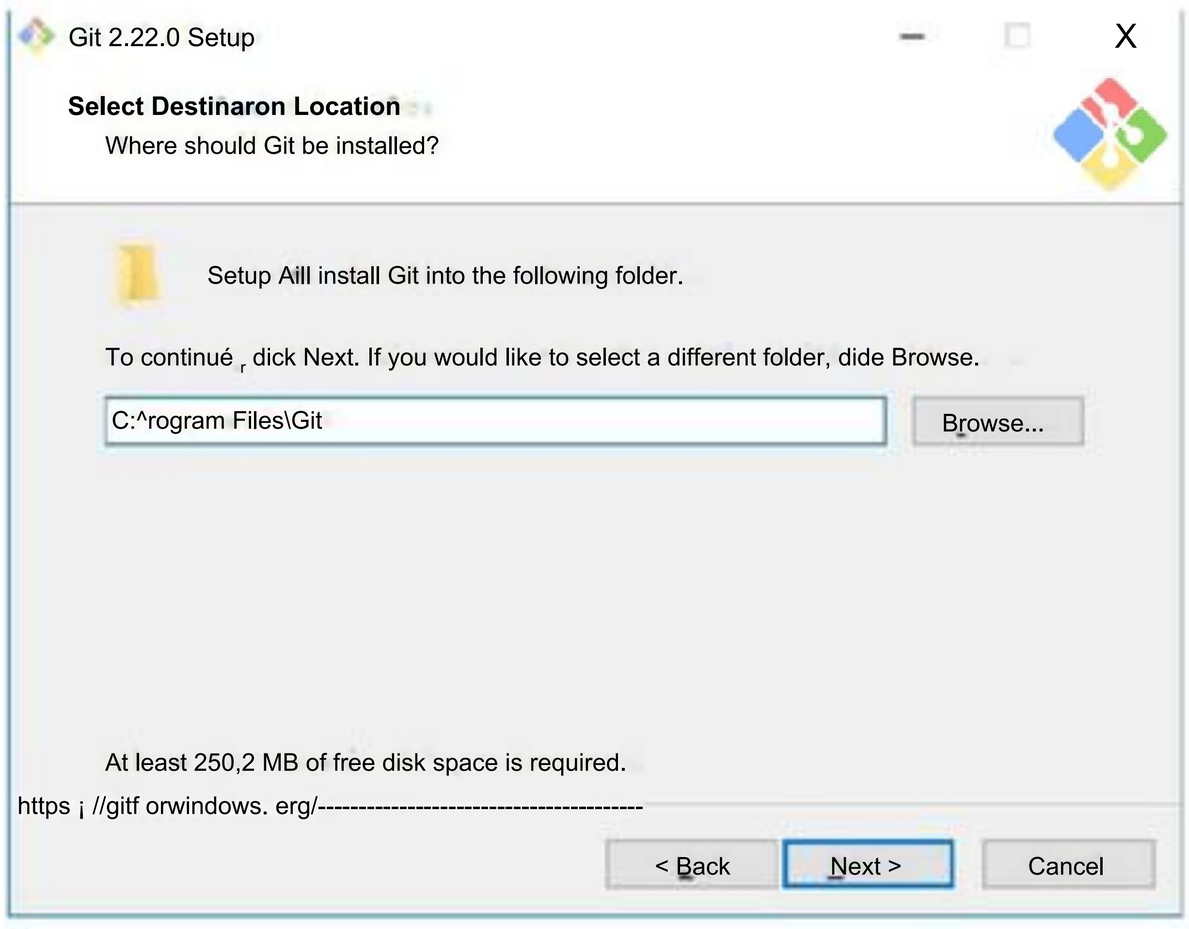


Figura 2.32: Ruta de instalación de Git en Windows

[5] Marcamos las opciones de instalación deseadas. La Figura 2.33 muestra las opciones que habitualmente se suelen marcar.

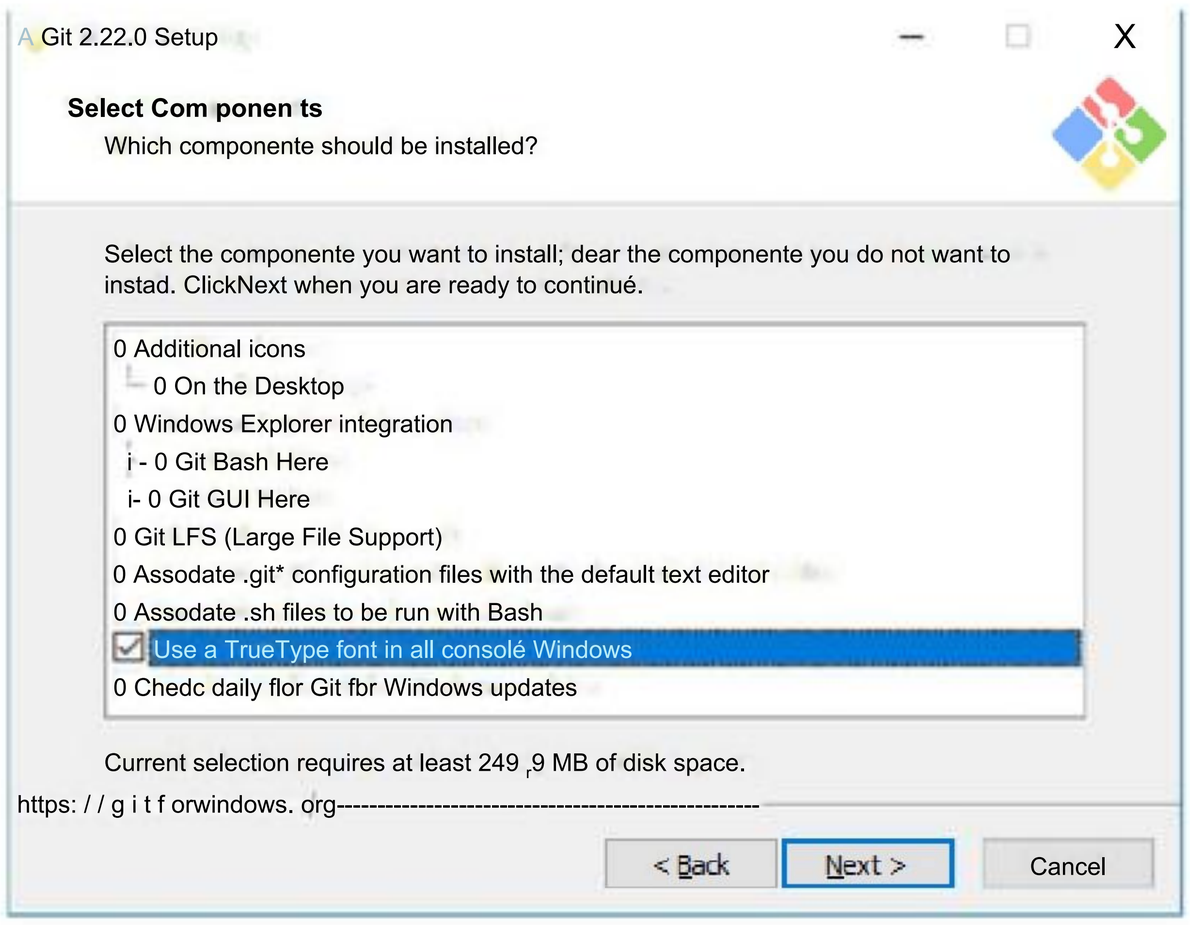


Figura 2.33: Opciones habitualmente marcadas en la instalación de Git para Windows

[6] Se nos preguntará el nombre que daremos al grupo de programas en el menú de Inicio. El nombre propuesto: **Git,** parece el adecuado.

[7] En el siguiente paso, se nos pregunta por el software que queremos usar como editor por defecto para los comandos Git. En nuestro caso, lo lógico sería usar Visual Studio Code, pero hay más opciones posibles:

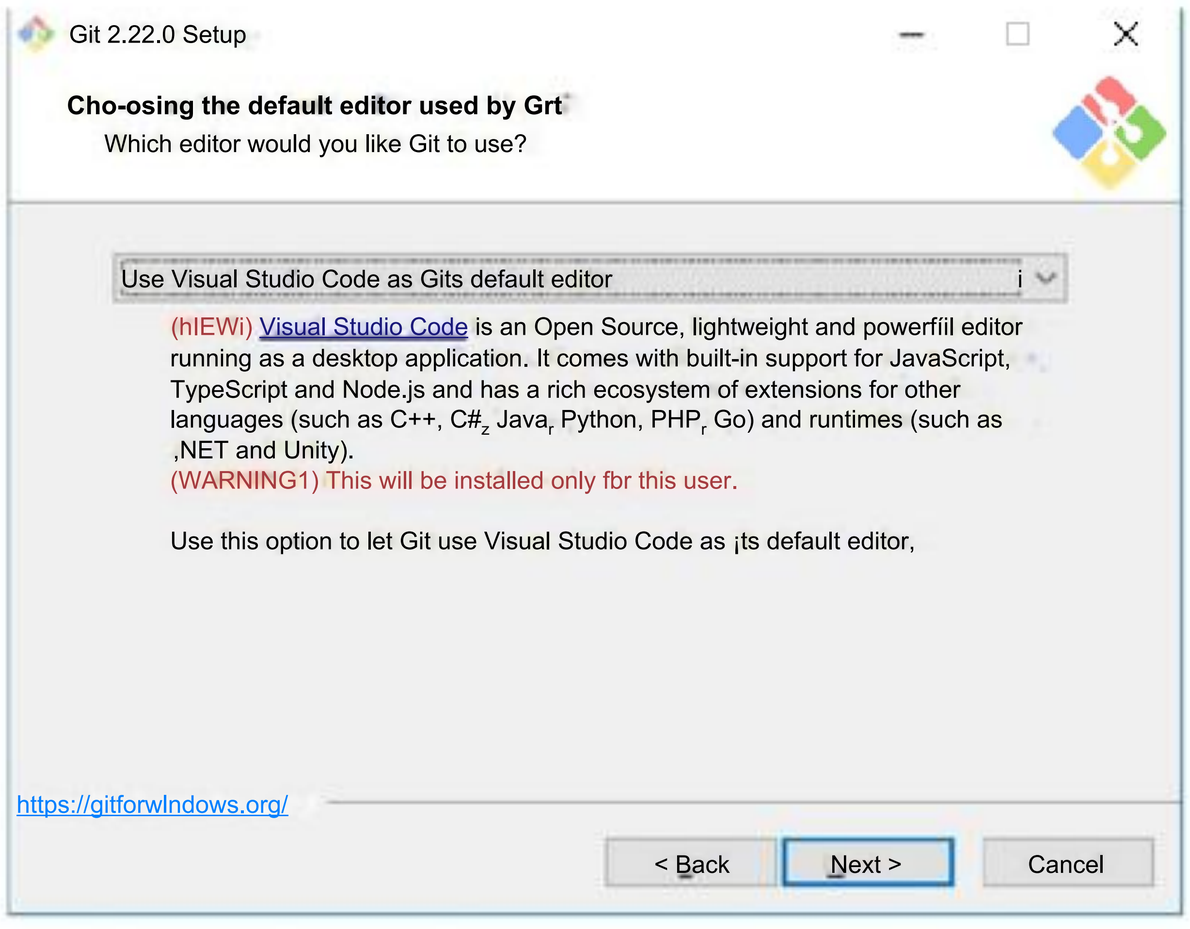


Figura 2.34: Visual Studio Code marcado como editor por defecto de Git

[8] El siguiente paso nos permite indicar dónde estarán disponibles los comandos Git. La primera opción solo los deja disponibles para el terminal Git Bash que se instala junto conm Git. La segunda los coloca disponibles en cualquier terminal del sistema. La tercera es como

la segunda pero, además, añade comandos Linux en el terminal de Windows. Es recomendable esta opción si queremos usar este tipo de comandos.

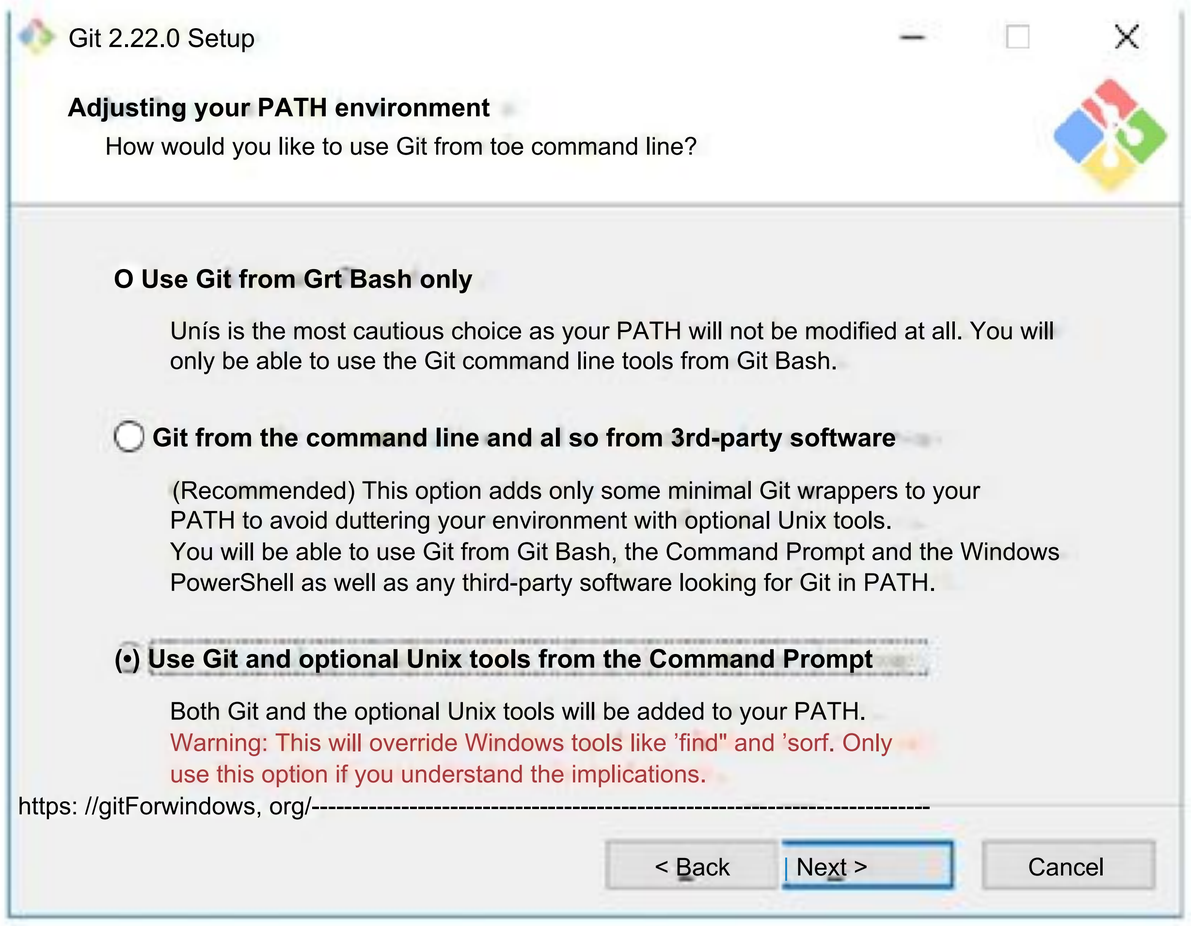


Figura 2.35: Indicación de dónde deseamos los comandos Git. La tercer opción añade comandos típicos de Linux al propio sistema Windows.

[9] En el paso siguiente se nos pregunta por la librería a usar para las comunicaciones cifradas. En caso de duda habría que elegir la librería **OpenSSL.**

[10] Se nos permite elegir cómo queremos marcar el final de línea, si al estilo de los sistemas Windows o al estilo de los sistemas Linux.

[11] El siguiente paso nos permite elegir el terminal para usar con Git. Lo normal es elegir el terminal **MinTTY.**

[12] Revisamos los últimos pasos y finalizamos la instalación.

[13] Para comprobar la instalación de Git, vamos a la consola del sistema y escribimos:

```
> git —versión
```

[14] Aparecerá el número de versión de Git instalada.

[15] Conviene configurar los parámetros globales con nuestro nombre y email con el que asociar nuestras grabaciones de instantáneas Git, por ejemplo:

```
> git conñg —global user.ñame "Jorge Sánchez"
> git conñg —global user.email "jorgesanchez@jorgesanchez.net"
```

Práctica 2.8: Uso de Git para crear versiones

- Crea una carpeta llamada **Practica2-8.**
- Prepara esa carpeta para usar con Git.
- Crea **index.html** como una página que tiene un título que dice ***"Practica 2.8".***

- Crea **estilos.css** dentro de una carpeta llamada CSS con código que permita mostrar el título en letra genérica sans-serif y una línea por debajo.
- Graba esa versión del código con Git como primera instantánea del mismo.
- Modifica la hoja de estilos para que el texto salga centrado. Añade a la página **index.html** un párrafo que diga ***"Aprendiendo Git".***
- Cambia la hoja de estilos para que el texto del párrafo tenga un sombreado gris claro.
- Graba esta versión del código.
- Consigue cambiar de una versión a otra mediante comandos Git y comprueba cómo se modifican los archivos.

SOLUCIÓN: PRÁCTICA 2.8

[1] Hay muchas maneras de empezar. Por ejemplo, podemos crear el directorio desde Windows (podemos usar carpetas creadas para las prácticas anteriores) y abrir dicho directorio en Visual Studio Code para trabajar más cómodos ahí.

[2] Una vez en el editor. Creamos el archivo **index.html** que tendrá este código:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
initia1-scale=l.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Práctica 2.8</title>
        <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
        <hl>Práctica 2.8</hl>
</body>
</html>
```

[3] Después creamos la carpeta **css** y en ella el archivo **estilos.css** con este código:

```
hl{
        font-family: sans-serif;
        border-bottom: lpx solid black;
}
```

[4] Ya tenemos nuestra primera versión del código ahora vamos a empezar a trabajar con Git. Empezamos creando el repositorio Git. No hace falta salir de Visual Studio Code, tenemos el terminal disponible en el propio Visual Studio Code, pulsamos **Ctrl+ñ** y accederemos al terminal (el menú Ver-Terminal es otra opción para conseguirlo).

[5] Desde el terminal, asegurándonos que estamos en la raíz de nuestra carpeta para esta práctica, escribimos el comando:

```
git init
```

[6] Un detalle interesante es que el botón dedicado al control de versiones, dentro del propio Visual Studio Code, ya se habrá percatado de que usamos Git y nos aparecen dos avisos relativos a los dos archivos que tienen cambios no guardados en repositorios Git.



Figura 2.36: Visual Studio Code indicando, en el lado izquierdo, los archivos cuyas modificaciones no están confirmadas

[7] Podemos ahora escribir más comandos en el terminal, o bien confirmar los cambios directamente con el botón confirmar del panel de Visual Studio Code. Al hacerlo, Visual Studio Code nos dice que no hemos guardado provisionalmente los cambios (lo que hace normalmente el comando **git add),** lo hacemos y aceptamos.



Figura 2.37: Detalle d el panel de control de código fuente con el botón de confirmación resaltado

[8] Un nuevo panel nos permite indicar el mensaje para la confirmación de grabación de la instantánea. Escribiremos el texto ***Título con borde.*** Pulsamos **lntro.**

[9] Una buena idea es instalar la extensión **GitLens** en Visual Studio Code que nos va a permitir usar un nuevo botón en el lado izquierdo del editor, desde el que podremos realizar acciones avanzadas sobre nuestras versiones del código.

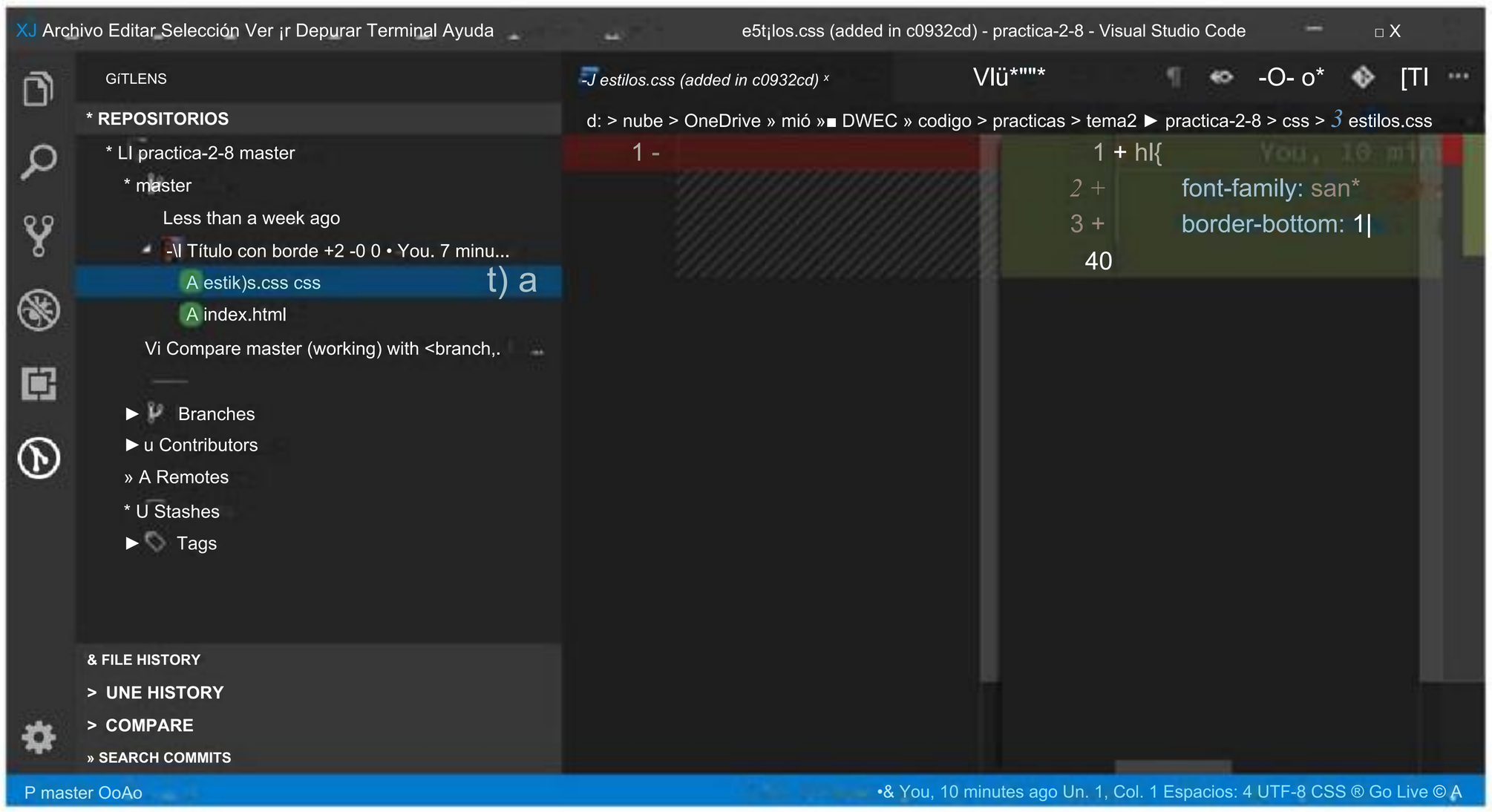


Figura 2.38: Visual Studio Code con GitLens instalado y mostrando los cambios en el archivo estilos.css

[10] Volvemos a ver los archivos de forma normal. Hacemos clic en el icono de navegación, primer botón del panel de iconos de la izquierda.

[11] Añadimos en el archivo **index.html** el código siguiente, el siguiente párrafo:

```
<p>Aprendiendo Git</p>
```

[12] Añadimos este código al archivo **estilos.css:**

```
PÍ
        background-color: lightgray;
}
```

[13] Confirmamos los cambios en la pestaña de Control de código fuente. El mensaje de confirmación será ***Párrafo gris.*** Esto se realiza igual que en el paso 7.

[14] Si volvemos al panel de **GitLens.** Veremos que en la rama **master** (rama que aparece por defecto) tenemos ya dos instantáneas del código.

[15] La forma de volver a un commit (instantánea) anterior usando GitLens sería ir al apartado branches y buscar el mensaje de commit al que deseamos regresar. Por ejemplo, el primer mensaje: **Título con borde.** Pulsando el botón secundario sobre ese mensaje y eligiendo **Checkout to commit** del menú contextual, conseguiremos que el código cambie para que aparezca tal cual estaba en ese commit.

[16] La opción de menú nos lleva al terminal, ahí tendremos que aceptar el comando que aparece para que realmente se lleve a cabo la operación. El comando será:

```
> git checkout númeroDeCommit
```

[17] Tras aceptar el comando, desaparecerá el párrafo con el texto ***Aprendiendo Git*** y el código con el sombreado gris dentro del archivo ***estilos.css.***

[18] Volvemos al commit del párrafo gris haciendo la misma operación sobre la instantánea con el mensaje ***Párrafo gris***

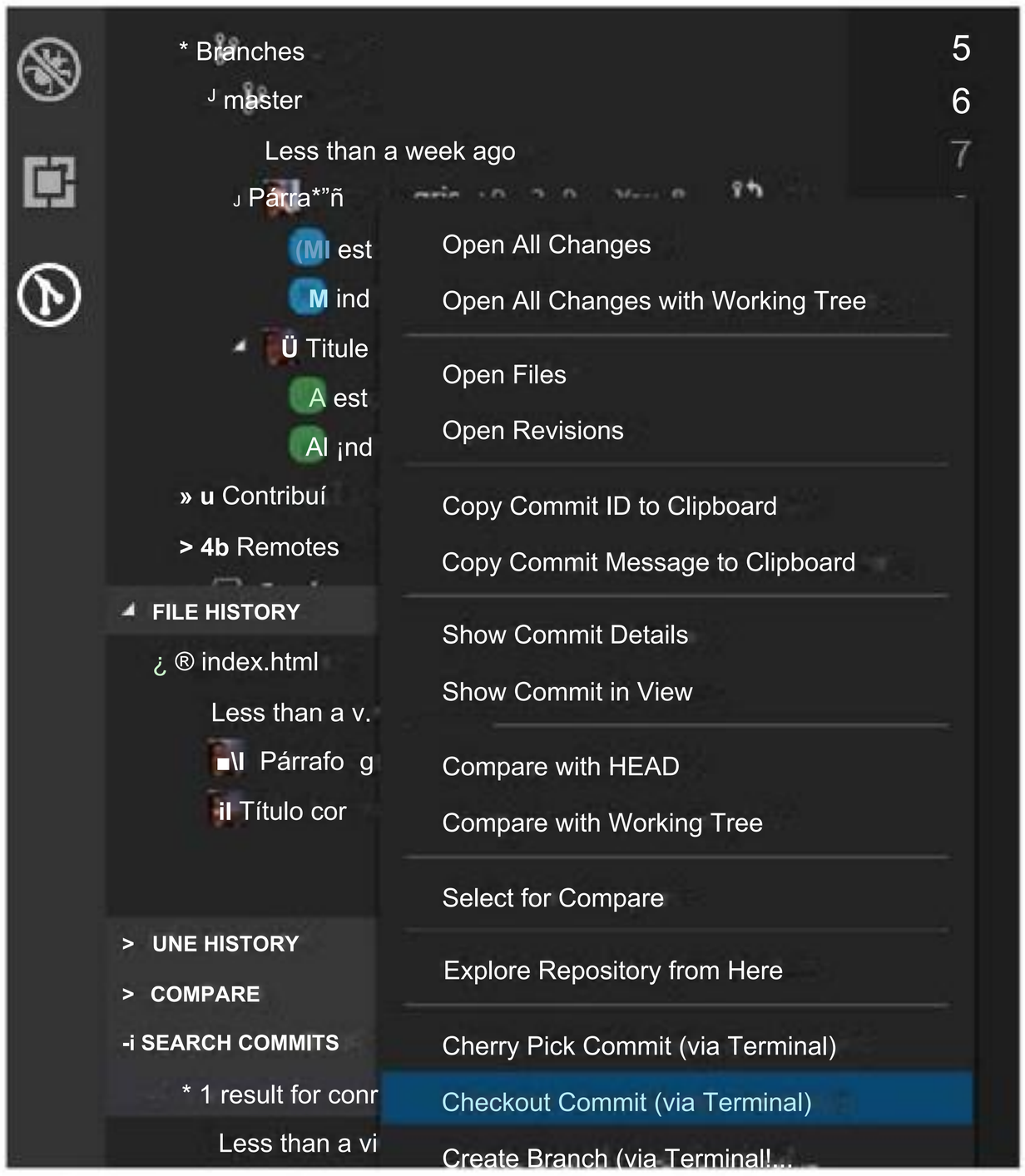


Figura 2.39: Panel de GitLens a punto de elegir la opción que deja el código en la instantánea que tiene el mensaje ***"Párrafo gris"***

Práctica 2.9: Subir nuestro código a GitHub

- Necesitas cuenta en GitHub para hacer esta práctica
- Sube el proyecto de la Práctica 2.8 a un nuevo repositorio de tu cuenta de GitHub

SOLUCIÓN: PRÁCTICA 2.9

[1] Crea, si no la tienes ya, una nueva cuenta en GitHub. No explicaremos aquí como hacerla, pero es el mismo proceso que en cualquier otro servicio de Internet.

[2] Crea un nuevo repositorio en GitHub. Pon el nombre que quieras al repositorio. Nosotros utilizaremos el nombre ***practica-2-8.*** Para ello, entra en tu cuenta de GitHub y pulsa el botón New en la zona de repositorios.

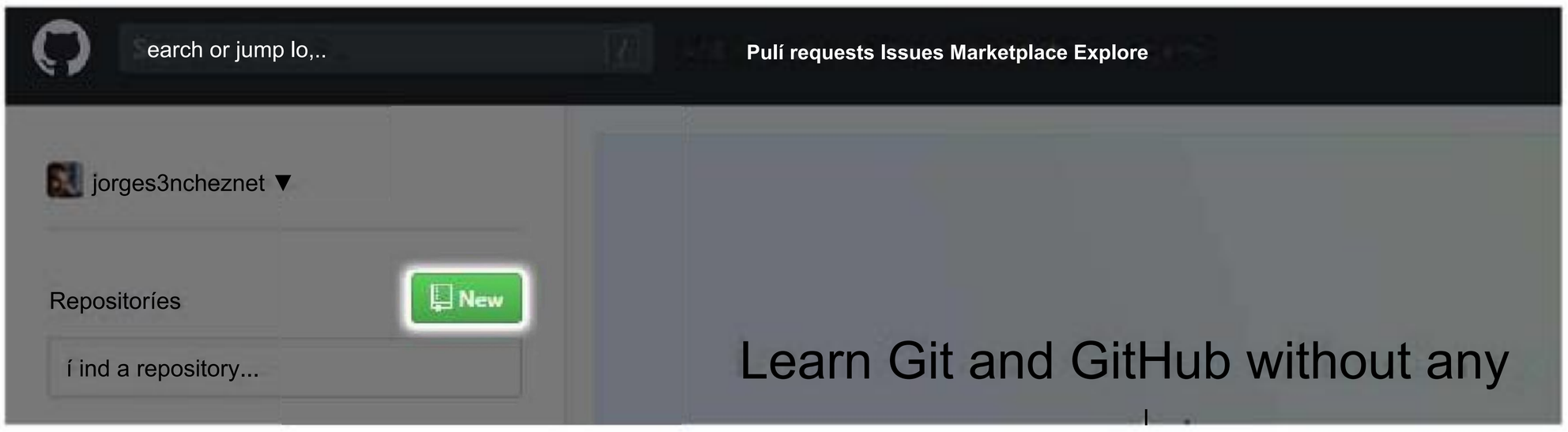


Figura 2.40: Panel de GitHub con el botón para crear nuevos repositorios resaltado

[3] Pon nombre al repositorio y elige público o privado si quieres que sea oculto o no.

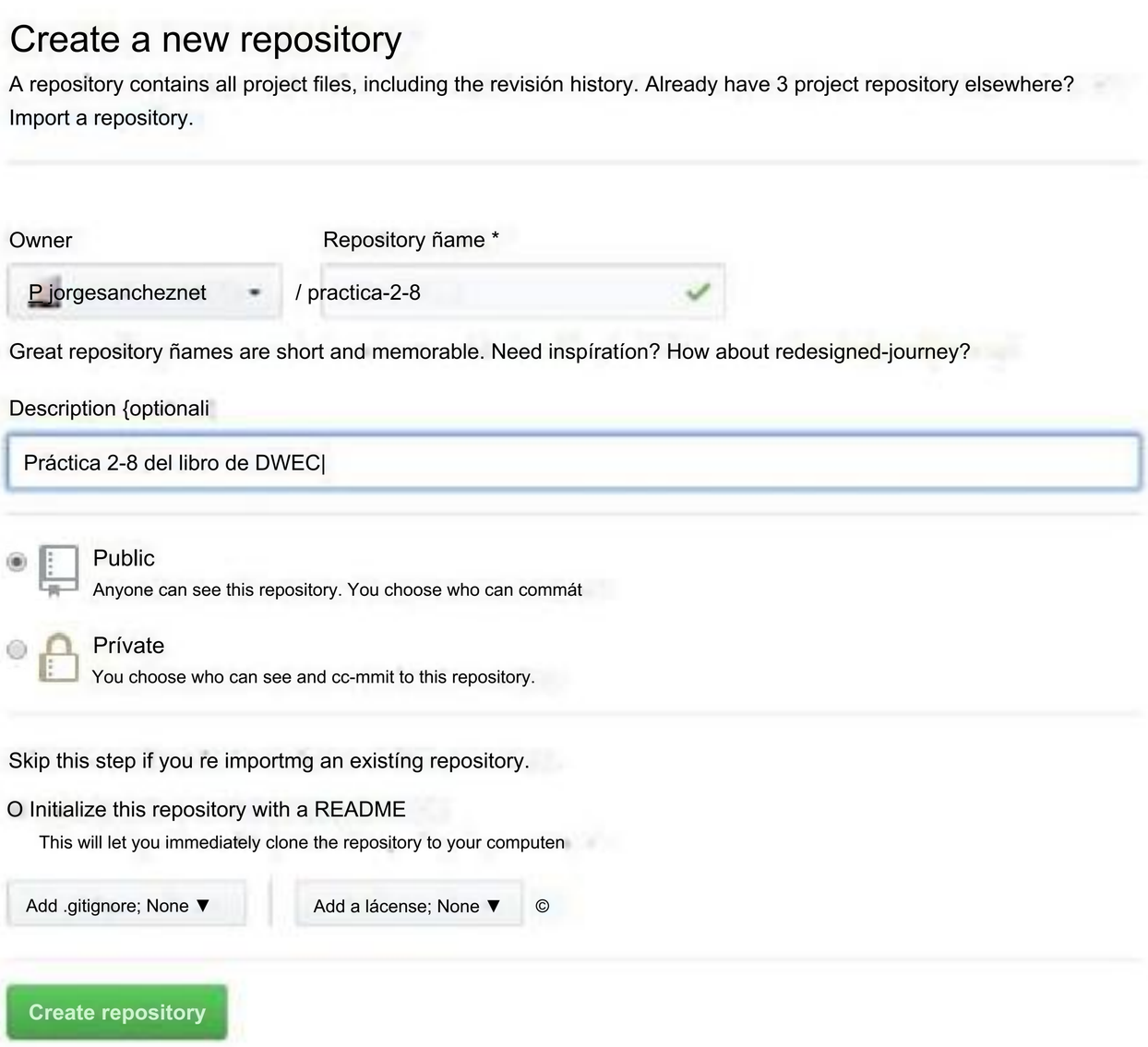


Figura 2.41: Opciones del nuevo repositorio de GitHub

[4] Pulsa el botón ***Create repository.*** El repositorio estará creado

[5] Aparece una pantalla en la que se nos sugieren los comandos a utilizar para empezar a trabajar con el repositorio remoto. Copiaremos el primer comando (el que empieza por **git remote...).**

[6] Desde Visual Studio Code abrimos el terminal **(Ctrl+ñ)** y pegamos el comando copiado. Pulsamos lntro para ejecutar el comando. No aparece ningún mensaje, pero el comando se habrá ejecutado.

[7] A partir de ese momento, cada vez que confirmemos los cambios, podremos subirlos al repositorio remoto. Para ello, deberemos ir al ***Panel de control de código fuente*** y hacer clic

en el botón con tres puntos y elegir **Publicar rama.** Si todo va bien, podremos ver la publicación en el apartado de remotos en el panel de GitLens.

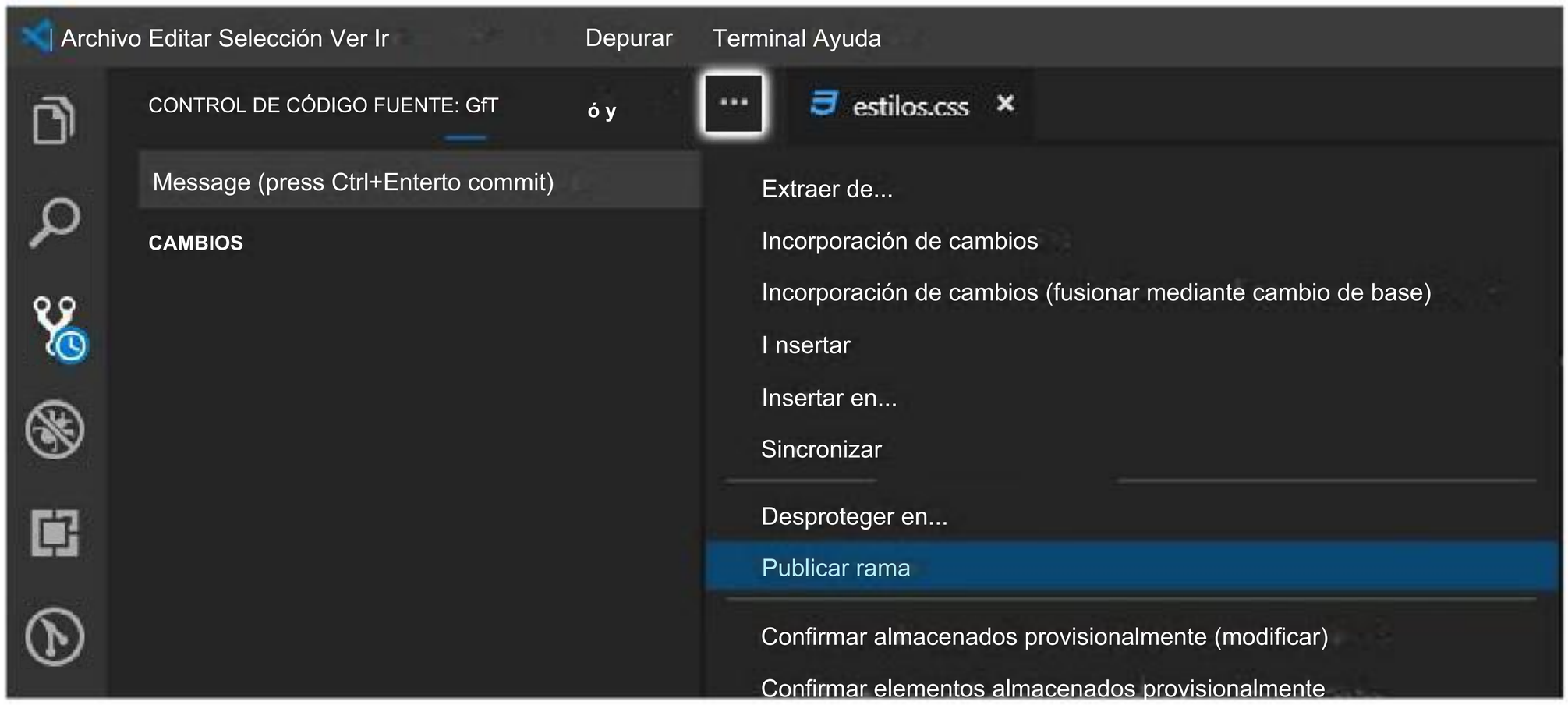


Figura 2.42: Mentí para publicar ramas (se ha resaltado el botón del que sale el menú)

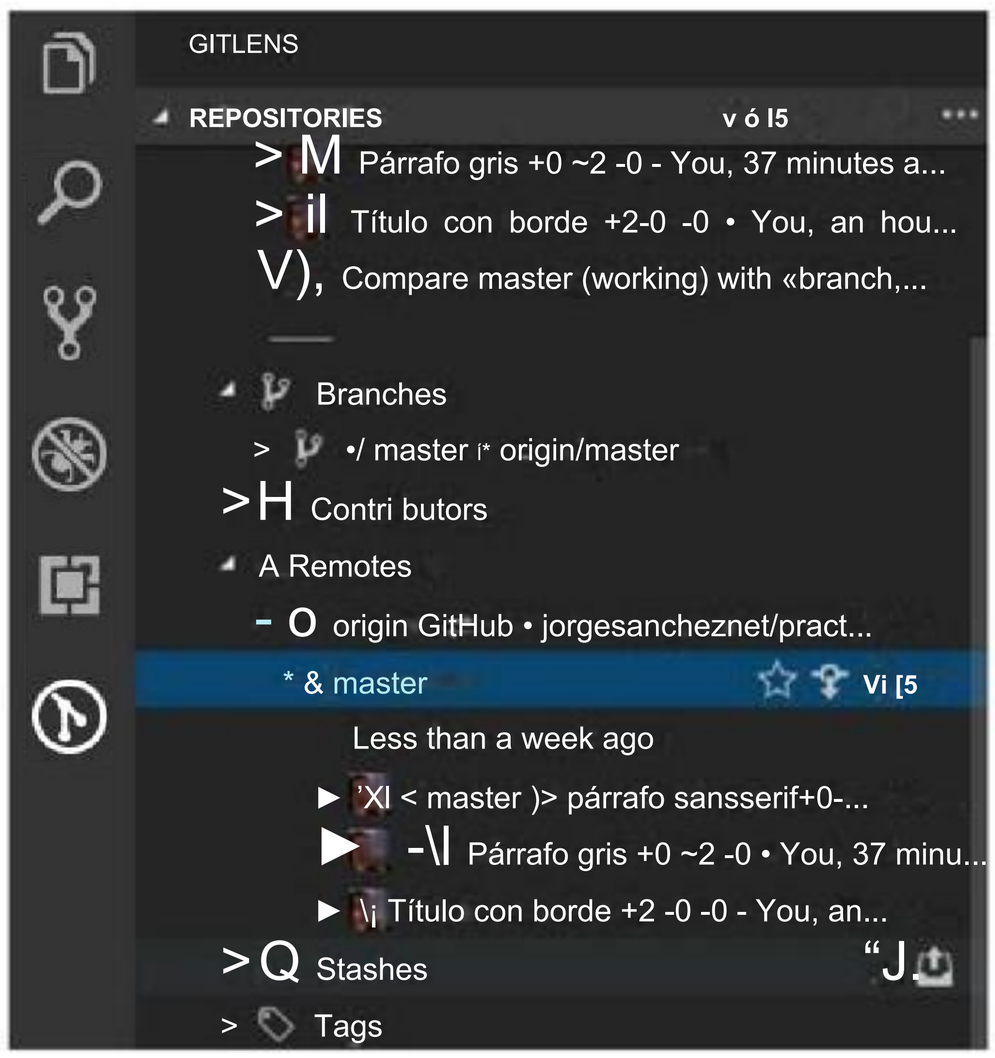


Figura 2.43: Panel de GitLens mostrando las publicaciones en el repositorio remoto de GitHub

2.7 RESUMEN DE LA UNIDAD

- Es importante que las personas que trabajan en el desarrollo de aplicaciones web dispongan de un buen entorno de trabajo.
- A este respecto las herramientas imprescindibles son:
  - Los navegadores principales.
  - Un buen editor de código.

■ Además es conveniente instalar:

- El intérprete de JavaScript **node.js** para poder ejecutar código JavaScript fuera del navegador.
- El instalador de paquetes **npm** que viene incorporado a node.js y que permite instalar herramientas y utilidades muy interesantes.
- Un sistema de control de versiones para nuestro código. El más popular es **Git.**
- Un servicio en la nube para alojar nuestro código y que sea compatible con nuestro sistema de control de versiones. Sin duda **GitHub** es el servicio mas popular.

- Que nuestro editor pueda integrar en un solo entorno todas las herramientas, es una posibilidad muy interesante que nos permitirá mejorar la productividad. En ese sentido, nuestro editor deberá proporcionar extensiones y plugins para poder integrar todas nuestras necesidades en el mismo sistema. Visual Studio Code es uno de los editores que permite integrar todas las herramientas en el mismo sistema.
- A medida que los profesionales aprenden nuevas destrezas, se detectan de forma más precisa las herramientas que se necesitan, de modo que cubran todos nuestros requisitos.

2.8 TEST DE REPASO

1.- **¿Qué le falta al bloc de notas para ser un buen editor de código?**

a) Coloreado de sintaxis
b) Posibilidad de administrar extensiones
c) Corrección de código en caliente
d) Todo lo anterior

2.- **¿Qué es nvm?**

a) Un editor de código
b) Un intérprete de JavaScript
c) Un software capaz de administrar varias versiones de node.js
d) Un sistema de control de versiones

3.- **¿Qué es node.js?**

a) Un editor de código
b) Un intérprete de JavaScript
c) Un servicio de almacenamiento de código en la nube
d) Un sistema de control de versiones

4.- ¿Qué es GitHub?

a) Un editor de código
b) Un intérprete de JavaScript
c) Un servicio de almacenamiento de código en la nube
d) Un sistema de control de versiones

5. - ¿Qué es Git?

a) Un editor de código
b) Un intérprete de JavaScript
c) Un servicio de almacenamiento de código en la nube
d) Un sistema de control de versiones

6.- **¿Qué es Chrome Canary?**

a) Una versión del navegador Chrome compatible con Internet Explorer
b) Una versión del navegador Chrome para probar características nuevas
c) La última versión beta del navegador Chrome
d) Una versión del navegador Chrome compativle con Mozilla Firefox

7.- **¿Cómo podemos ver los paquetes instalados de forma global con npm?**

a) npm ls available
b) npm list available
c) npm list global
d) npm list -g

8.- **¿Cómo podemos ver las versiones instaladas de node.js en nuestro sistema con nvm?**

a) nvm ls
b) nvm ls available
c) nvm ls installed
d) nvm ls —installed

9.- **¿Qué empresa fabrica Visual Studio Code?**

a) Google
b) Adobe
c) Apple
d) Microsoft

10.- **¿Qué comando permite abrir Visual Studio Code desde el terminal?**

a) code
b) vscode
c) vs-code
d) Ninguno

11.- **¿Qué comando permite iniciar un repositorio Git desde la raíz de un proyecto?**

a) git add .
b) git init
c) git commit
d) git push origin master

12.- **¿Qué comando Git podría servir para publicar el proyecto actual Git en un repositorio remoto?**

a) git add.
b) git init
c) git commit
d) git push origin master

UNIDAD 3

PRINCIPIOS DE PROGRAMACIÓN EN JAVASCRIPT

OBJETIVOS

- Identificar las distintas formas de ejecutar código JavaScript
- Reconocer las normas básicas de escritura en lenguaje JavaScript
- Identificar los distintos tipos de datos de JavaScript y la forma de definir su información de forma literal
- Crear variables y utilizarlas en el código asignando valores y operando entre ellas
- Identificar los operadores del lenguaje y aplicarlos correctamente dentro del código
- Resolver problemas de conversión de tipos de datos
- Utilizar cuadros de diálogo del navegador para informar y recoger información del usuario
- Utilizar instrucciones condicionales para crear mecanismos de decisión en el código
- Aplicar instrucciones de creación de bucles verificando su correcta utilización
- Aplicar bucles para resolver problemas complejos a través del código

CONTENIDOS

3.1 EJECUTAR CÓDIGO JAVASCRIPT
- 3.1.1 EJECUCIÓN DE JAVASCRIPT EN EL NAVEGADOR
- 3.1.2 EJECUCIÓN DE JAVASCRIPT FUERA DEL NAVEGADOR

3.2 FUNDAMENTOS BÁSICOS
- 3.2.1 CONCEPTOS FUNDAMENTALES

3.3 VARIABLES Y TIPOS DE DATOS
- 3.3.1 VALORES
- 3.3.2 DECLARACIÓN DE VARIABLES
- 3.3.3 TIPOS DE DATOS PRIMITIVOS

3.4 OPERADORES
- 3.4.1 ARITMÉTICOS
- 3.4.2 RELACIONALES
- 3.4.3 OPERADOR DE ENCADENAMIENTO
- 3.4.4 OPERADORES LÓGICOS
- 3.4.5 OPERADORES DE BIT
- 3.4.6 OPERADORES DE ASIGNACIÓN

3.5 CONVERSIÓN DE TIPOS
- 3.5.1 CONVERSIÓN AUTOMÁTICA
- 3.5.2 FORZAR CONVERSIONES
- 3.5.3 FUNCIÓN ISNAN

3.6 CUADROS DE DIÁLOGO DEL NAVEGADOR
- 3.6.1 MENSAJES DE ALERTA
- 3.6.2 CUADROS DE CONFIRMACIÓN
- 3.6.3 CUADROS DE ENTRADA DE TEXTO

3.7 CONTROL DEL FLUJO DEL PROGRAMA
- 3.7.1 INTRODUCCIÓN
- 3.7.2 NÚMEROS ALEATORIOS
- 3.7.3 INSTRUCCIÓN **IF**
- 3.7.4 INSTRUCCIÓN **WHILE**
- 3.7.5 BUCLE **DO...WHILE**
- 3.7.6 BUCLE **FOR**
- 3.7.7 ABANDONAR UN BUCLE
- 3.7.8 BUCLES ANIDADOS

3.8 PRÁCTICAS RESUELTAS

3.9 PRÁCTICAS PROPUESTAS

3.10 RESUMEN DE LA UNIDAD

3.11 TEST DE REPASO

3.1 EJECUTAR CÓDIGO JAVASCRIPT

En esta unidad vamos a comenzar a aplicar y comprender las bases del lenguaje JavaScript. Por lo tanto, lo que se ve en esta unidad es vital para poder progresar en el resto de unidades de este libro.

Lo primero que hemos de saber es cómo podemos ejecutar el código JavaScript. Hay que recordar que este libro es un manual de programación de aplicaciones web en entorno cliente, lo que implica que usaremos JavaScript para crear aplicaciones que se ejecuten en un navegador.

Pero también es cierto que podemos ejecutar JavaScript fuera del navegador (con ayuda de **node.js)** para aprender más rápidamente los fundamentos del lenguaje.

3.1.1 EJECUCIÓN DE JAVASCRIPT EN EL NAVEGADOR

JavaScript nació para ser un lenguaje interpretado por un navegador. Por ello, ejecutar JavaScript desde el código de una página web, sigue siendo considerado como el método principal de ejecución de este lenguaje. Los navegadores interpretan todo el código JavaScript que se encuentre dentro de una etiqueta script. Ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
	<meta charset="UTF-8">
	<title>Prueba Javascript</title>
</head>
<body>
<script>
	document.write("Hola desde Javascript");
</script>
</body>
</html>
```

La etiqueta **script** contiene el código JavaScript, que en el ejemplo anterior provoca (gracias al método ***documentwrite)*** que se escriba el texto ***Hola desde Javascript*** en el cuerpo de nuestra página web.

La etiqueta script se puede colocar tanto en la cabecera del documento (dentro de head) como dentro de la propia etiqueta **body.** Normalmente, se coloca en la cabecera si es código para cargar librerías general o código que define funciones y objetos globales, pero que no interacciona con el documento. Se coloca en el cuerpo y justo antes del body, el código JavaScript que se comunica con los objetos del documento.

Ahora bien, el código JavaScript, aun siendo colocado al final de body puede ejecutar su código antes de que se carguen todos los elementos de la página, ya que la carga de una página web se realiza de forma asincrona. De cómo solucionar este problema hablaremos más adelante.

3.1.1.1 ESCRITURA POR CONSOLA

Todos los navegadores actuales poseen una ventana de depuración para poder examinar, modificar y testear el código de las aplicaciones web a fin de verificar su eficacia y de poder hacer pruebas y ensayos en él. El lenguaje JavaScript posee un objeto llamado **consolé** que permite utilizar la consola de depuración desde el propio código. El método log se utiliza para escribir en la consola, de modo que esta página web:

```
<!DOCTYPE html>
<html lang=”es”>
<head>
        <meta charset=”UTF-8”>
        <title>Document</title>
</head>
<body>
<script>
        consolé.log(“Hola”);
</script>
</body>
</html>
```

Si ejecutamos este código en un navegador, no muestra nada en la ventana, pero en el panel de depuración veremos escrito el texto **Hola.**

Podemos ver la consola, dentro del panel de depuración, en prácticamente todos los navegadores pulsando F12 (algunos navegadores usan otras teclas). El panel de depuración sirve para hacer pruebas y detectar fallos en nuestras aplicaciones web. En este panel siempre hay una pestaña llamada ***Consola (Consolé).***

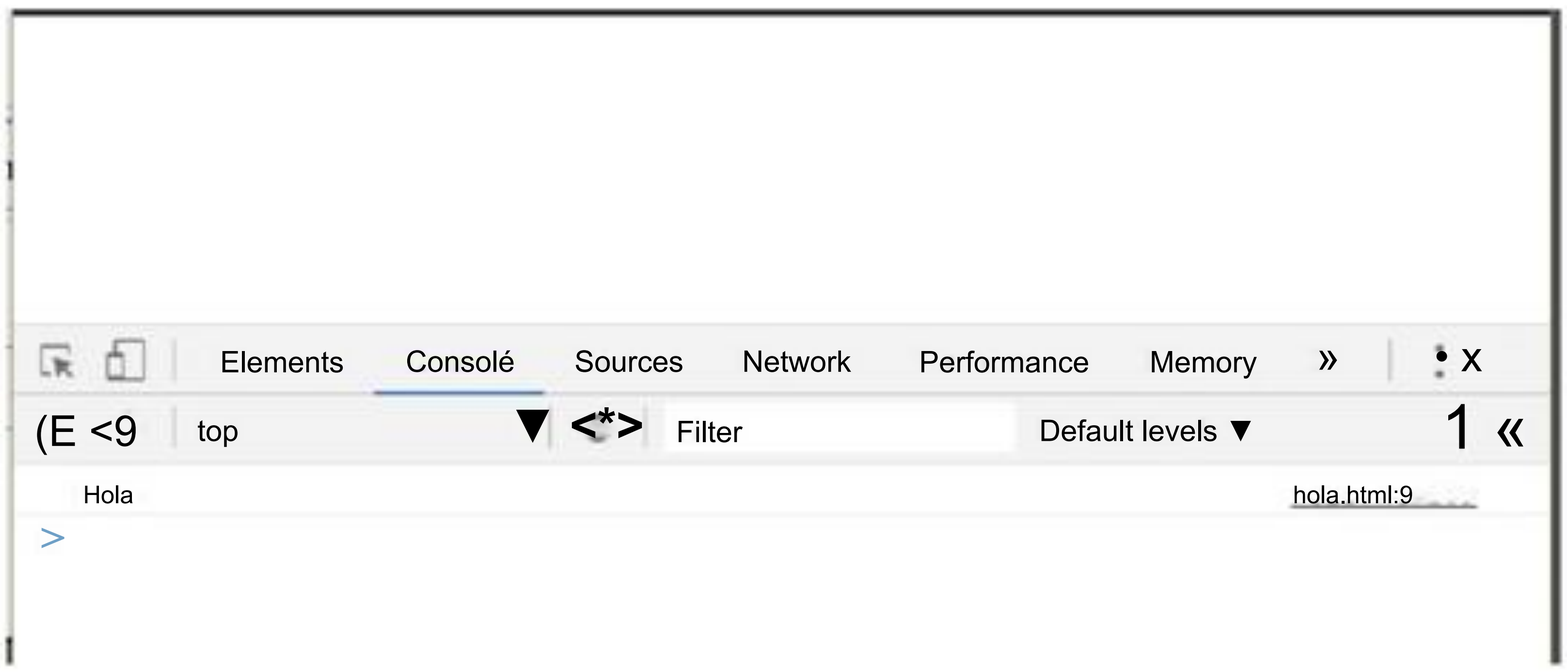


Figura 3.1: Google Chrome mostrando la consola de la página actual. El cuerpo, parte superior, aparece en blanco, pero sí se muestra ***Hola*** en la consola.

3.1.1.2 CARGAR ARCHIVOS EXTERNOS JAVASCRIPT

Si queremos, podemos escribir el código en otro archivo separado (al que se le debe colocar la extensión **js)** y, desde el código HTML, cargar su código. Para ello, la etiqueta **script** dispone de un atributo llamado **src** al que se le indica la ruta (URL) del archivo que queremos cargar:

```
<script src="js/acciones. js" > </script>
```

Esta etiqueta carga y ejecuta el código JavaScript del archivo **acciones.js** que se encuentra dentro del directorio js que estará, a su vez, en el mismo directorio en el que se encuentre la aplicación web.

ACTIVIDAD 3.1: EJECUTAR JAVASCRIPT EN ARCHIVO EXTERNO

- Realiza la Práctica 3.1 ***"Ejecutar código en consola desde archivo externo",*** en la página 105

3.1.1.3 ATRIBUTOS DE LA ETIQUETA SCRIPT

La etiqueta script, en HTML 4, requería, obligatoriamente, utilizar un atributo llamado **type** cuyo valor era el lenguaje concreto que se usa en el script (normalmente **type="JavaScrípt").** Actualmente, desde HTML 5, este atributo es opcional porque se da por hecho que lo normal es que el código sea JavaScript.

Cuando se cargan archivos externos, es decir, si se usa el atributo src, disponemos de más atributos:

- El atributo **async** que puede tomar los valores **true** (verdadero) o **false** (falso). Con valor true hace que la ejecución del código JavaScript sea asincrona. Es decir, se carga el código de forma independiente a la carga de elementos de la página.
- El atributo **defer** (con los posibles valores **true** o **false,)** en caso de valer true, indica que la carga del archivo se realizará cuando los componentes de la página web se hayan ya cargados. Es útil (si el navegador entiende este atributo) cuando el código que queremos cargar sirve para manipular el contenido de la página web.
- El atributo **integrity** se usa para poder verificar que el archivo JavaScript que estamos cargando realmente es el que deseamos cargar y no se ha manipulado de forma malévola en el servidor que lo contiene por parte de terceros. Para ello este atributo puede tomar un valor ***hash*** que el navegador puede usar para verificar el archivo.

  Actualmente, la librería de JavaScript más utilizada es **jQuery** que, hace un tiempo, tuvo el problema de que su código fue modificado con malas intenciones en sus propios servidores de Internet. Por ello, se recomienda cargar de forma remota utilizando este atributo. Por ejemplo:

```
<script
   src="https://code.jquery.com/jquery-3.4.1.js"
   integrity="sha256-WpOohJOqMqqyKL9FccASB9O0KwACQJpFTUBLTYOVvVU="
   crossorigin="anonymous"> </script>
```

  La carga de JavaScript desde este CDN[1] pedirá al navegador comprobar que al aplicar el algoritmo SHA-256 en el mismo, el código hash resultante es el que se indica. En definitiva, asegura que el archivo cargado no ha sido manipulado por un tercero.

---

[1] Un CDN es una red de distribución de contenidos ***(Content Distribution Network)*** una red pen-

- El atributo **crossorigin** permite establecer atributos relacionado con **CORS** (tema del que hablaremos más adelante). CORS ***(Cross-Origin Resource Sharing,*** Compartición de recursos de origen cruzado) es una técnica que permite añadir atributos a las peticiones http que, a veces, se necesitan para poder acceder a recursos de servidores remotos. El atributo **crossorigin** de la etiqueta script, permite configurar algunos de esos parámetros.

3.1.2 EJECUCIÓN DE JAVASCRIPT FUERA DEL NAVEGADOR

Si deseamos ejecutar JavaScript fuera del navegador, necesitamos la ayuda de node.js. En el tema anterior (véase 2.4.2 ***"Instalar node.js",*** en la página 27) ya hemos hablado de **node.js,** ahora solo nos interesa su capacidad para traducir código JavaScript. Podemos ejecutar directamente código JavaScript de forma rápida si, desde un terminal del sistema, invocamos a node y (tras la aparición del símbolo ">") escribimos el código deseado. Al pulsar lntro se ejecutará ese código. Ejemplo:

```
$ node
> consolé.log("Hola");
Hola
undefined
```

Vemos cómo el método **console.log** escribe directamente la palabra **Hola.** En node.js no hay página web y **console.log** muestra directamente en el terminal el resultado de escribir en la consola. Tras la escritura de la palabra ***Hola,*** en la línea siguiente se muestra el texto ***undefined*** que, aunque ahora no es posible entender su significado, adelantaremos que lo que hace es escribir el valor que devuelve la expresión (como esta expresión no retorna valor alguno, indica que el valor es indefinido).

Podemos seguir escribiendo código JavaScript, hasta que decidamos salir del entorno pulsando dos veces la combinación de teclas **Ctrl+C.**

También podemos escribir código JavaScript en un archivo (es recomendable que tenga extensión **js)** y después ejecutar el código de ese archivo utilizando node. Supongamos que hemos grabado un archivo que se llama **prueba.js,** entonces desde el terminal nos debemos situar en el directorio en el que está el archivo y ejecutar este comando:

```
$ node prueba.js
```

Veremos en el terminal el texto que el programa escriba por consola.

ACTIVIDAD 3.2: EJECUTAR JAVASCRIPT DESDE NODE

- Realiza la Práctica 3.2 ***"Ejecutar código en node",*** en la página 105

3.2 FUNDAMENTOS BÁSICOS

JavaScript es un lenguaje de programación cuya sintaxis y forma de escribir sentencias recuerda a la mayoría de los lenguajes que han tenido éxito en estas últimas décadas. Esto significa que, para personas que ya sepan programar en algún lenguaje de programación, el aprendizaje de las bases fundamentales, resulta muy sencillo.

Por ello, este primer capítulo puede resultar muy simple, pero, indudablemente, es de gran importancia porque en él pretendemos explicar los fundamentos necesarios e inolvidables para programar en JavaScript.

3.2.1 CONCEPTOS FUNDAMENTALES

3.2.1.1 ¿JAVASCRIPT ES UN LENGUAJE INTERPRETADO?

Desde hace muchos años se ha hecho una división de los distintos lenguajes de programación entre los que son **compilados** y los que son **interpretados.**

Antes de la década de los noventa del siglo XX, la mayoría de lenguajes que florecían eran compilados. La razón es que el código de un lenguaje compilado se convierte a código máquina, analizando todas las líneas del mismo antes de empezar a convertir en binario el código. De esta forma, se consigue un código más eficiente. Si hay un error, por ejemplo, en la línea de código número 200, en el proceso de compilación se nos indicará el error de esa línea y no veremos el resultado que producen las primeras 199 líneas (que son correctas) si no arreglamos el problema de la línea 200.

En los lenguajes interpretados se realiza la conversión a código máquina línea a línea. Usando el mismo ejemplo, si tenemos un error en la línea 200 veríamos con normalidad el resultado de las primeras 199 líneas. Sería al intentar interpretar la línea 200 cuando se notificaría el error.

Sabiendo que la depuración es más compleja y que el código final de los lenguajes interpretados es menos eficiente (al no optimizar el código a través del análisis de todas las líneas) no parece haber razón alguna para utilizar lenguajes interpretados.

Sin embargo, el código interpretado ofrece ventajas cuando se debe ejecutar en un entorno donde la carga del programa es lenta, como ocurre en Internet (aunque cada vez menos). El hecho de que sea interpretado permite ir ejecutando las primeras líneas que se carguen antes incluso de que se carguen las siguientes, lo cual es mucho más eficiente en el caso de código incrustado en el lado del cliente de las aplicaciones web.

Tradicionalmente, se considera a JavaScript un lenguaje interpretado. Pero la realidad es más compleja. Inicialmente JavaScript solo era apto para realizar acciones no muy complejas, hoy en día sus capacidades no le diferencian de la mayoría de los lenguajes existentes, y eso ha implicado que los navegadores precompilen el código JavaScript para que sea más veloz su rendimiento.

Actualmente, JavaScript utiliza un compilador **JIT** ***(Just In Time)*** gracias a una máquina virtual de JavaScript **(JVM)** que proporcionan los navegadores. Sin extendernos demasiado, diremos que JavaScript es un lenguaje con una compilación semejante a la del lenguaje **Java.** El código de

JavaScript se compila al vuelo (ya que se debe ejecutar lo más rápido posible) para que se pueda servir lo más rápido posible, pero habiendo optimizado el código.

No se consigue la rapidez de lenguajes totalmente compilados como el lenguaje C, pero tampoco adolece de la falta de eficiencia del lenguaje JavaScript original. Al final, la velocidad de este proceso de conversión de código depende de los trucos y habilidades que proporcionan los motores de ejecución de JavaScript. El famoso V8 del navegador Chrome de Google, consiguió acelerar la ejecución de código JavaScript de forma notable.

En todo caso, sí conviene pensar que JavaScript tiene mucho todavía de lenguaje interpretado, y que no todos los errores que cometamos se marcarán al comenzar a ejecutar nuestro código.

3.2.1.2 SENTENCIA O INSTRUCCIÓN

Como todos los lenguajes de programación, el código se divide en **sentencias** o **instrucciones.** Se trata de la estructura mínima del programa. Una sentencia es una estructura básica de código que es capaz de realizar una acción simple. Podemos entender un programa como una sucesión de sentencias o instrucciones.

En JavaScript las instrucciones simples se pueden finalizar con el símbolo de punto y coma (;). Sin embargo, no es obligatorio y podemos escribir instrucciones sin indicar el final. Nuestra recomendación es finalizar siempre con punto y coma, ya que eso posibilita un mejor aprendizaje del lenguaje si entendemos cuándo debemos indicar el punto y coma.

Un ejemplo de sentencia es:

```
var nombre="Jorge";
```

Lo que hace es asignar el texto ***Jorge*** a una variable llamada ***nombre.***

3.2.13 IDENTIFICADORES Y PALABRAS RESERVADAS

La palabra **var** del código anterior es lo que se conoce como **palabra reservada.** Un término especial que, dentro del lenguaje, tiene un significado especial.

En cualquier lenguaje de programación se puede dar nombre a elementos que el programador necesita en su código. Por ejemplo, en el código anterior ***nombre*** es como hemos decidido denominar a la variables que almacena la palabra ***Jorge,*** que se entrecomilla porque es un texto. No podríamos haber llamado a esa variable **var,** como tampoco **if** o **try,** porque todas ellas (y hay muchas más) son palabras reservadas del lenguaje.

Es decir, las palabras reservadas son términos que ofrece el lenguaje para poder programar. Mientras que los nombres que damos a variables ***y*** otros elementos del lenguaje se conocen como **identificadores,** ya que su labor es precisamente identificar de forma inequívoca a un elemento del lenguaje.

Los identificadores deben de cumplir las siguientes reglas:

- Deben comenzar por una letra, un guión bajo (_) o el símbolo dólar ($)
- Después se pueden usar letras, números o el guión bajo

- Se pueden usar letras Unicode, aunque estén fuera del código ASC11, como ***á*** o ***é.***
- Se distingue entre mayúsculas y minúsculas. ***salarioFinal*** es un identificador distinto de ***SalarioFinal.***

3.2.1.4 REGLAS FUNDAMENTALES

- JavaScript es un **lenguaje sensible a mayúsculas.** Es decir, se distingue entre mayúsculas y minúsculas. No es lo mismo **var** que **VAR,** por eso este código produce un error:

```
VAR x=7; //¡Error!
```

- Los **comentarios de una línea** comienzan con los símbolos //. Un comentario es un texto explicativo que no se tendrá en cuenta al interpretar el código, y por ello, se usa para documentar el código. De esa forma podemos explicar mejor lo que hacemos en el código especialmente si trabajamos en equipo, pero también incluso para nosotros mismos ya que, con el paso del tiempo, se nos olvidará el funcionamiento de muchas de las instrucciones que escribamos:

```
var tam=nombre.length(); //tam recoge el tamaño del nombre
```

• Los **comentarios de varias líneas** comienzan con los símbolos /* y terminan con */

```
/* Comienzo de la función principal
     de proceso de datos */
```

■ Se usan cuando necesitamos escribir mucho texto explicativo. Aunque están pensados para comentarios largos, se pueden utilizar en pequeños bloques de una línea:

```
var n = /^Ejemplo*/ 17;
```

La variable identificada como ***n*** valdrá ***17.***

- Se pueden agrupar sentencias en **bloques** de sentencias. Los bloques son sentencias contenidas entre llaves (símbolos { y }). Normalmente los bloques forman parte de sentencias complejas. Ejemplo:

```
var x=9;
if(x==9){
     x++;
     consolé.log(x);
}
```

En este ejemplo, la sentencia if contiene un bloque de sentencias, las cuales se ejecutan si se cumple la condición x==9 (x igual a 9). En este caso, si probamos el código veremos que, por consola, se escribe el número 10.

3.3 VARIABLES Y TIPOS DE DATOS

3.3.1 VALORES

Los programas en JavaScript utilizan ***valores.*** Los valores son datos que el programa necesita almacenar, manipular o mostrar.

En JavaScript hay tres tipos de valores básicos: **Textos** (llamados ***Strings),*** **Números** y **Valores booleanos.** A estos tres tipos hay que añadir el tipo complejo **Objeto** y los valores **indefinido** y **nulo.**

Existe un operador llamado **typeof** que permite conocer el tipo JavaScript de un valor concreto. Aunque más adelante explicamos cómo escribir valores de todo tipo, vamos a ver ejemplos de uso de estos valores. Para ello vamos a invocar a **node** desde la línea de comandos, y pediremos que nos escriba los tipos correspondientes a varios valores:

```
$ node
>  typeof 12
' number'
>  typeof 12.3
' number'
>  typeof(12*3)
' number'
>  typeof "Hola”
'string'
>  typeof 'Hola'
'string'
>  typeof Hola
'undefmed'
>  typeof (1/0)
'number'
>  typeof nuil
' object'
>  typeof {nombre:”Jorge”}
' object'
>  typeof [1,2,3,4,5]
' object'
>  typeof true
' boolean'
>  typeof (12>3)
' boolean'
```

Explicamos algunos detalles:

- El operador **typeof** valora la expresión que escribamos a su derecha. Si es compleja, se debe indicar entre paréntesis para asegurar que se evalúa el conjunto completo. La

respuesta de este operador es un texto que indica el tipo de datos al que pertenece 1 expresión.

- Todos los números tienen el mismo tipo ***(number)*** sin importar si son decimales o no. Incluso se devuelve el tipo number para la expresión 1/0 que realmente no se puede evaluar. Sin embargo, se considera que uno dividido entre cero produce infinito, y en JavaScript, el infinito es un número válido por eso typeof (1/0) devuelve ***number.***

- Los textos se entrecomillan. Cualquier expresión entrecomillada (sin importar si usamos comillas simples o dobles) se entiende que es un texto **(string).**

- El tipo ***undefined*** se emplea cuando JavaScript no puede evaluar una expresión. La expresión **typeof hola** provoca un resultado indefinido porque se busca la variable hola (al no ser un texto entrecomillado) y, como no existe, se decide que es una expresión indefinida.

- Los tipos complejos ***(object)*** se utilizan mucho. Incluso el valor **nuil** se considera un objeto. Más adelante explicaremos el funcionamiento de este tipo de datos.

3.3.2 DECLARACIÓN DE VARIABLES

En los lenguajes de programación, los valores se almacenan en variables. Las variables son uno de los elementos fundamentales en cualquier lenguaje de programación.

Cada variable tiene un identificador. A ese identificador se le asocian los valores que deseemos. Lo interesante es que podemos operar con las variables para realizar los cálculos que deseemos.

En JavaScript las variables deben de ser declaradas antes de poderse usar. Declarar una variable es asignarla un identificador. Tradicionalmente la palabra reservada **var** es la que se utiliza para declarar una variable:

```
var x;
```

De esta forma estamos avisando de que se podrá utilizar, a partir de ese momento, en el código una variable que se llama ***x.***

Es muy habitual asignar un valor al declarar una variable:

```
var x=9;
```

Con esto declaramos la existencia de una variable que se llama ***x.*** Además, estamos indicando que, inicialmente, vale nueve.

Las variables solo se declaran una vez. Después la variable se usa con normalidad:

```
var x = 9;
x = 25;
console.log(x); //escribe 25
```

3.3.2.1 DIFERENCIA ENTRE **LET** Y **VAR**

Desde la versión 6 del estándar **ECMAScript** (conocido como **ES2O15** o **ES6),** se permite el uso de las palabras reservadas **let** y **const** para declarar variables (además de **var).**

let se utiliza para declarar variables de forma más local que var. Para entender la diferencia veamos este código:

```
{
      let x=9;
}
consolé.log(x);
```

Si ejecutamos este código, aparecerá este error:

```
ReferenceError: x is not deñned
```

La razón es que la variable ***x*** se ha definido dentro de un bloque mediante la palabra clave **let** y no se puede utilizar fuera. Digamos que solo existe esa variable en el ámbito del bloque en el que se declaró.

Sin embargo, este otro código funciona sin errores:

```
{
      var x=9;
}
consolé.log(x);
```

Ahora aparece el número 9 como resultado de ejecutar el código.

Por lo tanto, la diferencia es que **let** restringe el uso de la variable al bloque concreto en el que se encuentra. En el caso de **var,** el ámbito de aplicación es mayor, la variable sobrevive fuera del bloque.

Realmente las variables declaradas con **var** también tienen restricciones en cuanto a su ámbito de uso. Vamos a ver este código:

```
function f(){
      var x=9;
}
consolé.log(x);
```

Al ejecutar el código, volvemos a producir un error de variable indefinida. Aunque las funciones serán objeto de nuestro estudio más adelante, podemos entender que las funciones también utilizan bloques de código (como siempre, delimitados entre llaves). La diferencia, ahora, es que esos bloques son más restrictivos: incluso las variables definidas por **var** no pueden utilizarse fuera del bloque definido por una función.

Siendo más formales, se dice que el ámbito de uso de las variables declaradas con **let** es el bloque, y el ámbito de uso de las variables declaradas con **var** es el **contexto de ejecución** en el

que se encuentra la palabra var. Este último ámbito puede ser **global,** si se declara fuera de toda función, y abarca todo el archivo, o puede ser relativo a una función si se declara dentro de ella.

Este otro código nos ayuda a reflexionar sobre la idea:

```
var x=7; //x se declara en ámbito global
function f(){
        var x=9; //x se de declara en el ámbito de la función
}
consolé.log(x);
```

El resultado de este código es la escritura por pantalla del número 7. Realmente hay dos variables llamadas ***x,*** la primera tiene ámbito global y la segunda se restringe a la función. Por eso, como consolé, log(x) está fuera de la función, se usa la x global, que vale siete.

33.2.2 DECLARACIÓN CON **CONST**

Este término se usa para definir constantes. Funciona bajo las mismas condiciones de ámbito que **let.** La diferencia es que una variable declarada con **const** no puede modificar su valor:

```
const PI=3.14592;
PI=9;
```

La ejecución de este código produce un error:

```
TypeError: Assignment to constant variable
```

No podemos modificar el valor de una variable declarada mediante la palabra **const.**

Aunque se pueden usar tanto minúsculas como mayúsculas para declarar constantes puras, es recomendable usar mayúsculas, ya que así las distinguimos más fácilmente de las variables.

3.3.3 TIPOS DE DATOS PRIMITIVOS

33.3.1 NÚMEROS

A diferencia de otros lenguajes (como **C, C++** o **Java),** JavaScript utiliza un solo tipo de datos para los números, sin importar si el número tiene decimales o no.

Los números se escriben tal cual, sin comillas ni símbolos extra. Si tienen decimales, se usa el punto como separador decimal.

```
let entero=1980;
let decimal=0.21;
```

El hecho de no diferenciar enteros de números decimales, hace que todos los números ocupen realmente el mismo espacio en memoria: **64 bits.**

Otras formas de indicar números

JavaScript admite formatos especiales numéricos. Por ejemplo, admite la notación científica:

```
const AVOGADRO=6.022e+23;
```

Lo cual significa: $6{,}022\text{-}10^{23}$

La notación científica es ideal para indicar números muy grandes o muy pequeños.

Por otro lado, no solo se aceptan números decimales, también se aceptan números hexadecimales si se les antepone el prefijo **Ox.**

```
let hexa=0xAB12;
consolé.log(hexa);
```

El resultado es el **43794,** resultado de convertir a decimal el número hexadecimal **AB12.**

También es posible usar números octales. Éstos usan el cero como prefijo seguido de la letra o minúscula:

```
let octal=0o27652;
consolé.log(octal);
```

Ahora aparece el número 12202, resultado de convertir a decimal el octal 27652.

Por si fuera poco, podemos también indicar números en binario mediante el prefijo **0b** ***(cero*** seguido de ***b).***

```
let binario=0bl0111011;
consolé.log(binario);
```

El número binario se convertirá al decimal correspondiente (187).

Números especiales

JavaScript permite utilizar el número infinito **(lnfinity).** Ejemplo:

```
var x=l/0;
consolé.log(x);
```

Por pantalla aparece **lnfinity,** ya que cualquier número dividido por cero provoca ese resultado. El término lnfinity es reconocido de forma independiente, así:

```
var y=Infinity;
consolé.log(y+l);
```

Este código también produce la escritura de ***Infinity,*** porque sumar uno al número infinito, sigue resultando el propio infinito. Lo único que demuestran estos resultados es que el infinito se maneja perfectamente por parte de JavaScript.

Hay otro término especial. Veamos un ejemplo para entenderlo:

```
var x="Hola" * 3;
consolé.log(x);
```

Lo lógico sería pensar que la primera línea provoca un error de ejecución de código, al intentar operar de forma numérica con un texto. Pero no es así. Lo que resulta de ese código es que, por pantalla, aparezca el texto **NaN.** Es más, podemos afirmar que **x** vale **NaN.**

NaN es el acrónimo de ***Not a Number (No es un Número),*** que se provoca cuando una expresión intenta operar de forma numérica un valor que no es un número. Es muy interesante que JavaScript aporte este término directamente, ya que facilita mucho la validación de errores, como veremos más adelante.

Las indeterminaciones matemáticas también provocan este valor. Por ejemplo, la expresión **0/0** (cero partido por cero) o **Infinity - Infinity.** Estas habilidades matemáticas de JavaScript permiten resolver de forma muy notable problemas complejos sobre números.

3.33.2 STRINGS

Delimitar strings

Ya hemos comentado que los strings son cadenas de caracteres. En definitiva, textos. Desde su primera versión, JavaScript permite delimitar textos con comillas dobles o simples. Así, estas expresiones son equivalentes:

```
texto="Hola a todo el mundo";
texto='Hola a todo el mundo';
```

Y esto permite que dentro del texto delimitado puedan aparecer comillas dobles o simples.

```
frase=”Mi apellido es O’Donnell”;
frase2= José se acercó y dijo "Hola" ;
```

Desde la versión 6 se permiten también las comillas invertidas (conocidas como ***backticks*** en inglés):

```
texto= Hola a todo el mundo';
```

Uso de plantillas de string

La tercera forma de delimitar textos, con el uso de comillas invertidas, permite, de forma muy sencilla, colocar expresiones dentro de las comillas. Vamos a explicar la idea:

```
var nombre='Jorge";
consolé.log("Me llamo nombre");
```

Por pantalla, la ejecución del código escribe ***Me llamo nombre.*** Pero, es posible que pretendiéramos escribir ***Me llamo Jorge,*** ya que la variable ***nombre*** contiene el texto ***Jorge.*** Pero, entre comillas, el texto se toma de forma literal. Hay una solución gracias al operador de concatenación (+).

```
var nombre="Jorge";
consolé.log("Me llamo " + nombre);
```

Ahora sí se escribe ***Me llamo Jorge.***

Aunque el operador de concatenación es muy interesante, si el texto a escribir es complejo, es pesado tener que encadenar expresiones con el operador +.

Las plantillas de string aportan otra solución. Se llaman plantillas de string ***(String templates)*** a los textos delimitados con comillas invertidas. La cuestión es que se admite incluir el signo $ (dólar) dentro del texto delimitado y después, entre llaves, indicar una expresión JavaScript. El texto dentro de las llaves no se tomará como texto literal, se interpretará literalmente como si estuviera fuera de las comillas. Ejemplo:

```
var nombre="Jorge";
consolé.log('Me llamo ${nombre} );
```

Nuevamente veremos ***Me llamo Jorge.***

Podemos incluso indicar expresiones más complejas:

```
consolé.log('Cada día hay ${24*60*60} segundos');
```

Escribe:

```
Cada día hay 86400 segundos
```

Secuencias de escape

Hay caracteres de no se pueden escribir y que, sin embargo, son muy necesarios. Es el caso del salto de párrafo (relacionado con la tecla Intro) o el fabulador. Se consiguen escribir esos caracteres usando un código especial que comienza con la barra inclinada a la izquierda ***(backslash)*** seguida del código concreto. Por ejemplo:

```
let texto=”Una línea\nOtra línea”;
```

Por consola, aparece el texto ***Una línea*** en la primera línea y debajo el texto ***Otra línea.***

Algunas de las secuencias más utilizadas son:

SECUENCIA	SIGNIFICADO
\n	Salto de línea
\t	Tabulador
\r	Retorno de carro
\f	Salto de página
\v	Tabulador vertical
\”	Comillas dobles
\'	Comillas simples
\b	Retroceso
\\	El propio carácter ***backslash***

Ejemplo de uso de comillas usando secuencias de escape:

```
let texto=”Paseando por la calle O’Donell dijo \" ¡Qué calor!\"";
```

Es necesario usar las comillas dobles del texto ***"¡Qué calor!"*** como secuencia de escape, para distinguirlas de las comillas dobles que sirven para delimitar el texto.

Es posible también usar secuencias de escape para poder almacenar en strings caracteres Unicode. Con esa finalidad se utiliza el código \u seguido de cuatro cifras hexadecimales. Ejemplo:

```
consolé.log( \u2A1D ); //escribe el símbolo: XI
```

En el ejemplo anterior hemos conseguido escribir el símbolo correspondiente a la operación ***JO1N*** de las bases de datos relaciónales: X

La tabla Unicode de caracteres tiene incluso más símbolos que los que pueden mostrarse con cuatro cifras hexadecimales (16 bits), pueden usar incluso el doble de bits. Para este tipo de símbolos se usa el formato: ***\u{código}*** siendo el código un número hexadecimal de hasta ocho cifras (32 bits). Ejemplo:

```
consolé.log("\u[lF48B}");
```

Escribe el símbolo 0, que es el clásico emoticono de marca de labios.

33.3.3 BOOLEANOS

Los valores booleanos solo pueden tomar los valores **true** (verdadero) o **false** (falso). Estos valores son palabras reservadas que se pueden asignar directamente:

```
let b=true;
```

Sin embargo, suele ser más habitual que esos valores se produzcan al evaluar una expresión lógica:

```
let x=9;
let y=10;
let b=(x>y);          //b vale false porque x no es mayor que y
```

Incluso podemos decidir que toda expresión y valor en JavaScript se asocia con valores verdaderos o falsos. Para poder comprobar que esto es cierto, podemos usar la función ***Boolean*** (escrita así con la primera letra mayúscula) que escribe el valor booleano equivalente para cualquier valor:

```
consolé.log(Boolean(true));                 //Escribe      true
consolé.log(Boolean(1));                    //Escribe      true
consolé.log(Boolean(0));                    //Escribe      false
consolé.log(Boolean( 'Hola"));              //Escribe      true
consolé.log(Boolean( ));       ""           //Escribe      false
consolé.log(Boolean(NaN));                  //Escribe      false
consolé.log(Boolean(undefined));                    //Escribe false
consolé.log(Boolean(Infinity));                     //Escribe true
consolé.log(Boolean(nul1)); //Escribe false
```

En realidad, solo los textos vacíos (comillas sin contenido), el valor cero, el valor false, el valor **NaN,** el valor **undefined** y el valor nulo **(nuil),** se consideran falsos.

Este hecho se puede utilizar para poder evaluar de forma sencilla valores en instrucciones de control como, por ejemplo la sentencia if (Véase 3.7.3 ***"Instrucción if",*** en la página 93).

3.4 OPERADORES

3.4.1 ARITMÉTICOS

Los operadores aritméticos permiten realizar operaciones matemáticas sobre los números. Son los siguientes:

OPERADOR	SIGNIFICADO	EJEMPLO
+	Suma	consolé.log(5 + 4.5); //Escribe 9.5
-	Resta	consolé.log(5 - 4.5); //Escribe 0.5
*	Multiplicación	consolé.log(5 * 4.5); //Escribe 22.5
/	División	consolé.log(7 / 3); //Escribe 2.333333
%	Resto	consolé.log(7 *% 3);* //Escribe 1
**	Exponente	consolé.log(3 ** 2); //Escribe 9

El exponente (**) se añadió en la versión 2017 del estándar, por lo que puede que algunos navegadores no soporten su uso.

3.4.2 RELACIONALES

Los operadores relaciónales permiten comparar expresiones. El resultado de la operación es siempre un valor booleano. Son:

OPERADOR	SIGNIFICADO	EJEMPLOS
>	Mayor	consolé.log(5 > 4); //Escribe true consolé.log(5 > 5); //Escribe false
<	Menor	consolé.log(5 < 4); //Escribe false consolé.log(5 < 5); //Escribe false
>=	Mayor o igual	consolé.log(5 >= 4); //Escribe true consolé.log(5 >= 5); //Escribe true
<=	Menor o igual	consolé.log(5 <= 4); //Escribe false consolé.log(5 <= 5); //Escribe true
==	Igual	consolé.log(5 == 4); //Escribe false consolé.log(5 == 5); //Escribe true
!=	Distinto	consolé.log(5 != 4); //Escribe true consolé.log(5 != 5); //Escribe false

Un error muy habitual es confundir el operador de asignación (=) con el operador de valoración de igualdad (==). El primero permite asignar un valor a una variable, el segundo compara y devuelve verdadero o falso en función de si se cumple o no la igualdad. Veamos este ejemplo:

```
consolé.log(5=4);
```

Esa expresión provoca un error de ***ReferenceError: Invalid left-hand side in assignment,*** referido a que el lado izquierdo de la asignación (el número 5) no permite realizar una asignación. Sin embargo:

```
consolé.log(5==4);
```

Ahora se mostrará el valor false porque 5 no es igual a 4.

Podemos comparar strings:

```
consolé.log("saco">"saca");
```

Escribirá **true,** porque, en orden alfabético, ***saco*** es mayor que ***saca.*** Sin embargo esta expresión:

```
consolé.log("Saco' >"saca");
```

Escribe, para nuestra sorpresa, **false.** El cambio es que, ahora, hemos usado la "S" mayúscula y las mayúsculas son menores que las minúsculas según el código que se asigna en la tabla Unicode. El mismo problema ocurre con algunos símbolos:

```
consolé.log("ñ' <"o"); //Escribe false
```

A la ***eñe*** se le asigna un código mucho mayor que el que se le asigna a la letra **o,** lo que invalida estos operadores para ser utilizados con textos en español. Como veremos más adelante, la comparación correcta se debe realizar con el método **localCompare** (véase 4.1.2.1 ***"Método localeCompare",*** en la página 121).

3.4.2.1 COMPARACIONES ESTRICTAS

JavaScript posee un operador de comparación estricta que utiliza tres signos de igual (===). La razón procede de la capacidad de JavaScript de comparar datos de diferente tipo:

```
consolé.log("2ⁿ==2);
```

Esa expresión escribe ***true,*** porque aunque "2" es un texto, JavaScript puede comparar de forma numérica el contenido de ese texto. Sin embargo:

```
consolé.log("true'==true); //Escribe false
```

No funciona con todo tipo de datos, el texto ***true*** no se puede convertir a su equivalente booleano. Pero hay más sorpresas:

```
consolé.log(l==true); //Escribe true
```

Estamos comparando un número con un valor booleano, el resultado es true porque se considera que 1 es perfectamente convertible al valor true. Algo que ocurre con el cero también:

```
consolé.log(0==false); //Escribe true
```

Pero no con otros números:

```
consolé.log(2==true); //Escribe false
consolé.log(2==false); //También escribe false
```

Se debe esta situación a compatibilizar el código con la forma de usar true y false con uno y cero en algunos lenguajes (como el lenguaje C). Incluso funciona esta comparación:

```
consolé.log("l"==true); //Escribe true
```

Y también comparaciones curiosas como:

```
consolé.log(undefined==null); //Escribe true
```

Pero no:

```
consolé. log(únelefined==false); //Escribe false
```

Dejando de lado tantas posibilidades, hay que tener en cuenta que en JavaScript es muy habitual que recojamos números que el usuario escribe y estos números se reciben como strings. Por eso viene bien que esta expresión devuelva verdadero:

```
consolé.log("2"==2); //Escribe true
```

Pero puede ser que necesitemos comparar de forma más estricta, y eso es lo que consigue el operador ===. Este operador compara el valor y el tipo de dos expresiones, ambas condiciones se tienen que coincidir para que el operador devuelva true:

```
consolé.log( ===2);"          //Escribe false
consolé.log(l+l===2);         //Escribe true
```

1+1 suma dos números, por eso el resultado es el número 2. Sin embargo "2" en realidad es un texto (un string) por eso, al compararlo con el número 2, devuelve false.

Hay también un operador estricto para valorar si una expresión es diferente de otra. Se trata del operador !==

```
consolé.log(      "2"!=2);      //Escribe      false
consolé.log( !==2); //Escribe true
```

La primera sentencia escribe false porque considera iguales las expresiones "2" y 2. Sin embargo, el operador de diferencia estricta escribe true porque entiende que el dos no es igual como texto que como número.

3.4.3 OPERADOR DE ENCADENAMIENTO

El signo más (+) en JavaScript tiene dos posibilidades. Ya hemos visto que sirve, como es lógico, para sumar números. Sin embargo, también sirve para unir textos en una operación que se conoce como encadenar. Ejemplo:

```
let hoy="Hoy es          ";
let          dia='Viernes";
consolé.log(hoy + dia);
```

Por pantalla aparece ***Hoy es viernes.*** El signo más, en este caso, sirve para unir los textos. El único problema, como veremos más adelante, es tener cuidado de delimitar muy bien cuándo sumamos y cuándo encadenamos.

3.4.4 OPERADORES LÓGICOS

Las expresiones lógicas permiten valorar condiciones para tomar decisiones en nuestro programa. Los operadores lógicos permiten poder utilizar expresiones lógicas complejas. Estos operadores son:

OPERADOR	SIGNIFICADO
f	NOT (no)
&&	AND ("y" lógico)
ll	OR ("o" lógico)

El operador AND (&&) permite comparar dos expresiones lógicas. El resultado será verdadero si las dos expresiones que se comparan lo son:

```
let x=9;
let y=10;
consolé.log(x==9       && y==10);    //true
```

Este código escribe por consola el valor **true,** ya que se cumple que ***x*** vale 9 y que **y** vale 10. Si cualquiera de esas expresiones cambiara el resultado sería falso:

```
let x=9;
let y=12;
consolé.log(x==9       && y==10);    //false
```

El operador OR (||) también compara dos expresiones pero, en este caso, devuelve true si cualquiera de las dos expresiones es verdadera:

```
let x=9;
let y=12;
consolé.log(x==9 || y==10); //true
```

Finalmente, el operador NOT (!) niega la expresión que tenga a su derecha: si es falsa devuelve true y si es verdadera, devuelve false:

```
let x=9;
let y=12;
consolé.log(!(x==9 || y==10)); //false
```

Hay que tener cuidado con este operador, ya que tiene más prioridad que los operadores AND y OR, por lo que si en el código anterior retiramos los paréntesis:

```
let x=ll
let y=12;
consolé.log(!x==9 || y==12); //true
```

La expresión, en este caso, x==9 devolvería false, pero como está matizada con el operador **NOT,** devuelve ***true,*** luego se valor la expresión y==12 (que es verdadera) y luego se une ambas con el operador OR, que resulta otra vez ***true.*** Todo cambiaría con un paréntesis:

```
let x=ll;
let y=12;
consolé.log(!(x==9 II y==12)); //false
```

Ahora se evalúa primero x==9 que es falso, y luego y==12 que es verdadero. Se unen ambas expresiones con el operador OR (||) que devolverá true porque una de las dos es verdadera. Pero al final (gracias al paréntesis) se invierte este resultado por el uso de NOT, lo que provocará un resultado falso.

Un hecho interesante del operador AND es que la segunda expresión solo se evalúa si la primera es verdadera, ya que si la primera es falsa ya sabemos que el resultado será falso. Es decir:

```
let edad=17;
let conduce=(edad>=18          &&  carnet==true);
```

La variable ***conduce*** será falsa porque la ***edad*** (en la línea de arriba) es de 17. Es más, a lo mejor ni hemos declarado la variable ***carnet,*** lo que, en principio, provocaría un error por indefinición de variable. Pero la realidad es que no se va a evaluar la segunda expresión porque la edad no es mayor o igual a 18, y sabemos que el resultado de la expresión es falso independientemente del valor de la segunda parte de la expresión.

3.4.5 OPERADORES DE BIT

La realidad de la computación es que solo maneja números binarios. Por lo que, cualquier número, internamente, se almacena de forma binaria. JavaScript aporta operadores para trabajar a nivel de BIT.

OPERADOR	SIGNIFICADO
~	NOT
&	AND
1	OR
A	XOR
»	Desplazamiento derecho
«	Desplazamiento izquierdo

Empezaremos explicando el operador NOT:

```
consolé.log(~25); //Escribe -26
```

En binario (con 32 bits), el 25 se escribe: 00000000000000000000000000011001

El operador NOT invierte los números: 11111111111111111111111100110 y eso, en decimal, se interpreta como -26 (ya se usa complemento a dos como representación binaria).

Por otro lado:

```
consolé.log(26          & 17); //Escribe 16
```

Veamos por qué se escribe 16:

```
26 binario:                 00000000000000000000000011010
17 binario:                 00000000000000000000000010001
--------------------------------------------------------
Resultado AND:              00000000000000000000000010000          (16 en decimal)
```

El operador AND de bits opera de manera individual con cada bit de modo que el resultado es 1 si ambos bits valen 1.

Sin embargo:

```
consolé.log(26          1 17); //Escribe 27
```

La razón de ese resultado:

```
26 binario:                 00000000000000000000000011010
17 binario:                 00000000000000000000000010001
--------------------------------------------------------
Resultado OR:               00000000000000000000000011011          (27 en decimal)
```

OR opera con cada bit y asigna un uno en cada bit como resultado si cualquiera de los bits con los que opera es un uno.

En el caso del OR exclusivo (XOR):

```
consolé.log(26 A 17); //Escribe 11
```

El razonamiento es:

```
26 binario:                 00000000000000000000000011010
17 binario:                 00000000000000000000000010001
--------------------------------------------------------
Resultado XOR: 00000000000000000000000001011                    (11 en decimal)
```

XOR retorna en cada bit un uno si uno de los dos bits es un uno, pero no hay dos unos a la vez.

En la actualidad, los operadores de desplazamiento se utilizan menos. EL operador « añade el número de bits con valor cero a la derecha que se indiquen. Así:

```
consolé.log(9<<3); //Escribe 72
```

El funcionamiento es este:

```
9 binario: 00000000000000000000000001001
--------------------------------------------------------
9 << 3:             00000000000000000000001001000          (72 en decimal)
```

Se han añadido tres ceros a la derecha y eso es como multiplicar tres veces por 3 al número.

El desplazamiento derecho funciona al revés:

```
consolé.log(9>>3); //Escribe 1
```

Ya que:

```
9 binario: 00000000000000000000000000001001
————————————————————————————————
9 >> 3:          00000000000000000000000000000100T          (1 en decimal)
```

En la práctica el resultado es equivalente a dividir tres veces por dos.

En la actualidad los operadores de desplazamiento están en desuso.

3.4.6 OPERADORES DE ASIGNACIÓN

3.4.6.1 INCREMENTOS Y DECREMENTOS

En JavaScript (al igual que ocurre con otros lenguajes), es muy habitual tener que realizar esta operación:

```
x=x+1;
```

Por ello, se dispone de esta forma más rápida:

```
x++;
```

De tal manera que conseguimos rápidamente este resultado:

```
let x=8;
x++;
consolé.log(x);          //escribe 9
```

El operador ++ se llama incremento. También disponemos del operador -- para el decremento:

```
let x=8;
x—;
consolé.log(x); //escribe 7
```

Estos operadores se pueden colocar por detrás de la variable (x++) o por delante (++x).

```
let valor=50, incremento, decremento;
//Ejemplo de postincremento, primero se asigna y luego se
//incrementa la variable
incremento=valor++; //incremento vale 50 y valor vale 51
de□remento=valor—; //decremento vale 51, valor vale 50
valor=50;
//Ejemplo de preincremento, primero se incrementa
//y luego se asigna la variable
incremento=++valor; //incremento vale 51 y valor también
decremento=—valor;          //decremento y valorl valen 50
```

En el ejemplo se puede observar la diferencia. En el caso del **postincremento (x++)** o **post-decremento (x—)** si es parte de otra expresión el incremento o decremento se hace después. Es decir, en la expresión y=x++ la variable ***y*** tomaría el valor que tenga ***x*** (sin incrementar) y luego se incrementaría x, pero la variable ***"y",*** ya no se modifica.

En el caso de los **preincrementos (++x)** o **predecrementos (—x)** siempre se realiza primero la operación de incrementar o decrementar. De ahí los resultados en el código anterior.

3.4.6.2 OTROS OPERADORES DE ASIGNACIÓN

Hay otros operadores que permiten abreviar las operaciones de asignación:

```
var x=18;
x+=9; //x vale ahora 27
```

La expresión **x+=9** sirve para abreviar la expresión **x=x+9.** Es un operador de asignación y suma. La lista completa de operadores de asignación es:

OPERADOR	SIGNIFICADO	EJEMPLO	EQUIVALENTE A...
+=	Suma y asignación	**x+=5**	**x=x+5**
-=	Resta y asignación	**x-=5**	**x=x-5**
*=	Multiplicación y asignación	**x*=5**	**x=x*5**
/=	División y asignación	**x/=5**	**x=x/5**
%=	Resto y asignación	**x%=5**	**x=x%5**
**=	Exponente y asignación	**x**=2**	**x=x**2**
&=	AND de bit y asignación	**x&=65**	**x=x&65**
1=	OR de bit y asignación	**x | =65**	**x=x|65**
A=	XOR de bit y asignación	**x^=65**	**x=x^65**
>>=	Desplazamiento derecho y asignación	**x>>=3**	**x=x> >3**
<<=	Desplazamiento izquierdo y asignación	**x< <=3**	**x=x< <3**

3.4.63 OPERADOR CONDICIONAL

Se trata de un operador capaz de evaluar una expresión booleana, y si ésta es verdadera dar como resultado un valor y si es falsa dar otro. A este operador se le conoce también como ***inline if*** porque su labor se asemeja mucho a la que realiza la estructura if.

De hecho, este operador se suele utilizar cuando deseamos hacer evaluaciones rápidas que normalmente requerirían de una instrucción **if.**

La sintaxis de uso de este operador es la siguiente:

```
condición ? valorSiVerdadera : valorSiFalsa
```

Ejemplo de uso:

```
let tipo = (numero%2==0) ? "par" : "impar";
```

Esta sentencia asigna el valor adulto a la variable estado si edad es mayor o igual a 18, de lo contrario le asigna el valor menor.

3.5 CONVERSIÓN DE TIPOS

3.5.1 CONVERSIÓN AUTOMÁTICA

JavaScript no es un lenguaje muy ***tipado.*** Lo que significa que, a diferencia de lenguajes más fuertemente tipados como **Java,** no se producen errores con facilidad al mezclar tipos diferentes de datos. El lenguaje JavaScript se basa en buscar la lógica más razonable de cada expresión. Ejemplo:

```
consolé.log(’2' * 3);
```

Esta expresión produciría un error en muchos lenguajes de programación al intentar multiplicar un número por un string. Sin embargo, JavaScript realiza conversiones automáticas de tipos de datos, siempre que sea posible. Gracias a ello, el resultado de esa operación es 6.

Pero observemos este código:

```
consolé.log('2' + 3);
```

Escribirá 23. Ahora tenemos un operador capaz de sumar, cuando opera con números, y de encadenar, cuando opera con strings. En este caso, la conversión automática opera con el número 3, que se convertirá en un string para poder aplicar el encadenamiento. El resultado sería el mismo en este caso:

```
consolé.log(2+'3');
```

En cuando el signo más tiene un string alrededor, JavaScript entenderá que debe encadenar y no sumar.

Por supuesto no todo es convertible:

```
consolé.log('Hola’ * 3);
```

No se produce error, pero se escribe el valor **NaN** porque no se puede conseguir un valor numérico de esa expresión. Pero vemos claramente que JavaScript se esfuerza por conseguir un resultado coherente sin provocar un error. Y por ello, las siguientes expresiones no provocan error:

```
consolé.log(null * 3); //Escribe 0
consolé.log(true * 3); //Escribe 3 (true se asocia al número 1)
consolé.log(false * 3); //Escribe 0 (false se asocia al número cero)
consolé.log(undefined * 3); //Escribe NaN
```

3.5.2 FORZAR CONVERSIONES

Algunas veces deberemos forzar la conversión de los números. En este ejemplo:

```
let x='2';
let y=’3l;
consolé.log(x + y);
```

Como ya hemos visto, se escribe 23 por pantalla. Pero ¿y si quisiéramos sumar de forma numérica? Para eso disponemos de la función **Number:**

```
let x='2';
let y=’3l;
consolé.log(Number(x) + Number(y)); //Ahora escribe 5
```

Por supuesto no todo se puede pasar a número:

```
consolé.log(Number("Hola' )); //Escribe NaN
```

Disponemos de una función para convertir a strings. Se trata de **String.**

```
let x=2;
let y=3;
consolé.log(String(x) + String(y)); //Escribe 23
```

Hay una función muy poderosa llamada **parselnt** para convertir a números enteros. En este sentido es como **Number,** pero con capacidad solo de convertir a números no decimales. Sin embargo, permite cambiar de base y así pasar, por ejemplo, de números binarios a decimales:

```
consolé.log(parselnt("101101",2))
```

Tras el número a convertir (101101) se coloca una coma (para separar el segundo parámetro de la función) y el número 2, indicando que el texto original contiene un número en base dos (binaria).

parselnt además tiene la capacidad de extraer el número con el que comienza un texto (el método **Number** no tiene esta capacidad):

```
consolé.log(parselnt( 22veces',10));
```

Escribe 22 en lugar de **NaN** (que es lo que haría la función Number).

Cuando el número está en base decimal, no hace falta explicitar el número 10:

```
consolé.log(parselnt( 22veces')); //Sigue escribiendo 22
```

parselnt no es capaz de coger decimales:

```
consolé.log(parselnt( 22.5veces")); //Sigue escribiendo 22
```

Pero hay una función llamada **parseFloat** que sí que tiene esta capacidad:

```
consolé.log(parseFloat("22.5veces )); //Ahora escribe 22.5
```

3.5.3 FUNCIÓN ISNAN

Hay problemas en JavaScript para determinar si una expresión no es numérica. Aunque el valor **NaN** es un valor válido en JavaScript, y sabemos que, por ejemplo, "Hola"*4 produce un resultado no numérico. La comparación ("Hola"*4)==NaN produce un resultado falso.

Por ello se utiliza una función llamada **isNaN.** Esta función recibe una expresión y devuelve verdadero si la expresión no es numérica:

```
consolé.log(isNaN(NaN)); //escribe true
consolé.log(isNaN("Hola"*5)); //escribe true
consolé.log(isNaN("3"*5)); //escribe false, "3"*5 retorna 15, un número
```

3.6 CUADROS DE DIÁLOGO DEL NAVEGADOR

Los navegadores permiten utilizar un objeto del sistema llamado **window.** Sin querer entrar (aún) en cómo utilizar los objetos del sistema, en este apartado vamos a aprender a utilizar los métodos que proporciona este objeto para poder comunicarse con el usuario e incluso requerirle información.

Actualmente en una aplicación web profesional, no se usan estas técnicas al ser mucho más eficiente y versátil el uso de controles de formulario y manejo de componentes para informar al usuario y pedirle datos, pero es un método muy fácil para poder empezar a crear pequeños programas en JavaScript y probar las funcionalidades básicas del lenguaje.

3.6.1 MENSAJES DE ALERTA

Los mensajes de alerta son cuadros que muestran información al usuario. Lo que muestran es un texto que aparece en un pequeño cuadro de diálogo:

```
<!DOCTYPE html>
<html lang="es">
<head>
	<meta charset="UTF-8">
	<title>Mensajes de alerta</title>
</head>
<body>
	<script>
		alert("Hola");
	</script>
</body>
</html>
```

El resultado es:



Figura 3.2: Cuadro de alerta en el navegador Google Chrome

Por supuesto, es posible utilizar para el mensaje todo tipo de strings:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Mensajes de alerta</title>
</head>
<body>
        <script>
                let nombre="Jorge";
                alert( Mi nombre es ${nombre } );
        </script>
</body>
</html>
```

Resultado:



Figura 3.3: Cuadro de alerta usando plantillas de strings

3.6.2 CUADROS DE CONFIRMACIÓN

Otro tipo de cuadro interesante es el de confirmación. El método encargado de esta tarea es **confirm.** Al igual que ocurre con **alert,** confirm requiere de un texto que será el mensaje que mostramos al usuario. El cuadro es distinto porque admite aceptar o no el mensaje. Para saber lo que ha elegido el usuario, el resultado de este método es el valor true o false. Ejemplo:

```
<script>
    let resultado=confirm("¿Aceptas nuestras condiciones?");
    consolé.log(resultado);
</script>
```

Se mostrará este cuadro:



Figura 3.4: Cuadro de confirmación

Después se mostrará por consola el valor **true** si el usuario acepta el cuadro y **false** si cancela el mensaje.

Más adelante conoceremos sentencias, como por ejemplo, la sentencia if, que nos permitirán realizar unas acciones u otras en virtud de lo que elija el usuario.

3.6.3 CUADROS DE ENTRADA DE TEXTO

El cuadro más completo es **prompt.** Es un cuadro que muestra un mensaje y, además, recoge lo que el usuario escriba en él (el cuadro muestra un control de texto para que el usuario escriba, si lo desea, en él). Ejemplo:

```
<script>
    let resultado=prompt("Escribe tu nombre");
    alert(Tu nombre es $¡resultado ')
</script>
```

Inicialmente aparecerá este cuadro:



Figura 3.5: Cuadro de entrada de texto. El usuario ha escrito **Andrea**

Después un cuadro de alerta muestra el nombre en el mensaje:



Figura 3.6: Cuadro de alerta mostrando el nombre escrito en el cuadro anterior

El método prompt admite indicar un valor por defecto. Por ejemplo:

```
let resultado=prompt("Escribe tu edad",18);
```

Se mostrará un cuadro en el que, por defecto, ya aparece el valor 18.

Recoger valores del usuario es fundamental para poder programar. Aunque usar este tipo de cuadros, como ya habíamos adelantado, no es el método ideal para recoger valores, nos va a ser muy útil utilizar estos métodos para entender el funcionamiento de las estructuras de control de javaScript.

3.7 CONTROL DEL FLUJO DEL PROGRAMA

3.7.1 INTRODUCCIÓN

Hasta ahora las instrucciones, que hemos visto, son instrucciones que se ejecutan secuencialmente. Es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las instrucciones de control de flujo permiten alterar esta forma de ejecución. El uso de esas instrucciones propicia que haya líneas en el código que se ejecutarán o no dependiendo de una condición. Incluso lograremos que se puedan repetir continuamente una serie de instrucciones mientras se siga cumpliendo la condición que deseemos.

Esa condición se construye utilizando lo que se conoce como **expresión lógica.** Una expresión lógica es cualquier tipo de expresión válida en JavaScript que proporcione un resultado booleano (verdadero o falso). Las expresiones lógicas se construyen usando variables booleanas, o bien escribiendo expresiones que combinen los operadores relaciónales (==, >, <,...) y lógicos (&&,||,!) vistos anteriormente (véase operadores lógicos, página 20).

3.7.2 NÚMEROS ALEATORIOS

Los números aleatorios en JavaScript requieren del uso del objeto **Math** que se explicará más adelante (véase 6.4.1 ***"Objeto Math",*** en la página 217). Su uso permite que las aplicaciones rea-

licen acciones imprevisibles y son fundamentales para programar juegos de todo tipo o simulaciones. En nuestro caso, estimamos conveniente aprender a utilizar números aleatorios en este punto porque facilitan el aprendizaje del lenguaje javaScript.

Para generar números aleatorios en JavaScript se usa el método **random** del objeto **Math** de esta forma:

```
Math.random()
```

Esa expresión da como resultado un número decimal entre cero y uno. Si queremos números más grandes, esta otra expresión:

```
Math.random()*2
```

Obtiene como resultado un número entre el cero y el dos (siendo casi imposible que salga el número 2). Si deseamos un número decimal entre 6 y 10, la expresión correcta es:

```
Math.random()*4+6
```

Se multiplica por el número 4 porque el rango entre 6 y 10 es 4. Se suma el número 6 para que el rango devuelto empiece por el número 6.

Si deseamos números enteros, debemos ayudarnos de la función **parselnt.** Como ya hemos visto, esta función elimina los decimales. Calcular un número aleatorio del 0 al 10 (incluidos ambos) sería:

```
parselnt(Math.random()*11)
```

No multiplicamos por diez, sino por once. La razón es que Math. random( )*10 devuelve números entre cero y diez, pero sin pasar de 10. Es imposible que aparezca el 10. Por ello, hay que sumar uno más para que los números vayan de 0 a 10,999999.... y así al retirar los decimales con **parselnt** aparezca el rango deseado.

Si deseamos números enteros entre el 6 y el 10, la expresión sería:

```
parselnt(Math.random()*5)+6
```

Math. random()*5 devuelve números entre el cero y el cinco (sin llegar a cinco). Usando **parselnt** son enteros del cero al cuatro (al quitar los decimales ya no llegamos a cinco). Si sumamos seis a este resultado, tenemos enteros del seis al diez.

3.7.3 INSTRUCCIÓN IF

3.7.3.1 SENTENCIA CONDICIONAL SIMPLE

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutarán dichas instrucciones.

Sintaxis:

```
if(expresión lógica) {
    instrucciones
    ...
}
```

Las llaves se requieren solo si va a haber varias instrucciones. En otro caso se puede crear el **if** sin llaves, es decir:

```
if(expresión lógica) sentencia;
```

Ejemplo:

```
if(nota>=5){
    consolé.log("Aprobado");
    aprobados++;
}
```

La idea gráfica del funcionamiento del código anterior es la indicada en la Figura 3.7

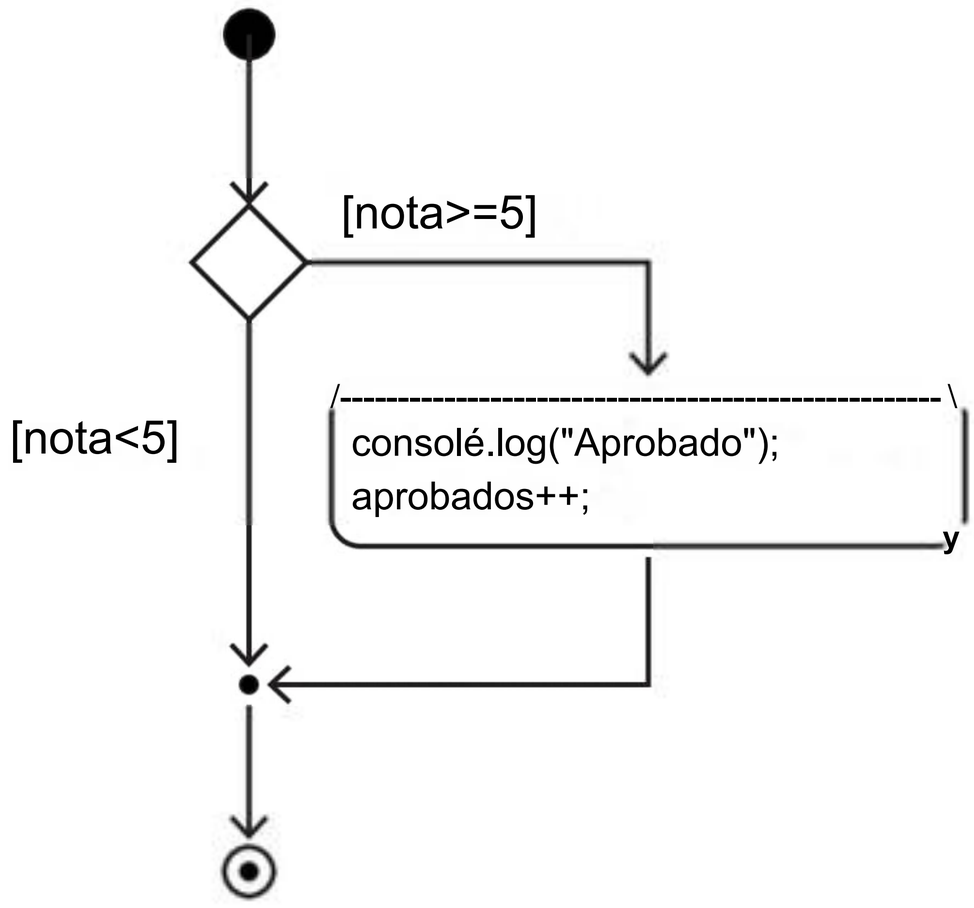


Figura 3.7: Diagrama de actividad de la sentencia **if** simple

3.7.3.2 SENTENCIA CONDICIONAL COMPUESTA

Es igual que la anterior, solo que se añade un apartado else que contiene instrucciones que se ejecutarán si la expresión evaluada por el if es falsa. Sintaxis:

```
if(expresión lógica){
    instrucciones
    ….
}
```

```
else {
   instrucciones
   …
}
```

Como en el caso anterior, las llaves son necesarias sólo si se ejecuta más de una sentencia.

Ejemplo de sentencia **if-else:**

```
i f(nota>=5){
    consolé.log("Aprobado");
    aprobados++;
}
else {
    consolé.log("Suspenso");
    suspensos++;
}
```

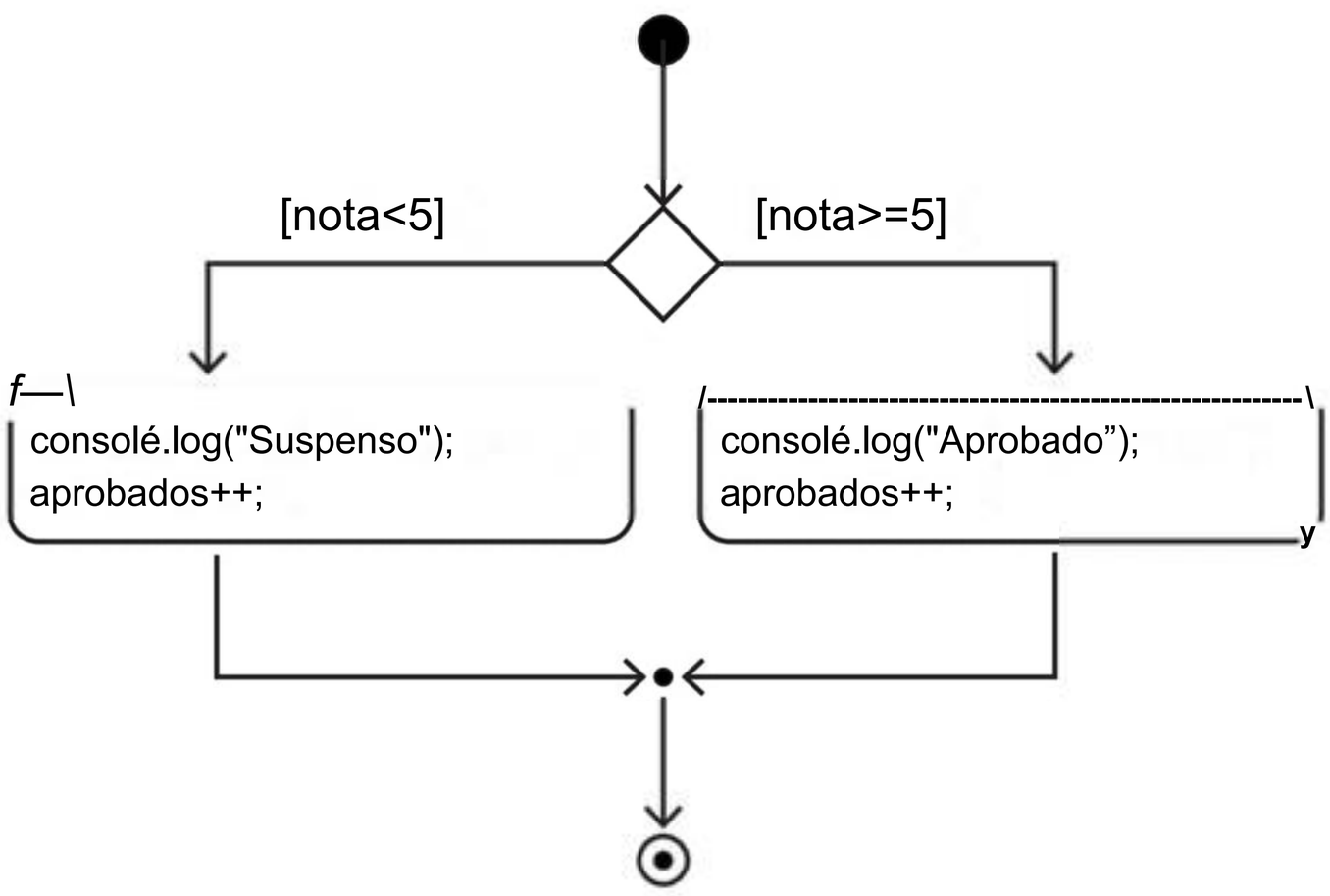


Figura 3.8: Diagrama de actividad de la sentencia **if-else**

37.3.3 ANIDACIÓN

Dentro de una sentencia **if** se puede colocar otra sentencia **if.** A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas. La nueva sentencia puede ir tanto en la parte **if** como en la parte **else.**

Las anidaciones se utilizan muchísimo al programar. Sólo hay que tener en cuenta que siempre se debe cerrar primero el último ***if*** que se abrió. Es muy importante también tabular el código correctamente para que las anidaciones sean legibles.

El código podría ser:

```
if (x==l) {
    instrucciones
...
}
else {
    if(x==2) {
        instrucciones
        ...
    }
    else {
        if(x==3) {
            instrucciones
            aaa
        }
    }
}
```

Una forma más legible de escribir ese mismo código sería:

```
if (x==l) {
    instrucciones que se ejecutan si x vale 1
    aaa
}
else if (x==2) {
    instrucciones que se ejecutan si x no vale 1 y vale 2
    aaa
}
else if (x==3) {
    instrucciones que se ejecutan si x no vale ni 1 ni 2 y vale 3
    aaa
}
else{
    instrucciones que se ejecutan si x no vale ni 1 ni 2 ni 3
}
```

Lo cual da lugar a la llamada instrucción **if-else-if.**

3.7.4 INSTRUCCIÓN WHILE

3.7.4.1 BUCLE WHILE

La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles **while** agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.

La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while

Sintaxis:

```
while (expresión lógica) {
   sentencias que se ejecutan si la condición es verdadera
}
```

El programa se ejecuta siguiendo estos pasos:

[1] Se evalúa la expresión lógica

[2] Si la expresión es verdadera ejecuta las sentencias, si no el programa abandona la sentencia **while**

[3] Tras ejecutar las sentencias, volvemos al paso 1

Ejemplo (escribe números del 1 al 100):

```
var i=l;
while (i<=100){
    consolé.log(i);
    i++;
}
```

3.1.1.1 BUCLES CON CONTADOR

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un contador (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos,... o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:

[1] Lo que vale la variable contadora al principio. Antes de entrar en el bucle

[2] Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.

[3] Las acciones a realizar en cada vuelta del bucle

[4] El valor final del contador. En cuanto se rebase el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.

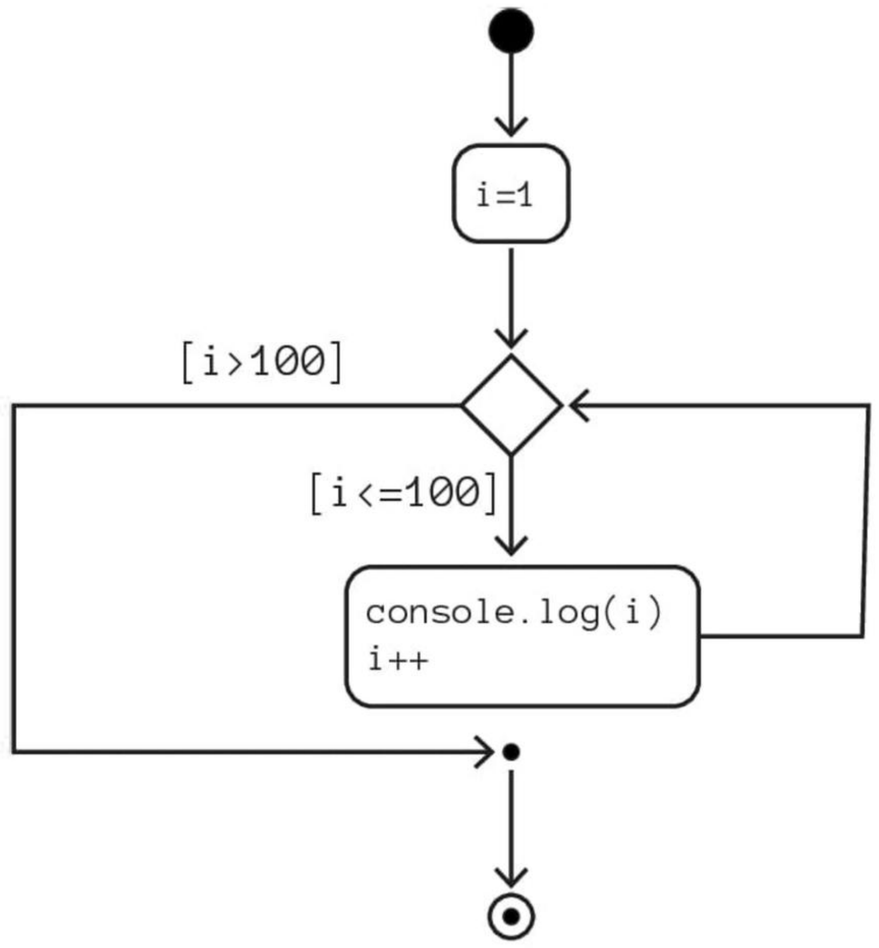


Figura 3.9: Diagrama de actividad del bucle **while**

Ejemplo:

```
let i=10; //Valor inicial del contador, empieza valiendo 10

/* condición del bucle, mientras i sea menor de
200, el bucle se repetirá, cuando i rebase este
valor, el bucle termina */

while (i<=200){
    consolé.log(i)
        /*en cada vuelta del bucle, simplemente se escribe el valor
      del contador por consola*/
    i+=10;
        /* Variación del contador, en este caso cuenta de 10 en
              10 */
}
/* Al final, el bucle escribe por consola:
10
20
30
...
y así hasta 200
*/
```

3.1.1.2 BUCLES DE CENTINELA

Es el segundo tipo de bucle básico. Una condición lógica llamada **centinela,** que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.

Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana.

Ejemplo:

```
let salir=false; /* En este caso el centinela es una variable
                    booleana que inicialmente vale falso */
let n;
while(salir==false){
      /* Condición de repetición: que salir siga siendo falso.
           Ese es el centinela.
           También se podía haber escrito simplemente:
               while(!sal ir)
   */
   n= parseInt(Math.random()*5)+l; // Lo que se repite en el bucle:
  consolé.log(i);                                  // calcular un número aleatorio de 1 a
                                                   // 5 y escribirlo por consola
   salir=(i%7==0); /* El centinela vale verdadero si el número
                                    es múltiplo de 7*/
                                    saldremos*/
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no.
- Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará).
- Un bucle de contador está relacionado con la programación de algoritmos basados en series.

Un bucle podría ser incluso mixto: de centinela y de contador. Por ejemplo, imagina un programa que escriba números de uno a 500 y se repita hasta que llegue un múltiplo de 7, pero que como mucho se repita cinco veces. Sería:

```
let salir = false;  //centinela
let n;
let i=1; //contador
while (salir == false && i<=5) {
    n = parseInt(Math.random() * 500 + 1);
    console.log(n);
```

```
        i++;
        sal ir = (n % 7 == 0);
}
consolé.log("Último número "+n)
```

3.7.5 BUCLE **DO...WHILE**

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir, el bucle al menos se ejecuta una vez. Los pasos son:

[1] Ejecutar sentencias.

[2] Evaluar expresión lógica.

[3] Si la expresión es verdadera volver al paso 1, si no continuar fuera del while.

Sintaxis:

```
do {
   instrucciones
} while (expresión lógica)
```

Ejemplo (contar de uno a 1000):

```
let i=0;
do {
    i++;
    consolé.log(i);
} while (i<1000);
```

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).

De hecho cualquier sentencia **do..while** se puede convertir en while. El ejemplo anterior se puede escribir usando la instrucción while, así:

```
let i=0;
i++;
consolé.log(i);
while (i<1000) {
     i++;
     consolé.log( i);
}
```

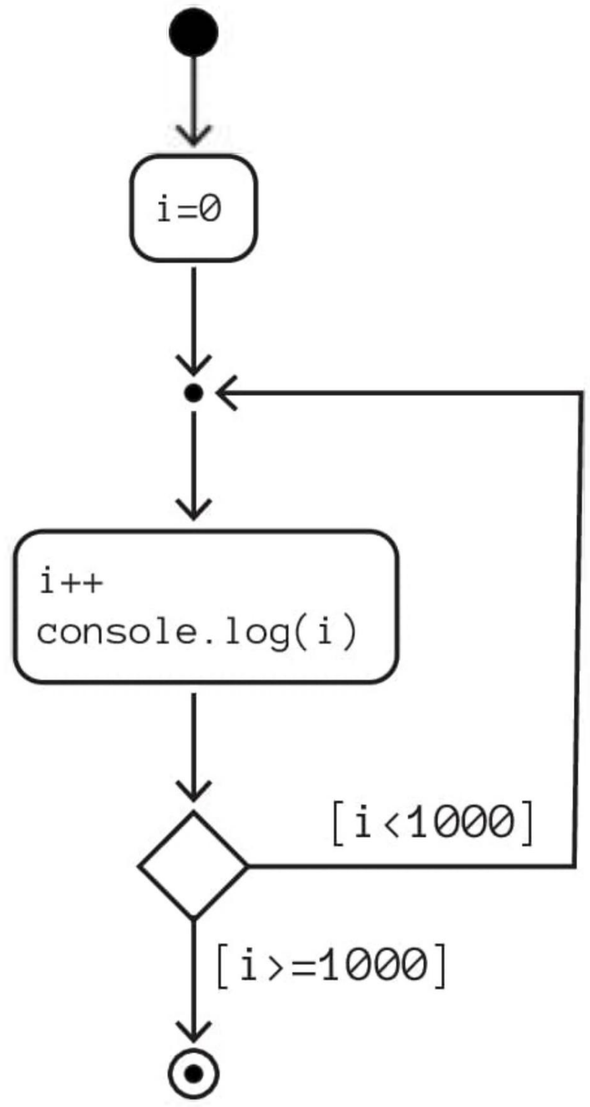


Figura 3.10: Diagrama de actividad del bucle **do..while**

3.7.6 BUCLE FOR

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador.

Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for(inicialización;condición;incremento){
   sentencias
}
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además, antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir, el funcionamiento es:

[1] Se ejecuta la instrucción de inicialización

[2] Se comprueba la condición

[3] Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque **for**

[4] Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Ejemplo (contar números del 1 al 1000):

```
for(let i=l;i <=1000;i++){
    consolé.log(i);
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle **while:**

```
var i=l; /*sentencia de inicialización*/
while(i<=1000) { /*condición*/
    consolé.log(i);
    i++; /*incremento*/
}
```

Como se ha podido observar, es posible declarar la variable contadora dentro del propio bucle fór. De hecho, es la forma habitual de declarar un contador. De esta manera se crea una variable que muere en cuanto el bucle acaba.

3.7.7 ABANDONAR UN BUCLE

JavaScript proporciona una instrucción llamada **break** que permite abandonar de golpe un bucle:

```
let m=parseInt(Math.random()*10)+1;
for(let i=l;i <=1000;i++) {
        consolé.log('i=$ i , m=$ m );
        if(m==10){
                break;
        }
        m=parseInt(Math.random()*10)+1;
}
```

Posible salida:

```
i=l,    m=7
i=2,    m=l
i=3,    m=8
i=4, m=10
```

Este extraño bucle cuenta crea un contador (i) que empieza valiendo uno y termina valiendo 1000 y cuenta de uno en uno. Sin embargo, otra variable llamada ***m,*** calcula un número aleatorio del uno al 10. La instrucción if comprueba si el número aleatorio calculado realmente es el 10, de ser así, interrumpe el bucle mediante la instrucción **break.**

La instrucción break puede ser un atajo muy interesante para salir de un bucle de forma creativa, pero también, si se usa demasiado y con poco cuidado, puede producir malos hábitos. Usar break crea código desestructurado, en el que es difícil reconocer fácilmente cuál es su flujo de trabajo.

3.7.8 BUCLES ANIDADOS

Es posible crear un bucle dentro de otro, en una operación que se llama anidamiento. Ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset= 'UTF-      8">
        <title>Document</title>
        <style>
                table{
                        border-collapse: collapse;
                        width:100%;
                }
                td{
                        border:lpx solid black;
                }
        </style>
</head>
<body>
<table>
        <script>
                for(let i=l;i <=10;i++){
                        document.write("<tr>");
                                for(let j=l;j<=5;j++){
                                        document.write("<td> </td>");
                                }
                        document.write("</tr>");
                }
        </script>
</table>
</body>
</html>
```

El resultado de este programa es el siguiente:

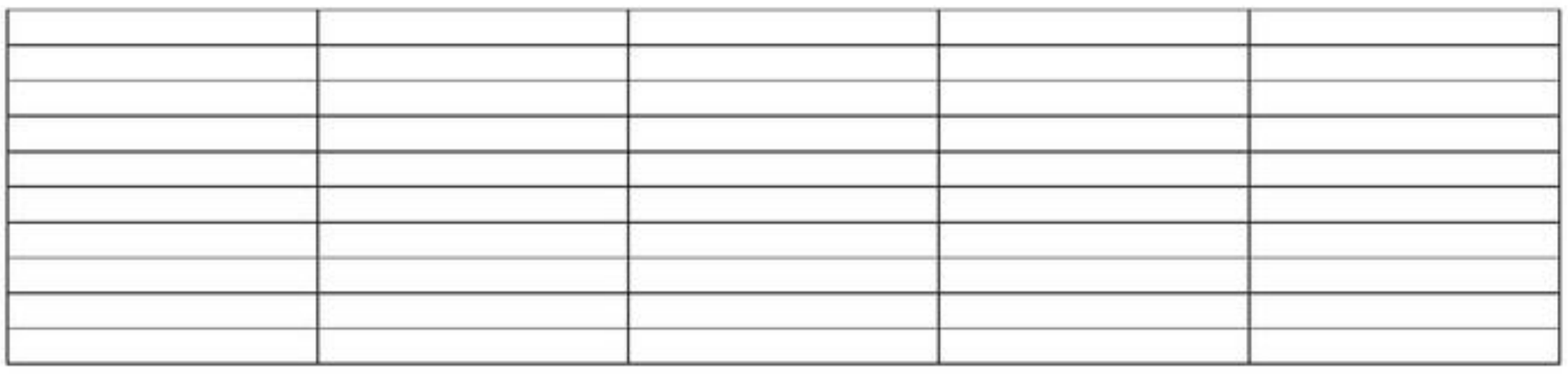


Figura 3.11: Tabla resultado del código con dos bucles anidados

El primer bucle ***for*** se repite diez veces, su contador es la variable ***i.*** Cada vez que se repite se escribe la etiqueta de inicio de fila de la tabla (tr) y después aparece el segundo bucle que se repite cinco veces. Las cinco veces que se repite ese bucle escribe una celda vacía. Finalmente se cierra la fila. Se escriben, por lo tanto, diez filas y cinco columnas en cada fila.

Los bucles anidados son muy potentes e interesantes. Por supuesto, no solo se pueden anidar bucles ***for,*** también se puede anidar cualquier otro tipo de bucle.

3.8 PRÁCTICAS RESUELTAS

Práctica 3.1: Ejecutar código en consola desde archivo externo

- Crea una página web en blanco que, simplemente, cargue un archivo JavaScript llamado ***saludo.js***
- Escribe código JavaScript en el archivo ***saludo.js*** que permita escribir hola por consola.

SOLUCIÓN: PRÁCTICA 3.1

[1] Crea un directorio para esta práctica (por ejemplo se puede llamar practica-3-1)

[2] Si deseas usar Visual Studio Code como editor, abre ese directorio para trabajar con él en Visual Studio Code.

[3] Crea una subcarpeta llamada **js** y en ella crea el archivo **saludo.js.** Escribe en ese archivo el siguiente código y guárdalo:

```
consolé.log("Hola");
```

[4] Crea la página index.html y escribe este código:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Saludo</title>
</head>
<body>
<script src="js/saludo.js"></script>
</body>
</html>
```

[5] Lanza **index.html** en un navegador. Si estás en Visual Studio Code en el archivo index.html (suponiendo que esté instalada la extensión **Live Server)** basta con pulsar **Alt+L** y luego **Alt+O.**

[6] Se mostrará la página en el navegador por defecto. Pulsando F12 podemos arcar después **Consola** (o ***Consolé)*** y, si todo se ha hecho correctamente, veremos el texto ***Hola.***

Práctica 3.2: Ejecutar código en node

- Crea un archivo llamado ***saludo.js*** que sirva para escribir **Hola** por consola.
- Ejecuta el código usando node.js.

SOLUCIÓN: PRÁCTICA 3.2

[1] Damos por hecho que Node.js está instalado y accesible desde el path del sistema.

[2] Suponiendo que usamos Visual Studio Code, creamos un directorio para la práctica y lo integramos en el editor. En él creamos el archivo **saludo.js** con este contenido:

```
consolé.log("Hola");
```

[3] Tras guardar el código, desde Visual Studio Code podemos ir al menú ***Depurar*** y elegir ***Ejecutar*** sin depurar o bien, simplemente, pulsar **Ctrl+F5.** Eso nos mostrará Hola en la consola de depuración de Visual Studio Code.

Práctica 3.3: Lectura de números

- Crea una página web que pida al usuario un número a través de un cuadro de lectura.
- Un cuadro de mensaje indica si realmente era un número o no, lo que escribió el usuario.

SOLUCIÓN: PRÁCTICA 3.3

La solución pasa por leer mediante el método **prompt** y tratar de convertir lo leído a número mediante la función de conversión **Number.** Si esta fruición retorna un valor distinto de **NaN** es que hemos leído un número. Por eso usamos la función NaN con el resultado de la función Number para ver si realmente se ha leído un número. El código podría ser:

```
<!DOCTYPE html>°
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Lectura de número</title>
</head>
<body>
<script>
        let numero=prompt("Escribe un número");
        i f(isNaN(Number(numero))){
                alert("No es un número");
        }
        else{
                alert("Es un número");
        }
</script>
</body>
</html>
```

Práctica 3.4: Fondo aleatorio

• Crea una página web que muestre un color de fondo aleatorio cada vez que entremos en ella

SOLUCIÓN: PRÁCTICA 3.4

Para resolver esta práctica nos basaremos en escribir desde JavaScript una capa que ocupe todo el fondo (usaremos CSS para darle tamaño).

Para que el fondo sea aleatorio usaremos la función de **CSS** llamada **rgb** que sirve para especificar colores utilizando un valor de rojo, otro de verde y otro de azul. El código podría ser el siguiente:

```
<!DOCTYPE html>
<html lang="es”>
<head>
        <title>Fondo aleatorio</title>
        <style>
        /* CSS para que la capa de fondo
        ocupe toda la pantalla*/

        #fondo{
                position:fixed;
                left:0;
                top:0;
                width:100%;
                height:100%;
        }
        </style>
</head>
<body>
<script>
let rojo=parseInt(Math.random()*255);
let verde=parselnt(Math.random()*255);
let azul=parseInt(Math.random()*255);
document.write(
        '<div id=’fondo1 '+
        'style=1background-color:rgb(${rojo},${verde},${azul})1 > ' +
        '</div>'
);
</script>
</body>
</html>
```

Práctica 3.5: Cálculo de salario

- Crea una aplicación web que pida, en sendos cuadros de lectura de datos: el nombre, los apellidos, el salario (número con decimales) y la edad de una persona (un número)
- Supondremos que el usuario escribe los datos correctamente (no los validaremos)
- La página indicará el nombre y apellidos escritos, la edad y el salario (una vez recalculado con lo que indican los puntos siguientes).
- Si el salario es mayor de 2000 euros, no cambiará
- Si el salario está entre 1000 y 2000:
  - Si además la edad es mayor de 45 años, se sube un 3%
  - Si la edad es menor de 45 o igual, se sube un 10%
- Si el salario indicado es menor de 1000
  - Los menores de 30 años cobrarán exactamente 1100 euros
  - Si la edad es de 30 a 45 años, el salario sube un 3%
  - A los mayores de 45 años, sube un 15%

SOLUCIÓN: PRÁCTICA 3.5

Es importante organizar la lógica del cálculo del salario para que el código sea lo más legible posible. La idea es leer desde javaScript los datos (sabiendo que no hay que validarlos) y luego definir el cálculo del salario y escribir el mensaje final.

La página web **(index.html)** simplemente cargará el archivo con el código JavaScript:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <title>Cálculo de salario</title>
</head>
<body>
<script src="salario.js"> </script>
</body>
</html>
```

El archivo **salario.js** contiene el código en sí de la aplicación:

```
/* Petición de datos */
let nombre=prompt("Escriba su nombre");
let apel1idos=prompt("Escriba sus apellidos");
let edad=prompt("Escriba su edad");
let salario=prompt("Escriba su salario");
```

```
/* valoramos el salario, empezamos con las
        situaciones a considerar cuando
        el salario es menor de 1000 euros
*/
i f(salario<1000){
        i f(edad<30)
                salario=1100;
                  else if(edad<=45) //edad entre 30 y 45
                        salario*=l.03; //subida de 3%
        else //más de 45 años
                salario*=l.15; //subida de un 15%
}
else if (salario<=2000){
        //salario entre 1000 y 2000 euros
        if(edad>45)
                salario*=l.03;
        else
                salario*=l.1;
}
//escritura final
document.write(
        " <p>Persona: ${nombre} ${apel1 idos}, " +
        "edad: ${edad} años, salario: ${salario}</p>
);
```

Práctica 3.6: Juego de adivinar número

- Haz una página web que implemente un juego de encontrar un número aleatorio bajo las premisas que se explican a continuación.
- La página calculará un número del 1 al 100.
- Luego preguntará al usuario por el número.
- Si el usuario escribe algo que no es un número, se indica el error y se vuelve a pedir el número.
- Si el número escrito por el usuario es correcto, se indica que se acertó y finalizaremos el juego.
- Si no, le dice si el número es menor o mayor y vuelve a preguntar cuál es.
- Si se cancela cualquier cuadro, el juego termina indicando que se canceló el juego.
- Al final, si se ha finalizado correctamente el juego se indica el número de intentos.
- Se permite volver a jugar al usuario mediante un cuadro de confirmación.

SOLUCIÓN: PRÁCTICA 3.6

La página web **(index.html)** simplemente cargará el archivo JavaScript que contiene todo el código del juego. Dicho código se muestra en la página siguiente:

```
<!DOCTYPE html>
<html lang=”es">
<head>
        <meta charset="UTF-8">
        <title>Juego de adivinar número</title>
</head>
<body>
<script src="Juego.js"></script>
</body>
</html>
```

El código del juego podría ser el siguiente:

```
let numero, n;
let salir=false;
let volver;
do { // se repite este código al menos una vez
        cont=0;
        // cálculo del número del 1 al 1000
        numero = parseInt(Math.random() * 1000) + 1;
        do {
        // bucle del juego, se repite hasta
        // que el usuario indique salir o acierte
                do{
                // bucle de petición del número
                // se repite mientras el usuario no
                // indique un número correcto o
                // pulse cancelar
                  volver=false;
                n = prompt("¿Qué número del 1 al 1000 crees que he pensado?");
                  // si se canceló el cuadro
                  // la variable centinela "salir"
                  // valdrá true
                  if(n==null) salir=true;
                  //comprobamos si el número es válido
                  else if(isNaN(n) II n<0 II n>1000){
                        alert("Número no válido");
                        volver=true;
                  }
}while(salir==false && volver==true);

cont++;
```

```
        if(salir==false) {
                // mensajes de lo cerca o no que
                // está el número del usuario
                // solo se muestran si no se pulsó en salir
                if (numero > n) {
                        alert("Mi numero es mayor")
                }
                else if (numero < n) {
                        alert("Mi numero es menor")
                }
        }
} while (salir==false && n != numero);
        // solo podemos salir si hemos acertado o
        // hemos cancelado algún cuadro
        i f(salir==false)
                // mensaje de acierto
                alert(" ¡Acertaste! Intentos: "+cont);
        // pedimos confirmación al usuario para
        // jugar otra vez si no ha indicado que quiere
        // salir
}while(sa1ir==false && confirm("¿Quieres jugar de nuevo?"));

//mensajes dependiendo de lo que eligió el usuario
if(salir) {
        document.write("<hl>Se canceló la partida</hl>");
}
else {
    document.write("<hl>Gracias por jugar</hl>");
}
```

Práctica 3.7: Triángulo de asteriscos

- Crea una aplicación web que pida al usuario un número entero positivo.
- Si lo que el usuario escribe no es un número o no es positivo, la página no mostrará nada. Se quedará en blanco.
- La aplicación escribirá un triángulo con tantos asteriscos como indique el número del usuario o usuaria.
- Para que el efecto sea más efectivo usaremos, para escribir los asteriscos, una letra monoespaciada.

• Ejemplo de resultado si el usuario indica el número 5:

```
*
* *
* * *
* * * *
* * * * *
```

SOLUCIÓN: PRÁCTICA 3.7

El código hace que la letra sea monoespaciada (de otro modo no se consigue el efecto) usando CSS. El código JavaScript usa dos bucles anidados para conseguir el efecto:

```
<!DOCTYPE html>
chtml lang="es">
<head>
	<meta charset="UTF-8">
	<title>Triángulo de asteriscos</title>
	<style>
	*{
		font-family:monospace;
	}
	</style>
</head>
<body>
<script>
let n;
do{
	n=prompt("Escribe el número de asteriscos que deseas " +
			"(debe de ser un número entero positivo)");
}while(isNaN(Number(n)) || n<=0);
for(let i=l;i <=n;i++){
	for(let j=l;j<=i;j++){
		document.write("* ");
	}
	document.write("<br>");
}
</script>
</body>
</html>
```

3.9 PRÁCTICAS PROPUESTAS

Práctica 3.8: Otros triángulos de asteriscos

- Haz una práctica similar a la anterior pero que consiga otros triángulos.
- Triángulo 1:



- Triángulo 2:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

- Triángulo 3:

```
* * * * *
 * * * *
  * * *
   * *
    *
```

Práctica 3.9: Factorial

- Crea una página web que pida un número entero positivo al usuario y calcule su factorial
- El factorial de 5 (se suele denotar como 5!) es el resultado de multiplicar: 1-2-3-4-5, es decir: 120.

Práctica 3.10: Tabla de códigos Unicode

- Crear una página que muestre los primeros 10000 símbolos de la tabla Unicode.
- Se mostrará en una tabla en la que en cada fila se indica el número de código, seguido del carácter de ese código.

- En cada fila se mostrarán 10 símbolos
- Ejemplo de algunas filas:

2MH	3*	2W2	□	2803	□	2804		2305	D	2906		1807	L	2WW		W		ISbi	1 0
2311		2312	□	2313		2814	□	2315	c	2916	rq	2S17		2SI8	Q°	2M9	ut	2810	□
3321	3	2Ü22	Ml	2S23	Q	2824	fe	2815	s	2«fi	a 1	3.527	a	2823	s	ZÍÍ29		2EHI	
2831		2332	£	2333		2334		2335		2336	ál	2B7	Q	2833	a	2339	s	2340	d
2841		2342	9	2843	£	2844	9	2845	e	2846	e	2847	9	2843	0	2849	9	2S5d	
2851	úl	2552	v	2553	a	2554		2355	a	2856	F 1	2857	z	1858	a	1839		2360	
2S6I	Q	2362	Cl	2363	0	2S64	ft	2365	0	2866	a	1867	rr	186%	□	2869	í	2B70	íl
2S71	y	2^2	H	2873		2374		2375	□	2876		2S77	Í	28?»	ól	J879		2850	ó1
28S1		2832	a	2W	SI	2SJU		2Jtó5	C	2886	□	2Ü8M		ISHí	Tr.	2339		2SWJ	□
3S91	fci	mz	€51	2893	n	2894		2895	□	2896	r	2W7		2898	□	2399	L1	ZWQ	□
2901		2902		2903	of	2904		2905	□	290$	□	2®R		2908		1909	e	≥910	j
2911	a	2912		2OL3	4	29 U	'f	3915	f	2916	□	291'		2913	□	'9B	c	2920	9
»21	<41	2922	u	2923	u	2924		2925	*1	2926	r	2927	£	2928	ef	2929	«	2930	I
2931		2932	q	2933	/	2934		2935	m	2936	□	2957	□	1938	Q	29»		2940	3
2941		2942		2943	n	2944	i"'I	2945	0	2946		2947	B fera	2948	□	2949	¿SH	2950	
2951	SI	2952	FT	2953	Í¡L	2954		2955		2956		2957		295»	€T	2959	ÍT	29fr:l	« .
2M1		2 0*2	p	29tf3		2W4		1965	&	2966		2967	2	2968	□	2969	lfcl	21*70	&
2971		2972	Si	»13		2974		2-975	L	2976	W	2977	LB	2973	□	2979	f®T	29ÍW	&
2931		2952	n	2983	u	2984	jb	2985	SST	2986	LI	2987	C	2988	□	2939	z	2990	LO
2991	111	29M2	a	2993	m	2994		2995	«T	2M9S	=p	2997	Al	i»m	'J9	2999	íIíl	3IMI0	6R*
30G1	üfll	3002	f"R	J4JO3		3004		3005	r	3006	0<T	3007	¿1	300»	8	3009	_TO	3010	•J®'
3011		3013		MIS	z	Mí 4	Qíí	3015	fe	301$	<3)'>	3017		MIS	Gl.jrr	3019	•2 'IT	MIO	íil'.fJT
3W1		3422		3023		3024	w	3025	□	3026	r	30Z7	c	3W»	□	3029	n	3*W	□
3M1		3032		3033		3034		3035	n	3036	□	3037		303»	□	30»		3940	J
3»4l	c	3042		3043		M44		3045	D	3046	0 1	»*7	&	3045	LL	3M»	■L	3#5O	
3051	fi»	3452	.'rrr	M53	&I	3054	«Cu	3055	«fe	3056	.-1	3057	m	305»		3059	a	3M0	lí?

Figura 3.12: Posible resultado de la práctica 3.10

Práctica 3.11: Cuadrados aleatorios

- Crear una aplicación web que muestre 2000 cuadrados de color aleatorio de 50x50 píxeles.
- Su posición en la pantalla será también aleatoria.
- Ejemplo de resultado:

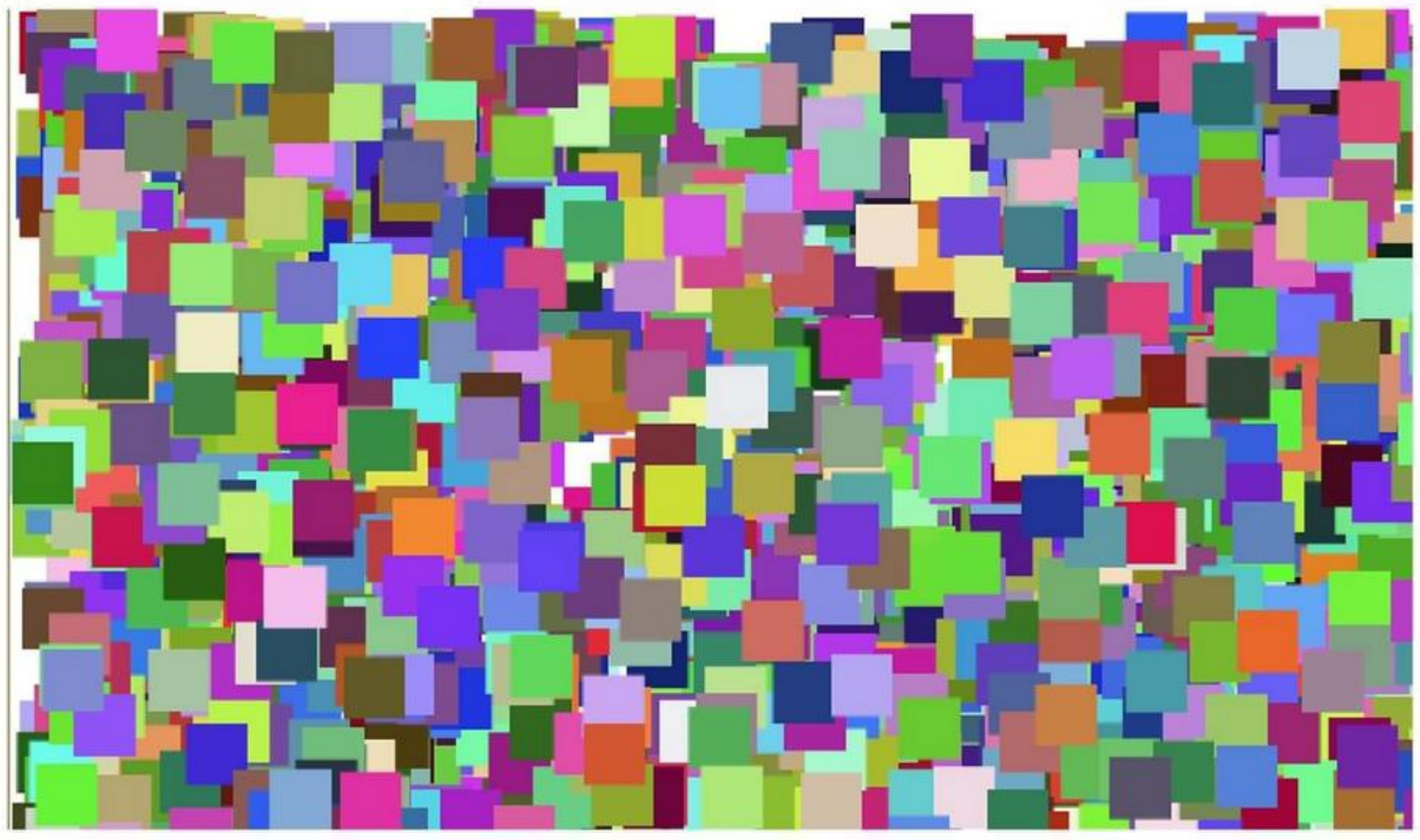


Figura 3.13: Posible resultado de la práctica 3.10

3.10 RESUMEN DE LA UNIDAD

- JavaScript es un lenguaje pensado para ser ejecutado desde los navegadores. Para ello el código debe de ser insertado mediante la etiqueta **script.** Esta etiqueta tiene capacidad para cargar código JavaScript de un archivo externo, gracias a su atributo **src.**

- Es posible ejecutar código JavaScript incluso fuera del navegador utilizando node.js. Para ello basta con grabar el código en un archivo y pedir su interpretación con el comando **node nombreDeArchi vo** en el terminal del sistema.

- La instrucción consolé, log permite mostrar en consola lo que indiquemos. La consola se muestra en un panel, que en principio no está visible, de los navegadores, node.js muestra la información de la consola directamente.

- Las normas fundamentales del lenguaje son:
  - El fin de cada línea de código se marca, opcionalmente, con punto y coma
  - Se distingue entre mayúsculas y minúsculas
  - Los comentarios de una línea se hacen con // y los de varias líneas se abren con /* y se cierran con */.
  - Los bloques de sentencias se marcan entre llaves

- Las variables se declaran con la palabra **var** o, más habitualmente, con las palabras **let** o **const.** Estas dos últimas declaran variables para ser usadas en un ámbito más restringido.

- Los tipos de datos principales posibles para las variables son: numérico, textos (strings), valores booleanos y objetos.

- Los textos deben ir entre comillados. Hay posibilidad de entrecomillar con comillas simples, comillas dobles o comillas invertidas (') . Estas últimas permiten realizar plantillas de textos para poder incorporar expresiones JavaScript en el propio texto gracias a la sintaxis $ {**expresión}.**

- Los operadores de JavaScript son:
  - Aritméticos: + – * / **
  - Encadenamiento de texto: +
  - Relaciónales: > < > = <= ! = == === !==
  - Lógicos: && | | ¡ ■
  - De BIT: & | ^ /V >><<
  - Asignación: ++ — += -= *= /= **= &= |= ^= >>= <<=

- JavaScript intenta convertir automáticamente los tipos de datos, pero es posible convertir un texto a número mediante las funciones **Number, parselnt** y **parseFloat.** Para convertir a texto disponemos del la función **String.**

Para poder mostrar y recoger información de los usuarios en las aplicaciones web disponemos de tres tipos de cuadros que todos los navegadores permiten utilizar:

- Cuadros de información, que se obtienen mediante el método **alert.**
- Cuadros de confirmación, que se obtienen mediante el método **confirm.**
- Cuadros de entrada, que se obtienen mediante el método prompt.

La sentencia if permite ejecutar instrucciones en caso de que se cumpla una determinada condición. La opción else permite que se indiquen otras instrucciones a ejecutar si se incumple la condición. Es posible combinar muchas sentencias if y else dando lugar a la llamada sentencia **if-elseif-else.**

Los bucles son estructuras que se generan a través de una sentencia **(while, do..while** o **for)** que hacen que se repitan una serie de instrucciones en el caso de que se cumpla una determinada condición.

La diferencia entre **while** y **do..while** es que en el primer caso se valora la condición antes de ejecutar las sentencias del bucle y en el caso de **do..while** se ejecutan las sentencias al menos una vez antes de valorar la condición.

El bucle **for** crea de forma más rápida bucles de contador al integrar en una sola línea la sentencia de inicio, la condición y el incremento o decremento del contador.

Disponemos, en JavaScript, de la sentencia **break** para salir de un bucle de forma directa.

3.11 TEST DE REPASO

1.- **¿Cuáles de los siguientes son comentarios válidos en JavaScript?**

a) /^comentario */
b) //comentario
c) —comentario
d) ^comentario
e) "comentario

2.- **¿Qué mostraría por consola este código?**

```
var x=19;
{
   let x=16;
}
consolé.log(x);
```

a) **19**
b) **16**
c) **undeñned**
d) Nada, ocurre un error

3.- **¿Qué mostraría por consola este código?**

```
let x=19;
{
   var x=16;
}
consolé.log(x);
```

a) **19**
b) **16**
c) **undeñned**
d) Nada, ocurre un error

4.- **¿Qué mostraría por consola este código?**

```
let x=19;
{
   x=16;
}
consolé.log(x);
```

a) **19**
b) **16**
c) **undeñned**
d) Nada, ocurre un error

5.- **¿Qué mostraría por consola este código?**

```
consolé.log("Hola \nodos");
```

a) Hola \nodos
b) Hola
odos
c) Hola
nodos
d) Hola odos

6.- **¿Cuáles de las siguientes proposiciones son ciertas?**

a) Cualquier bucle se puede escribir con la instrucción **while**
b) Hay bucles que no se pueden escribir con la instrucción **while**
c) El cuerpo del bucle **while** al menos se ejecuta una vez
d) No siempre se ejecuta el cuerpo de un bucle **while**

7.- **¿Qué muestra este código por pantalla?**

```
consolé.log(19 && true);
```

a) **true**
b) **false**
c) **NaaN**
d) **undefined**

8.- **¿Qué muestra este código por pantalla?**

```
consolé.log(12 + "25");
```

a) **37**
b) **1225**
c) **NaaN**
d) **undefined**

9.- **¿Qué muestra este código por pantalla?**

```
consolé.log("25" + 12);
```

a) **37**
b) **2512**
c) **NaaN**
d) **undefined**

10.- **¿Qué muestra este código por pantalla?**

```
let x=29;
consolé.log("Tengo ${x} años");
```

a) **Tengo ${x} años**
b) **Tengo 29 años**
c) **NaaN**
d) No se muestra nada, se produce un error

11.- **¿Qué muestra este código por pantalla?**

```
let x=29;
if(x){
   consolé.log("Correcto");
}
```

a) **Nada**
b) **Correcto**
c) No se muestra nada, se produce un error

12.- **¿Qué muestra este código por pantalla?**

```
let x=29;
if(x>15){
   consolé.log("Mayor que 15");
}
else if(x>10){
   consolé.log("Mayor que 10");
}
else{
   consolé.log("Otra");
}
```

a) **Mayor que 15**
Mayor que 10
Otra
b) **Mayor que 15**
Mayor que 10
c) **Mayor que 15**
d) No se muestra nada, se produce un error

13.- **¿Qué muestra este código por consola?**

```
let v=10;
while(v>0){
        consolé.log(v— + " ");
}
```

a) **10 987654321**
b) **9876543210**
c) **987654321**
d) Nada

14.- **¿Para qué sirve la función prompt de JavaScript?**

a) Muestra un mensaje al usuario
b) Para pedir confirmación al usuario
c) Para requerir información al usuario
d) Esa función no existe en JavaScript

15.- **¿Qué muestra este código por consola?**

```
let x=0;
for(let i=0;i<=10;i++){
     x+=i++;
}
consolé.log(x);
```

a) **55**
b) **10**
c) **11**
d) **30**

16. **¿Qué hace este código?**

```
do{
   x=prompt("Escribe un número");
   i f(isNaN(x%2)){
      x=0;
   }
}while(x%2==0);
```

a) Pide un número al usuario hasta que sea impar
b) Pide un número al usuario hasta que sea par
c) Pide un número y le deja a cero
d) Nada, se produce un error

UNIDAD 4

ESTRUCTURAS BÁSICAS DE DATOS

OBJETIVOS

— Descubrir las principales estructuras básicas de datos que aporta el lenguaje JavaScript

— Reconocer las ventajas que aporta el uso de strings

— Resolver problemas de texto utilizando los métodos de manejo incorporados de manejo de strings

— Analizar las ventajas que aportan los arrays en la programación de aplicaciones

— Crear aplicaciones web que requieran el uso de arrays

— Analizar las ventajas e inconvenientes de usar conjuntos y mapas

— Resolver problemas de código con la ayuda de mapas y conjuntos

— Distinguir el manejo de conjuntos, mapas y arrays

— Aplicar de forma correcta arrays, strings, conjuntos y mapas en la resolución de aplicaciones web

CONTENIDOS

4.1 STRINGS
- 4.1.1 USO DE STRINGS
- 4.1.2 COMPARACIÓN DE STRINGS
- 4.1.3 MÉTODOS DE LOS STRINGS

4.2 ARRAYS
- 4.2.1 ¿QUÉ ES UN ARRAY?
- 4.2.2 CREACIÓN DE ARRAYS

4.3 RECORRER ARRAYS
- 4.3.1 USO DEL BUCLE **FOR** PARA RECORRER ARRAYS
- 4.3.2 BUCLE **FOR...IN**
- 4.3.3 BUCLE **FOR..OF**

4.4 MÉTODOS DE LOS ARRAYS
- 4.4.1 TAMAÑO DEL ARRAY
- 4.4.2 SABER SI UN ELEMENTO ES UN ARRAY
- 4.4.3 MÉTODOS PARA AÑADIR ELEMENTOS
- 4.4.4 MEZCLAR ARRAYS
- 4.4.5 OBTENER Y AÑADIR SUBARRAYS
- 4.4.6 CONVERTIR ARRAY EN STRING
- 4.4.7 BÚSQUEDA DE ELEMENTOS EN UN ARRAY
- 4.4.8 MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY
- 4.4.9 DESESTRUCTURACIÓN DE ARRAYS

4.5 ESTRUCTURAS DE TIPO SET
- 4.5.1 INTRODUCCIÓN
- 4.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS
- 4.5.3 MÉTODOS DE LOS CONJUNTOS
- 4.5.4 CONVERTIR CONJUNTOS EN ARRAYS
- 4.5.5 RECORRER CONJUNTOS

4.6 MAPAS
- 4.6.1 ¿QUÉ SON LOS MAPAS?
- 4.6.2 DECLARAR MAPAS
- 4.6.3 ASIGNAR VALORES A MAPAS
- 4.6.4 OPERACIONES SOBRE MAPAS
- 4.6.5 CONVERTIR MAPAS EN ARRAYS
- 4.6.6 RECORRER MAPAS

4.7 PRÁCTICAS RESUELTAS

4.8 PRÁCTICAS RECOMENDADAS

4.9 RESUMEN DE LA UNIDAD

4.10 TEST DE REPASO

4.1 STRINGS

4.1.1 USO DE STRINGS

Ya hemos visto en el tema anterior cómo se deben utilizar los strings. Además, sabemos que son un tipo de datos básico (no son objetos como en otros lenguajes). Repasemos los conceptos ya vistos sobre strings:

- Los valores de tipo string tienen que delimitarse con comillas, bien sean dobles (“), simples (‘) o invertidas (').
- Es posible que los strings contengan comillas como carácter literal, siempre que no coincidan con el delimitador del string. Ejemplos:

```
let sl="Miguel dijo ‘venid' en voz baja";
let s2='Miguel dijo \'venid\' en voz baja';
```

- En el ejemplo ambos strings tienen el mismo contenido
- Se pueden escapar caracteres para representar caracteres no visibles, que no están en el teclado o que sean problemáticos:

```
let s3="Primera 1 inea\nSegunda línea";
let s4="Mi sonrisa: \u{lF600}"; //s4 vale: Mi sonrisa: ©
```

- Dentro de valores de string delimitados por comillas invertidas se pueden colocar directamente expresiones del lenguaje siempre que se coloquen dentro de los símbolos ${}. Ejemplo

```
let x=8, y=9;
consolé.log('La suma de x e y es: ${x+y}');
//Escribe: La suma de x e y es: 17
```

4.1.2 COMPARACIÓN DE STRINGS

Los textos se pueden comparar igual que los números.

```
let textol="ana";
let texto2="Ana";
i f(textol==texto2)
        consolé.log("son iguales");
else
        consolé.log("son distintos");
```

Por pantalla aparece el texto ***son distintos*** porque se distingue entre mayúsculas y minúsculas.

Al comparar qué texto es mayor, se usa el orden que tienen los caracteres en la tabla Unicode. Así ***Casa*** es menor que ***Caso.*** Pero, hay que tener en cuenta que las minúsculas, en la tabla Unicode, son mayores que las mayúsculas y por eso JavaScript considera que ***casa*** es mayor que ***Caso.*** También hay problemas con los caracteres especiales de cada lengua (los que no posee el

inglés) ya que ***Ñu*** se considera mayor que ***Oso,*** porque en la tabla Unicode la eñe tiene un código mayor que la letra O.

4.1.2.1 MÉTODO **LOCALECOMPARE**

Es un método que permite comparar sin discriminar entre mayúsculas y minúsculas, y considerando la forma de ordenar de cada lengua.

Este método recibe como parámetro el texto con el que deseamos comparar. Podemos indicar un segundo parámetro que es el código oficial de idioma. Sin ese segundo parámetro se utiliza la configuración local del equipo que ejecuta el código.

Este método devuelve un número negativo si el segundo texto es mayor que el primero, un número positivo si es el primer texto el que es mayor en orden alfabético, y cero si los dos textos son iguales.

```
let texto3="0so";
let texto4="Ñu";
consolé. Iog(texto3 . localeCompare(texto4));
```

Este código devuelve el número uno (1), porque considera a la eñe más pequeña en orden alfabético que la letra o. Es lo deseable al ordenar en español. Podemos indicar en el segundo parámetro el código del país cuyas reglas usamos al ordenar ***(es*** para español).

```
consolé . log( texto3 . localeCompare(texto4z "es" ));
```

4.1.3 MÉTODOS DE LOS STRINGS

JavaScript aporta numerosos métodos y propiedades a los strings. Los métodos y propiedades son parte de la programación orientada a objetos, pero, aun sin entrar en detalle en este tipo de programación (a la que dedicaremos mucho tiempo más adelante), lo único que implica este hecho es que cualquier texto tiene incorporadas funciones que nos ayudan en el uso de estos elementos.

Acceder a los métodos y propiedades es simplemente poner el nombre de la variable o el valor de tipo string, colocar un punto y escribir el nombre del método o función. Los métodos pueden requerir más información para hacer su labor, esos datos extra se conocen como parámetros.

Vamos a detallar los más interesantes métodos y propiedades.

4.13.1 TAMAÑO DEL STRING

La propiedad **length** devuelve el tamaño de un texto:

```
let texto="Nabucodonosor";
consolé.log(texto.length);
```

El resultado es el número 13 (tamaño de la palabra ***Nabucodonosor).***

4.1.3.2 OBTENER UN CARÁCTER CONCRETO

El método **charAt** permite extraer un carácter concreto del texto. Basta con indicar la posición del carácter deseado, teniendo en cuenta que el primer carácter es el número cero, el siguiente el uno y así sucesivamente. Ejemplo:

```
let texto="Nabucodonosor";
consolé. log(texto. charAt(5));
```

Escribe la letra **"o",** que es el sexto carácter, correspondiente a la posición 5.

Hay otro método llamado **charCodeAt** que funciona igual, pero devuelve el código del carácter de esa posición:

```
let texto="Nabucodonosor";
consolé.log(texto.charCodeAt(5));
```

Escribe el valor **99** que es el código Unicode de la letra ***o.***

4.1.3.3 CONVERTIR A MAYÚSCULAS Y MINÚSCULAS

Lo realizan los métodos (sin argumentos) **toUpperCase,** para pasar un texto a mayúsculas, y **toLowerCase,** para minúsculas.

```
let texto="En el año 2019 saqué el carnet de Camión";
consolé. log( texto . toUpperCase());
```

El resultado es:

```
EN EL AÑO 2019 SAQUÉ EL CARNET DE CAMIÓN.
```

Para evitar problemas debido a la forma específica de pasar a mayúsculas de algunas lenguas, disponemos de las funciones **toLocaleUpperCase** (pasar a mayúsculas) y **toLocaleLowerCase** (pasar a minúsculas). Se puede indicar un segundo parámetro con el código específico de país ***(es,fr, en,*** etc.) pero, hoy en día, las funciones habituales (toUpperCase y toLowerCase) funcionan correctamente en casi cualquier lengua.

4.1.3.4 MÉTODOS DE BÚSQUEDA DE CARACTERES

indexOf

Sintaxis:

```
texto.indexOf(texto [, inicio])
```

Devuelve la posición del **texto** indicado en la variable, empezando a buscar desde el lado derecho. Si aparece varias veces, se devuelve la primera posición en la que aparece. El segundo parámetro ***(inicio)*** es opcional y nos permite empezar a buscar desde una posición concreta.

Ejemplo:

```
var varl="Dónde esta la x, busca, busca";
```

```
document.write(vari.indexOf("x"));
//Escribe 14, posición del carácter "x"
```

En el caso de que el texto no se encuentre, devuelve -1

lastIndexOf

Igual que el método anterior, pero, en este caso, se devuelve la última posición en la que aparece el texto (o menos uno, si no se encuentra).

Itexto. lastIndexOf **(texto f. iniciol)** **I**

endsWith

itexto.endsWith(**textoABuscar[, tamañol)** **I**

Este método devuelve verdadero si el texto finaliza con el texto que indiquemos en el parámetro **textoABuscar.**

```
var texto="Esto es una estructura estática";
console. log( texto. endsWith ("estática "      ));
```

Escribe **true,** porque efectivamente el texto termina con la palabra ***estática.***

El parámetro ***tamaño*** solo mira si termina el string con el texto indicado considerando que el tamaño del texto es el indicado:

```
var texto="Esto es una estructura estática";
consolé.log(texto.endsWith("estructura",22));
```

El nuevo código vuelve a escribir **true,** porque los primeros 22 caracteres del texto terminan con el texto ***estructura.***

startsWith

Itexto. startsWithí textoABuscar [, posición])

Es muy parecido al anterior, solo que este método busca si el string comienza con el texto indicado. Si es así, devuelve true. El parámetro ***posición*** mira si el texto comienza por el indicado, pero empezando a mirar desde la posición indicada:

```
var texto="Esto es una estructura estática";
consolé.log(texto.startsWith("Esto")); //Escribe true
consolé.log(texto.startsWith( "es", 5)); //También escribe true
```

4.1.3.5 EXTRAER Y MODIFICAR SUBCADENAS

replace

Itexto.replaceftextoBuscado,textoReemp1azo)

Busca en el string el texto indicado en el primer parámetro ***(textoBuscado)*** y lo cambia por el segundo texto ***(textoReemplazo).*** Ejemplo:

```
var texto="Esto es una estructura estática";
consolé . log( texto . replace( "st", "xxtt"));
```

Escribe;

```
Exxtto es una estructura estática
```

trim

Es un método que no necesita argumentos. Se encarga de quitar los espacios en blanco a derecha y a izquierda del texto. Ejemplo:

```
let texto="                    texto con muchos espacios                    ";
consolé. log( "-'+texto+"-");
consolé.log("-"+texto.trim()+"-");
```

El resultado es:

```
-                    texto con muchos espacios                    -
-texto con muchos espacios-
```

El primer texto que se muestra es el original. Hemos colocado guiones a izquierda y derecha del mismo para que se aprecie realmente su contenido. El segundo texto deja pegados los guiones porque los espacios se han eliminado.

Es un método muy útil para ser usado cuando se realizan lecturas de datos sobre los usuarios. Es habitual necesitar que se eliminen los espacios a izquierda y derecha que, inadvertidamente, escriben los usuarios.

slice

```
texto.slice(inicio [,fin])
```

Extrae del texto los caracteres desde la posición indicada por ***inicio,*** hasta la posición ***fin*** (sin incluir esta posición). Si no se indica fin, se toma desde el inicio hasta el final.

Ejemplo:

```
let texto="Esto es una estructura estática";
consolé, log(texto. si ice(3,10));
```

El resultado obtiene desde el carácter de la posición número 3 (en realidad el cuarto) hasta el índice anterior al 10 (coge hasta el décimo carácter). En definitiva, escribe:

```
o es un
```

El parámetro ***fin*** puede usar números negativos, en ese caso, los números indican la posición contando desde el final del texto. Ejemplo:

```
let texto="Esto es una estructura estática";
```

```
consolé. log(texto.slice(3,-5));
```

Escribe por consola:

```
o es una estructura est
```

substring

Itexto. substrinq(inicio f z f in])

Funciona exactamente igual que slice anterior, pero no admite usar números negativos,

substr

Itexto. substr(inicio 1, tamaño])

Extrae del texto el número de caracteres indicados por el parámetro ***tamaño,*** desde la posición indicada por el número asociado al carácter ***inicio.*** Si no se indica tamaño alguno, se extraen los caracteres desde la posición indicada por ***inicio*** hasta el final.
Ejemplo:

```
let texto="Esto es una estructura estática";
consolé. log( texto. substr(3,10));
```

Escribe por consola:

```
o es una e
```

split

Itexto, sp 1 i t ([textoDelim í, límite!) ~"I

Es un potente método capaz de dividir el texto en un array de textos (los arrays se explican en el apartado siguiente: 4.2 ***"Arrays",*** en la página 126).

Sin indicar parámetro alguno, se genera un array donde cada elemento del array será cada carácter del string. Si se indica el parámetro ***textoDelim,*** este se usa como texto delimitador; es decir, se divide el texto en trozos separados por ese delimitador. Ejemplo:

```
let texto="Esto es una estructura estática";
consolé.log(texto.split(" "));
```

Escribe:

```
’ 'Esto', 'es', 'una', 'estructura', 'estática' ]
```

El segundo parámetro pone un límite tope de divisiones. Por ejemplo:

```
var texto="Esto es una estructura estática";
consolé.log(texto.split(" ",3));
```

Escribe:

```
[   ' Esto', 'es', 'una'          ]
```

El texto delimitador puede ser una expresión regular en lugar de un texto, eso permite dividir el texto en base a criterios más complejos

4.1.3.6 CONVERTIR CÓDIGO A TEXTO

El método **fromCharCode** permite que indiquemos una serie de números de código de la tabla Unicode y nos devolverá un string formado por los caracteres correspondientes a estos códigos.

Este método es estático, lo que significa que se usa indicando la clase String, es decir: **String. fromCharCode.** Lo cual es lógico porque este método crea un string. Ejemplo:

```
consolé.log(String.fromCharCode(65,66,67));
```

El código anterior escribe el texto ABC, porque 65 es el código de la letra A, 66 el de la B y 67 el de la C.

4.2 ARRAYS

4.2.1 ¿QUÉ ES UN ARRAY?

Todos los lenguajes de programación disponen de un tipo de variable que es capaz de manejar conjuntos de datos. A este tipo de estructuras se las llama **arrays** de datos. También se las llama **listas, vectores** o **arreglos,** pero todos estos nombres tienen connotaciones que hacen que se puedan confundir con otras estructuras de datos. Por ello, es más popular el nombre sin traducir al castellano: array.

Los arrays aparecieron en la ciencia de la programación de aplicaciones para solucionar un problema habitual al programar. Para entender este problema, supongamos que deseamos almacenar 25 notas de alumnos para ser luego manipuladas dentro del código JavaScript. Sin arrays necesitamos 25 variables distintas. Manejar esos datos significaría estar continuamente indicando, de forma individual, el nombre de esas 25 variables. Los arrays permiten manejar las 25 notas bajo un mismo nombre.

En definitiva, los arrays son variables que permiten almacenar, usando una misma estructura, una serie de valores. Para acceder a un dato individual dentro del array, hay que indicar su posición. La posición es un número entero conocido como **índice.** Así, por ejemplo ***nota[4]*** *es* el nombre que recibe el quinto elemento de la sucesión de notas. La razón por la que ***nota[4]*** *se refiere* al quinto elemento y no al cuarto es porque el primer elemento tiene índice cero.

Esto, con algunos matices, funciona igual en casi cualquier lenguaje. Sin embargo, en JavaScript los arrays son **objetos.** Es decir, no hay un tipo de datos array, si utilizamos **typeof** para averiguar el tipo de datos de un array, el resultado será la palabra **object.**

En algunos lenguajes como C, el tamaño del array se debe anticipar en la creación del array y no se puede cambiar más adelante. Este tipo de arrays son estáticos. Los arrays de JavaScript son totalmente dinámicos, su tamaño se puede modificar después de la declaración a voluntad.

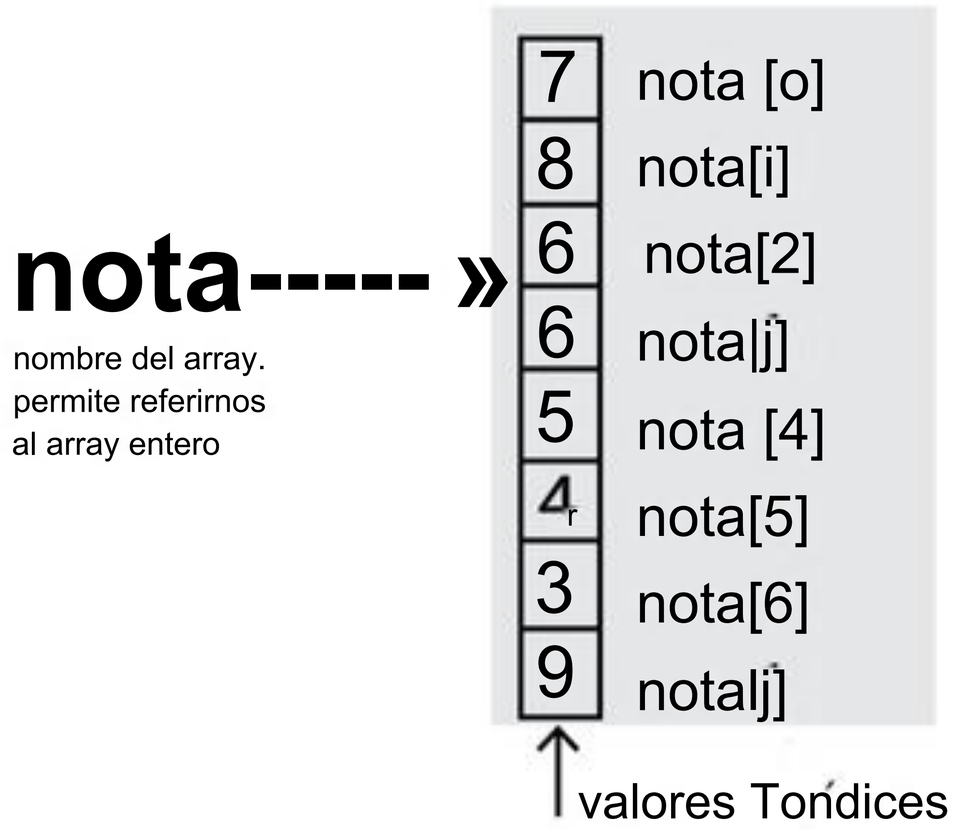


Figura 4.1: Ejemplo de array que almacena notas

Otro detalle importante, que diferencia a JavaScript respecto a lenguajes más formales, es que en JavaScript los arrays son **heterogéneos.** Los lenguajes como C o **Java** usan arrays homogéneos, que son arrays que solo pueden almacenar valores del mismo tipo. JavaScript admite, por ejemplo, que un elemento sea un número y otro un string. Cada elemento del array puede ser de un tipo distinto.

4.2.2 CREACIÓN DE ARRAYS

4.2.2.1 DECLARACIÓN Y ASIGNACIÓN DE VALORES

Hay muchas formas de declarar un array. Por ejemplo, si deseamos declarar un array vacío, la forma habitual es:

```
let a= [];
```

Los corchetes vacíos tras la asignación, significan array vacío. Un array que se llama simplemente ***a*** (que, por cierto, es un nombre muy poco recomendable) y que está listo para indicar valores.

Equivalente a ese código, podemos hacer:

```
let a=new Array();
```

Lo cual ya nos anticipa que los arrays, en realidad, son objetos. El operador **new** sirve para crear objetos.

Para colocar valores en el array se usa el índice que indica la posición del array en la que colocamos el valor. El primer índice es el cero, por ejemplo:

```
a[0]^"Antonio";
```

En este caso asignamos el texto ***Antonio*** al primer elemento del array.

Podemos seguir:

```
a[l]="Luis";
a[2]="Marta";
a[3]="Sofía";
```

Si quisiéramos mostrar un elemento concreto por consola haríamos:

```
consolé.log(a[2]); //Muestra: Marta
```

Podemos asignar valores en la propia declaración del array:

```
let nota=[7,8,6,6,5,4,3,9] ;
```

Disponemos también de esta otra forma de declarar:

```
let nota=new Array(7,8,6,6,5,4,3,9);
```

En este caso, la variable ***nota*** es un array de diez elementos, todos números. Para mostrar el primer elemento:

```
consolé.log(nota[0]); //Muestra: 7
```

El método **console.log** tiene capacidad para mostrar todo un array:

```
consolé.log(nota); //Muestra: [7,8,6,6,5,4,3,9]
```

Sin embargo, salvo para tareas de depuración, no es conveniente este uso. Lo lógico es utilizar bucles de recorrido de arrays (explicaremos los fundamentales más adelante) para mostrar el valor del array en la forma que deseemos.

4.2.2.2 USO DE CONST Y LET CON ARRAYS

Otro detalle a comentar es que es muy habitual declarar arrays con la palabra clave **const** en lugar de con **let.** En el tema anterior hemos explicado (3.3.2.1 ***"Diferencia entre let y var",*** en la página 73) la diferencia entre ambos términos. Pero con los arrays hay matices a tener en cuenta. Por ejemplo:

```
const datos=[4.5,6.78,7.12,9.123];
```

Hemos declarado un array de cuatro valores decimales. Lo hemos declarado con **const,** lo que, en principio, indica que el valor de la variable **datos** no puede variar. Sin embargo, este código es válido:

```
datos[0]=4.671;
```

Hemos modificado el primer elemento del array. Incluso es válido este otro:

```
datos[4]=3.87;
```

Hemos añadido un nuevo elemento al array. Luego, aun declarando las variables con la palabra clave **const,** se ha modificado el contenido del array.

La razón es que, realmente, los arrays son objetos. De hecho, este código:

```
consolé.log(typeof datos);
```

Como resultado escribe ***object.*** Los objetos serán tratados con profundidad más adelante (Unidad 6 ***"Programación de Objetos en JavaScript",*** en la página 203), pero ya adelantamos que cuando se crea un objeto, la variable a la que se asigna el array realmente es lo que nos sirve para llegar al valor del array, es una referencia al array, pero no es el array en sí. A los programadores en C++ les puede ser familiar que hablemos de que las variables objeto son referencias a la información que contienen, pero para programadores que comienzan a trabajar con objetos, esta idea puede resultar muy confusa.

Pero, sin entrar en estos conceptos, sí podemos entender que cuando se crea un objeto la variable que lo contiene es un enlace al mismo. Digamos que lo que realmente contiene esa variable es cómo llegar al objeto. Es decir, en el ejemplo anterior, la variable (aunque en realidad es una constante) en realidad es un identificador para acceder a los datos del array. Para entender mejor la idea, hemos de saber que este código sí provoca un error:

```
const datos=[4.5,6.78,7.12,9.123];
datos=[9.18,4.95];
```

En la primera línea declaramos ***datos*** como la forma de acceder al array que tiene valores: ***[45, 6.78, 7.12, 9.123].*** Esta variable es un enlace o una referencia ese array. Pero en la segunda línea decimos que la variable ***datos*** es un enlace a otro array. Por lo tanto, ahora sí estamos modificando la referencia y eso no se permite porque la variable ***datos*** se ha declarado con la palabra **const.**

Cuando modificamos los elementos del array, o eliminamos elementos o incluso cuando los añadimos, el array sigue siendo el mismo. La referencia no cambia.

4.2.23 OPERACIÓN DE ASIGNACIÓN CON ARRAYS

Aún hemos de entender esta idea con otro detalle:

```
const datos=[4.5,6.78,7.12,9.123];
const datos2=datos;
datos2[0]=400;
consolé.log(datos[0]); //escribe 400
```

Cuando asignamos la variable ***datos2*** al array ***datos,*** no se hace una copia del array. Tanto ***datos*** como ***datos2*** son dos variables que hacen referencia al mismo array. Por eso, cuando cambiamos el primer elemento del array ***datos2,*** también cambiamos el del array ***datos,*** ya que en realidad es el mismo array.

4.2.2.4 VALORES INDEFINIDOS

Veamos este código:

```
let a=["Saúl","Rocío"];
a[3]="María";
consolé.log(a[2]); //Escribe undefined
```

En la definición del array: let a= [“Saúl" , "Rocío"] ; estamos creando el array llamado ***a*** y dando valor a los dos primeros elementos ***(a[0] y a[l]).*** Luego, damos valor al cuarto elemento del array (a [3] ="María"). En ningún momento hemos dado valor al elemento ***a[2]*** y por eso, el código provoca la escritura del texto ***undefined.***

El resumen es que podemos dejar elementos sin definir en un array. Es más, incluso en la propia declaración se pueden dejar elementos vacíos:

```
let a=["Saúl","Rocío",,"María"];
```

Las dos comas seguidas en la definición (después del valor ***“María")*** hacen que el tercer elemento (que sería ***a[2]) se*** ignore.

4.2.2.5 ELIMINAR ELEMENTOS DE UN ARRAY

Para borrar un elemento se utiliza la palabra delete tras la cual se indica el elemento a eliminar,

```
const dias=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
delete dias[2];
consolé.log(dias);
```

Aparece:

```
[ 'Lunes',
    'Martes’,
    <1 empty item>,
    'Jueves',
    'Viernes',
    1 Sábado',
    'Domingo' ]
```

Ha desaparecido el miércoles (aparece el texto “<1 ***empty ítem>"*** en su lugar). Este elemento pasa a ser indefinido.

4.2.2.6 ARRAYS HETEROGÉNEOS

Lo hemos comentado antes, los arrays de JavaScript son heterogéneos. Admiten mezclar en el mismo array valores de diferente tipo:

```
const a=[3,4,"Hola",true, Math.random()];
```

El array contiene dos números enteros (el 3 y el 4), un valor booleano (true) y un número decimal (resultado de la expresión Math. random()).

Incluso podemos definir arrays de este tipo.

```
const b= [3,4, "Hola" , [99,55,33]] ;
```

El cuarto elemento ***(b[3J) es*** un array. Es decir, se pueden colocar arrays dentro de otros arrays. Es más, los elementos de un array pueden ser de cualquier tipo, hay arrays de todo tipo de objetos. De modo que esta instrucción es perfectamente posible:

```
consolé.log(b[3] [1]);//Escribe 55
```

En el ejemplo anterior, ***b[3][l]*** hace referencia al segundo elemento del cuarto elemento de ***b.*** En definitiva, las posibilidades de uso de arrays en JavaScript para almacenar datos son espectaculares.

4.3 RECORRER ARRAYS

4.3.1 USO DEL BUCLE FOR PARA RECORRER ARRAYS

Una de las tareas fundamentales, en la manipulación de arrays, es recorrer cada elemento de un array para poderlo examinar. Esto es fácil de hacer mediante el bucle **for:**

```
const notas=[5,6,7,4,9,8,9,9,7,8];
for(let i=0; i <notas. length; i++) {
        consolé.log('La nota ${i} es ${notas[i]}');
}
```

En el bucle se utiliza el método **length** que nos permite saber el tamaño de un array. El contador ***i*** recorre todos los índices del array ***notas.*** El resultado del código es:

```
La  nota   0  es  5
La  nota   1  es  6
La  nota   2  es  7
La  nota   3  es  4
La  nota   4  es  9
La  nota   5  es  8
La  nota   6  es  9
La  nota   7  es  9
La  nota   8  es  7
La  nota   9  es  8
```

El problema es cuando hay elementos indefinidos en el array:

```
const notas=[5,6, ,,,9,,,8,,9,,7,8];
for(let i=l;i<notas.length;i++){
        consolé.log('La nota ${i} es ${notas[i]}');
}
```

Ahora el array de notas contiene muchos elementos indefinidos. Si se deja así, el código mostrará lo siguiente cuando se ejecute:

```
La nota 1 es 6+
La nota 2 es undefined
```

```
La nota 3 es undefined
La nota 4 es undefined
La nota 5 es 9
La nota 6 es undefined
La nota 7 es undefined
La nota 8 es 8
La nota 9 es undefined
La nota 10 es 9
La nota 11 es undefined
La nota 12 es 7
La nota 13 es 8
```

Lo lógico es evitar los elementos indefinidos. Por lo que el código debería modificarse de esta forma:

```
const notas=[5,6, ,Z/9Z//8,Z9,,7,8];
for(let i=l;i cnotas.length;i++) {
        if(notas[i]!=undefined){
             consolé.log('La nota ${i} es ${notas[i]}');
        }
}
```

El bloque if permite comprobar primero que el elemento está definido y, solo si es así se muestra. El resultado sería el siguiente:

```
La nota 1 es 6
La nota 5 es 9
La nota 8 es 8
La nota 10 es 9
La nota 12 es 7
La nota 13 es 8
```

4.3.2 BUCLE FOR...IN

javaScript posee un bucle mucho más cómodo para recorrer arrays. Se trata de un bucle **for** especial llamado for..in. La ventaja es que no necesita tanto código, basta con indicar el nombre del contador y el array que se recorre. Su sintaxis general es la siguiente:

```
for(let índice in nombreArray){
    instrucciones
}
```

El ***índice*** es el nombre de una variable que se declara en el propio ***for,*** y que irá tomando todos los valores del índice del array que recorre. Esa variable se salta los elementos indefinidos, solo recorre los definidos. Por lo que el bucle ***for..in*** que nos permite recorrer las notas, saltándonos las indefinidas, es:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let i in notas){
```

```
        consolé.log('La nota ${i} es ${notas[i]}');
}
```

Se puede observar lo limpio que queda el código de recorrido con este bucle.

4.3.3 BUCLE FOR..OF

El estándar ES2015 incorporó un nuevo bucle llamado **for..of.** Pensado para simplificar aún más el recorrido de arrays, es un bucle similar al anterior, pero en el que la variable que se crea en el bucle va almacenando los diferentes valores del array (en lugar de los índices como hacía **for.. in).** Ejemplo:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let nota of notas){
        consolé.log(nota);
}
```

El resultado es:

```
5
6
undefined
undefined
undefined
9
undefined
undefined
8
undefined
9
undefined
7
8
```

Hay una gran diferencia: **for..of** no se salta los elementos indefinidos. Nuevamente, necesitamos usar la instrucción **if** para poder saltar los elementos indefinidos

```
for(let nota of notas){
        i f(nota!=undefined)
                consolé.log(nota);
}
```

Es un bucle muy simple de utilizar y se ha convertido en el bucle habitual de recorrido.

ACTIVIDAD 4.1: FOR..IN Y FOR..OF CON STRINGS

- Prueba ambos bucles con un string, como si fuera una array
- Podrás comprobar que los dos bucles funcionan con strings. Cada elemento del recorrido es uno de los caracteres del string

---

4.4 MÉTODOS DE LOS ARRAYS

Los arrays son objetos. Eso, entre otras cosas, significa que cada array tiene propiedades y métodos ya incorporados pensados para facilitarnos el trabajo. Para acceder a esos métodos basta con colocar un punto tras el nombre del array después escribir el nombre del método o propiedad.

4.4.1 TAMAÑO DEL ARRAY

La propiedad **length** nos permite obtener el tamaño de un array.

```
const dias=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
consolé.log(dias.length); //Escribe 7
```

Es fácil, gracias a este método, añadir un elemento al final de un array:

```
nota[nota.length]=8;
```

De esta forma, añadimos el número 18 al final del array ***nota.***

4.4.2 SABER SI UN ELEMENTO ES UN ARRAY

JavaScript aporta un operador llamado **instanceof.** Este método, en realidad, sirve para poder comprobar si un objeto pertenece a una determinada clase. Los arrays pertenecen a la clase **Array** (con la letra A mayúscula). Saber si una variable concreta es de clase Array es, simplemente, valorar con **instanceof** si pertenece a la clase Array. Ejemplo:

```
let a= [1,2,3,4,5,6,7,8,9];
let b=»Hola";
consolé.log(a instanceof Array); //Escribe true
consolé.log(b instanceof Array); //Escribe false
```

4.4.3 MÉTODOS PARA AÑADIR ELEMENTOS

El método **push** permite añadir un elemento al final del array. Requiere indicar qué valor le damos a ese elemento:

```
const cantantes^["Simón"];
cantantes.push("Garfunkel");
```

Ahora el array ***cantantes*** tiene dos elementos ***(Simón*** y ***Garfunkel).***

El método contrario es **pop.** Lo que hace es retirar del array el último elemento, podemos asignar dicho elemento a una variable:

```
const cantantes2=["Peter","Paúl","Mary"];
let x=cantantes2.pop();
consolé.log('El array ha quedado así: ${cantantes2}');
consolé.log('La variable x vale ${x}');
```

El método pop no tiene argumento, pero requiere paréntesis (como todos los métodos). Tras el uso de pop, se quita el valor ***Mary*** y se asigna dicho valor a una variable llamada ***x.*** Por eso, lo que se escribe en pantalla es:

```
El array ha quedado: Peter,Paul
La variable x vale Mary
```

Hay dos métodos más que permiten añadir y quitar elementos a un array. La diferencia es que lo hacen por delante, por el primer elemento del array. El método que quita el primer elemento del array se llama **shift.**

```
const cantantes3=["Crosby","Stilis"_z"Nash"];
let x=cantantes3 . shift();
consolé.log(SE1 array ha quedado: ${cantantes3}');
consolé.log('La variable x vale ${x}');
```

En este caso, es ***Peter*** el elemento eliminado, por eso lo que se escribe en pantalla es:

```
El array ha quedado: Stills,Nash
La variable x vale Crosby
```

El método contrario es ***unshift,*** al que hay que indicar como parámetro el valor a añadir por el lado izquierdo:

```
const cantantes4=["Crosby","Stilis","Nash"];
cantantes4.unshi ft("Young");
consolé.log('El array ha quedado: ${cantantes4)');
```

El texto ***Young*** pasa a ser el primer elemento, el resto se desplaza:

```
El array ha quedado: Young,Crosby,Stills,Nash
```

4.4.4 MEZCLAR ARRAYS

El método **concat** permite añadir los elementos de un array al array que usa el método. El resultado de este método es un nuevo array, no se modifica ninguno de los dos arrays originales, que contiene los elementos unidos de ambos arrays. Por eso, el resultado de ***concat*** debe de ser asignado a un nuevo array. Ejemplo:

```
const planetasl=["Venus","Mercurio","La Tierra","Marte"];
const planetas2=["Júpiter","Saturno","Urano","Neptuno"];
const planetas=planetasl.concat(planetas2);
consolé.log(planetas);
```

El array planetas contiene a todos los planetas, tanto los del primer array como los del segundo:

```
[  ' Venus',
   ' Mercurio 1,
   ' La Tierra',
   ' Marte',
   ' Júpiter',
```

```
' Saturno[1],
' Urano',
'Neptuno' ]
```

4.4.5 OBTENER Y AÑADIR SUBARRAYS

4.4.5.1 OBTENER SUBARRAYS

El método slice, permite copiar una serie de elementos de un array indicando el índice del primer elemento que deseamos copiar y el índice final. El resultado es un nuevo array.

```
const dias=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
const tresDias=dias.siice(3,6);
consolé.log(tresDias);
```

Resultado:

```
[' 'Jueves', 'Viernes', 'Sábado' ' ]
```

En la línea di as.slice (3,6) se está extrayendo los elementos con índice tres, cuatro y cinco (el sexto no se incluye). Esos tres elementos formarán un nuevo array (en el código se llama ***tresDias)***

Si el segundo número no se indica, se extrae hasta el final. Es decir, di as . si ice(3) obtiene los valores [***'Miércoles','Jueves', 'Viernes', 'Sábado']***

4A.5.2 COPIAR ARRAY

El propio método **slice** nos permite hacer una copia de un array. Para ello, basta que se use sin parámetros.

```
let array2=arrayl.slice(); //array2 es un copia de arrayl
```

4.4.53 ANADIR Y ELIMINAR ELEMENTOS

El método **splice** es muy interesante. Permite eliminar definitivamente los elementos de un array. Para ello, se indica como primer parámetro, desde dónde iniciamos la eliminación y cuántos elementos eliminamos. Este método modifica el array original y devuelve otro array que contiene los elementos eliminados.

```
const casti1laLaMancha= ["Guadalajara","Sevilla","Granada",
                                    "Ciudad Real","Albacete"];
const borrados=casti 1laLaMancha.splice(l,2);
consolé.log('Se han borrado los elementos: ${borrados}');
consolé.log('Ahora quedan: ${casti1laLaMancha}');
```

El resultado es:

```
Se han borrado los elementos: Sevilla,Granada
Ahora quedan: Guadalajara,Ciudad Real,Albacete
```

castillaLaMancha.splice(l,2) elimina desde el elemento número 1 ***(Sevilla),*** dos elementos ***(Sevilla*** y ***Granada).***

Aún más interesante es el hecho de que, además de eliminar elementos, los podemos cambiar por la serie de elementos que indiquemos:

```
const casti1laLaMancha= ["Guadalajara","Sevi1la","Granada",
                                    "Ciudad Real","Albacete"];
casti 1laLaMancha.splice(l,2,"Toledo","Cuenca");
consolé.log(castillaLaMancha);
```

Resulta:

```
[' 'Guadalajara', 'Toledo', 'Cuenca', 'Ciudad Real', 'Albacete' '
```

El número de elementos que se añaden no tiene por qué coincidir con los que se quitan.

4.4.6 CONVERTIR ARRAY EN STRING

El método **join** de los arrays permite convertir un array en un string. El string resultante contiene todos los elementos del array:

```
const dias=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
consolé.log(dias.join());
```

Escribe:

```
Lunes,Martes,Miércoles,Jueves,Viernes,Sábado,Domingo
```

Cada elemento se separa con una coma. Es posible indicar otro separador ya que se puede indicar un texto como parámetro de join para que sea utilizado como separador:

```
const días=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
consolé.log(días.join(""));
consolé.log(días.join(;          "-"))
consolé.log(días.join("<->"));
```

El resultado sería:

```
LunesMartesMiércolesJuevesViernesSábadoDomingo
Lunes-Martes-Miércoles-Jueves-Viernes-Sábado-Domingo
Lunes<->Martes<->Miércoles<->Jueves<->Viernes<->Sábado<->Domingo
```

4.4.7 BÚSQUEDA DE ELEMENTOS EN UN ARRAY

4.4.7.1 MÉTODO **INDEXOF**

Este método permite buscar el valor elemento en un array. Si lo encuentra devuelve su posición y si no, devuelve el valor -1. Ejemplo:

```
const días=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
consolé.log(días.indexOf('Jueves'));
```

Escribe el valor 3, ya que es la posición del elemento con valor ***Jueves*** en el array. Si hubiera dos elementos con ese valor, devuelve el primero empezando con la izquierda. También podemos indicar el inicio de la búsqueda, por ejemplo:

```
días.indexOf("Jueves",4);
```

Busca la palabra ***Jueves*** a partir del elemento con índice 4 (en este caso no lo encontrará y devolverá -1).

4.47.2 MÉTODO **LASTINDEXOF**

Método idéntico al anterior, solo que empieza a buscar desde el último elemento:

```
const números^[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
consolé.log(números.lastIndexOf(3));
```

Este código escribe el número 12, que es la última posición en la que aparece el valor 3 en el array. Al igual que ocurría con ***lastlndexOf,*** podemos empezar a buscar desde una posición concreta, pero, en este caso, se busca hacia la izquierda desde esa posición. Ejemplo:

```
const números^[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
consolé. log(números. lastlndexOf (3,6));
```

Escribe el número 5, porque empezando a buscar desde la posición 6 hacia la izquierda, el primer valor que tiene un tres es el elemento con índice 5.

4.47.3 MÉTODO **INCLUDES**

Apareció en el estándar ES2015. Es un método que permite buscar un elemento y devuelve **true** si ese valor está en el array y **false** si no lo está.

```
const días=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
consolé.log(días.includes('Jueves'));
```

Escribe **true.**

4.4.8 MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY

4.4.8.1 INVERTIR EL ORDEN

El método reverse es capaz de dar la vuelta a los elementos de un array:

```
const días=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
días.reverse();
consolé.log(días);
```

Aparece:

```
[ 'Domingo',
    'Sábado',
    'Viernes',
    'Jueves',
    'Miércoles',
    'Martes',
    'Lunes' ]
```

Se puede observar que reverse no devuelve un nuevo array, modifica el array original.

4.4.8.2 MÉTODO **SORT**

Sin duda, es uno de los métodos más interesantes de los arrays. Permite ordenar los elementos de un array. Ejemplo:

```
const días=["Lunes","Martes","Miércoles","Jueves",
                    "Viernes","Sábado","Domingo"];
días.sort();
consolé.log(días);
```

El resultado es:

```
[ 'Domingo',
    'Jueves',
    'Lunes',
    'Martes',
    'Miércoles',
    'Sábado',
    'Viernes' ]
```

Es una función de fácil uso pero tiene el problema habitual de no saber ordenar con las características nacionales y de considerar que las minúsculas son mayores que las mayúsculas. Observemos este ejemplo:

```
const palabras^["Ñu","Águila","boa","oso","marsopa","Nutria"];
palabras.sort();
```

```
consolé.log(palabras);
```

El resultado es terrible:

```
[’ 'Nutria', 'boa', 'marsopa',                    'oso', 'Águila', 'Ñu' ]
```

Aparece ***Nutria*** como primera palabra y el ***Ñu*** es el último (aunque hay un oso) y el ***Águila*** el penúltimo. La razón es que la letra **A** con tilde está muy por detrás de la **Z** en la tabla **Unicode.** Lo mismo le ocurre a la eñe y además, está el problema, ya comentado, de las mayúsculas y minúsculas.

La manera de solventar este problema es gracias a que el método **sort** admite indicar una función de tipo callback que nos permita personalizar la ordenación. Esta posibilidad se describe completamente en el apartado 5.4.4.1 ***"Ordenación avanzada de arrays",*** en la página 185

4.4.9 DESESTRUCTURACIÓN DE ARRAYS

La versión estándar ES2015 del lenguaje trajo una capacidad muy interesante relacionada con los arrays, pero que realmente lo que hace es añadir facilidades para manipular varios datos a la vez y recogerles en las estructuras idóneas. No afecta solo a los arrays, pero en este apartado nos centraremos en la sintaxis de desestructuración relacionada con los arrays.

Una primera capacidad que aporta es la de asignar valores a varias variables a la vez. Veámoslo con un ejemplo:

```
let [saludo,despedida,cierre]=["Hola","Adiós","Hasta nunca"];
consolé.log(saludo); //Escribe Hola
consolé.log(despedida);//Escribe Adiós
consolé.log(cierre); //Escribe Hasta nunca
```

Como vemos, se asignan de golpe valores para tres variables.

Las expresiones que se asignan pueden ser tan complejas como queramos:

```
let [ni,n2]=[10,Math.random()*20];
```

Incluso se facilita hacer la operación clásica de ***swap*** de variables. Es decir, intercambiar los valores de dos variables:

```
[a,b]=[b,a];
```

La variable ***a*** valdrá lo que valía ***b*** y viceversa.

Hay también un operador llamado operador de propagación **(spread** en inglés) que permite convertir un array en variables y viceversa. Este operador se usa con tres puntos seguidos (...) Ejemplo:

```
let array=[1,2,3];
let [x,y,z]=[...array];
```

La variable ***x*** valdrá uno, ***y*** valdrá dos y ***z*** valdrá tres. Incluso es válida este código:

```
let array=[l,2,3];
```

```
let [x,,y]=[...array];
```

Ahora ***x*** vale uno e ***y*** vale dos.

También es válida esta sintaxis:

```
let a,b,array;
[a,b,...array] = [1,2,3,4,5];
```

En este código, la variable ***a*** valdrá 1, fe valdrá 2 y ***array*** será efectivamente un array con tres elementos que valdrán 3, 4 y 5 respectivamente.

Esta nueva sintaxis es muy poderosa, pero conviene adaptarse a ella poco a poco.

4.5 ESTRUCTURAS DE TIPO SET

4.5.1 INTRODUCCIÓN

Los **Sets** (que se pueden traducir con **conjuntos)** son una estructura de datos (son objetos realmente) que permiten, de forma similar a los arrays, almacenar datos. Es una estructura de datos que apareció en el estándar ES2015-

A diferencia de los arrays, no admiten valores duplicados y esa es su virtud. Es muy habitual recoger datos, pero eliminando aquellos que ya están repetidos. En los arrays esta tarea es pesada de realizar, pero con los conjuntos es muy sencilla ya que no hay que implementar nada, ya vienen preparados para esta labor.

4.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS

Podemos declarar e iniciar un conjunto vacío de valores:

```
const lista=new Set();
```

Esta forma declarar e iniciar variables deja claro que los conjuntos son objetos.

Añadir nuevos valores al conjunto requiere del uso del método add:

```
lista.add(8);
lista.add(6);
lista.add(5);
lista.add(5);
lista.add(6);
consolé.log(lista);
```

Lo cual muestra el siguiente resultado:

```
Set { 8, 6, 5 }
```

Dejando claro que los valores duplicados no se mantienen en el conjunto.

Como el método **add** devuelve una referencia al propio conjunto, se puede realizar la misma acción de esta forma:

```
lista.add(8).add(6).add(5).add(5).add(6);
consolé.log(lista);
```

Podemos también iniciar la lista a partir de un array:

```
const lista=new Set( [5,6,4,5,6,5,5,6,4,6,6]);
consolé.log(1ista);
```

Muestra:

```
Set { 5, 6, 4 }
```

Aunque los datos procedan de un array, no se permiten valores duplicados.

Es posible también iniciar el conjunto con un texto:

```
const lista=new Set("Conjunto");
consolé.log(lista);
```

Sin embargo, el resultado es sorprendente:

```
Set { 'C', 'o', 'n', 'j', 'u', 't' }
```

Cada elemento del conjunto es una letra y, como siempre, no se repiten los mismos caracteres.

Sí es posible añadir strings largos, pero mediante el método add:

```
const lista=new Set();
lista.add("Conjunto”);
consolé.log(lista);
```

Resultado:

```
Set { 'Conjunto' }
```

4.5.3 MÉTODOS DE LOS CONJUNTOS

4.5.3.1 TAMAÑO DEL CONJUNTO

La propiedad size permite saber el tamaño de un conjunto:

```
const lista=new Set([2,4,6,8,10]);
consolé.log(1ista.size);
```

Escribe el número cinco, ya que hay cinco elementos en el conjunto.

4.53.2 ELIMINAR VALORES

El método **delete** elimina el valor indicado de un conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
```

```
lista.delete(6);
consolé.log(lista);
```

Escribe:

```
Set { 1, 2, 3, 4, 5, 7, 8, 9 }
```

El número 6 ha desaparecido del conjunto.

Hay un método más agresivo: **clear.** Este método elimina todo el conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
lista.clear();
consolé.log(lista);
```

Escribe:

```
Set {}
```

Indicando que el conjunto está vacío.

4.53.3 BUSCAR VALORES

El método **has** permite que se le indique un valor y devuelve **true** si dicho valor forma parte del conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
consolé.log(1ista.has(7)); //escribe true
consolé.log(lista.has(10)); //escribe false
```

4.5.4 CONVERTIR CONJUNTOS EN ARRAYS

JavaScript posee un operador muy interesante conocido como operador de **propagación** (en inglés se llama operador **spread).** Este operador usa tres puntos seguidos detrás de los cuales podemos indicar el conjunto a convertir.

```
const lista=new Set([1,2,3,4,5,6,7,8,9])
const array=[...1ista];
consolé.log(array);
```

Escribe:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

El conjunto se ha convertido en un array.

4.5.5 RECORRER CONJUNTOS

Al igual que ocurre con los arrays, es habitual necesitar recorrer cada elemento de un array. Para ello, disponemos de los bucles **for..of** que recorren cada elemento del conjunto:

```
const lista=new Set( [1,2,3,4,5,6,7,8,9] );
```

```
for(let numero of lista){
        consolé.log(numero);
}
```

El resultado escribe cada número en una línea distinta. Solo necesitamos en este tipo de bucles indicar una variable para que vaya recogiendo los valores de cada elemento del conjunto. Esta variable se declara justo antes de la palabra **of.** El nombre del conjunto a recorrer se coloca después de esta misma palabra.

4.6 MAPAS

4.6.1 ¿QUÉ SON LOS MAPAS?

Los mapas permiten, en JavaScript, crear estructuras de tipo **clave-valor,** en las cuales las claves no se pueden repetir y tienen asociado el valor. Los mapas son también parte del estándar ES2015.

Tanto las claves como los valores pueden ser de cualquier tipo. En un mismo mapa no puede haber dos elementos con la misma clave, pero sí pueden repetir valor.

4.6.2 DECLARAR MAPAS

Para declarar un mapa e iniciarlo sin contenido basta el siguiente código:

```
const provincias=new Map();
```

Como los arrays y los conjuntos, realmente los mapas son objetos.

4.6.3 ASIGNAR VALORES A MAPAS

4.6.3.1 MÉTODO SET

El método **set** es el que permite asignar nuevos elementos. Este método requiere la clave del nuevo elemento y después el valor con el que asociamos dicha clave:

```
const provincias=new Map();
provincias.set(l,"Álava");
provincias.set(28,"Madrid");
provincias.set(34,"Palencia");
provincias.set(41,"Sevilla");
consolé.log(provine ias);
```

El resultado es:

```
Map {
    1 => 'Álava',
    28 => 'Madrid',
    34 => 'Palencia',
```

```
  41 => 'Sevilla'
}
```

Si volvemos a añadir un elemento con la misma clave, este sustituye al anterior ya que no puede haber claves repetidas.

Es posible encadenar los métodos **set:**

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencía" )
        .set(41,"Sevilla");
consolé.log(provine ias);
```

4.63.2 USO DE ARRAYS PARA CREAR MAPAS

También podemos utilizar un array donde cada elemento es otro array, en el que el primer elemento es la clave y el segundo el valor de esa clave. Podemos, a partir de dicho array, crear un mapa con las claves y valores del array:

```
const personas=new Map([[1,"José"],[2,"María"],[3,"Elena"]]);
consolé.log(personas);
```

Escribe:

```
Map { 1 => 'José', 2 => 'María', 3 => 'Elena' }
```

4.6.4 OPERACIONES SOBRE MAPAS

4.6.4.1 OBTENER VALORES DE UN MAPA

En los mapas es posible obtener el valor de una clave a partir del método **get** al que se le indica la clave del elemento que queremos obtener. La potencia de los mapas está en que obtener el valor referente a una clave es una operación muy rápida.

```
consolé.log(provine ias.get (34)); //Escribe Patencia
```

4.Ó.4.2 BUSCAR UNA CLAVE EN UN MAPA

El método **has** permite buscar una clave en un mapa. Si la encuentra, devuelve **true** y si no, devuelve **false.**

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencia")
        .set(41,"Sevilla");
consolé.log(provincias.has(34)); //Escribe true
consolé.log(provine ias.has("Palencia"));//Escribe false
```

En el código anterior la búsqueda de la clave ***34*** produce un valor de **true** porque hay un elemento en el mapa que tiene esa clave. Al buscar el texto ***"Palencia"*** se devuelve false porque Palencia es un valor, pero no una clave.

4.6.43 BORRAR VALORES

El método **delete** permite eliminar un elemento del mapa. Para ello, hay que indicar la clave de dicho elemento:

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Paleneia")
     .set(41,"Sevilla");
provincias.delete(34); //Elimina el elemento con valor "Patencia"
consolé.log(provine ias);
```

Resultado:

```
Map { 1 => 'Álava', 28 => 'Madrid', 41 => 'Sevilla' }
```

4.6.4.4 OBTENER OBJETOS ITERABLES

Un objeto iterable es un tipo de objeto semejante a un array, ya que es una colección de valores que se pueden recorrer mediante bucles de tipo **for...of.** Sin embargo, los objetos iterables no son realmente arrays y, por lo tanto, se manipulan de forma distinta.

Los mapas permite crear objetos iterables que contienen solo las claves y objetos iterables solo con los valores. El método **keys** obtiene las claves y el método **valúes** obtiene los valores.

Ejemplo:

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencía")
       ,set(41,"Sevilla");
let claves=provincias.keys();
for(let k of claves){
      consolé.log(k);
}
consolé. log("----------" );
let valores=provincias.values();
for(let    v    of    valores){
      consolé.log(v);
}
```

Resultado:

```
1
28
34
41
----------
Álava
Madrid
Palencia
Sevilla
```

4.6.5 CONVERTIR MAPAS EN ARRAYS

Al igual que ocurre con los conjuntos, el operador de propagación de JavaScript es el ideal para esta labor:

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencia" )
        .set(41,"Sevilla");
consolé.log(...provincias);
```

El resultado de este código permite ver el array que produce el operador:

```
[ 1, 'Álava' ] [ 28, 'Madrid' ] [ 34, 'Palencia' ] [ 41, 'Sevilla' ]
```

4.6.6 RECORRER MAPAS

El bucle **for...of** es el ideal para recorrer el contenido de los mapas. Cada valor que recoge este bucle es un array de dos elementos: el primero es la clave y el segundo el valor. La forma de recorrer los índices de un array es la siguiente:

```
const provincias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencia")
        ,set(41,"Sevilla");
for(let elemento of provincias){
        consolé.log(elemento);
}
```

Usando la forma habitual de recorrido de **for..of,** resulta que se nos proporciona en la variable que recoge cada elemento, un array con dos elementos: el primero es la clave del elemento y el segundo es el valor. Por ello, el código anterior mostraría este resultado:

```
[ 1, 'Álava' ]
[ 28, 'Madrid' ]
[ 34, 'Palencia' ]
[  41, 'Sevilla' 1
```

También podemos desestructurar el array de los elementos para separar en dos variables la clave y el valor. Este podría ser otra versión del bucle para el mismo array:

```
for(let [clave,valor] of provincias){
        consolé.log('Clave: ${clave}, Valor: ${valor}');
}
```

La salida de este código es:

```
Clave: 1, Valor: Alava
Clave: 28, Valor: Madrid
Clave: 34, Valor: Palencia
Clave: 41, Valor: Sevilla
```

Este bucle es más versátil por separar de forma más cómoda la clave y el valor.

Los métodos **keys** y **valúes** también nos permiten recorrer las claves y los valores por separado. Para las claves sería:

```
for(let clave of provincias.keys()) {
    consolé.log(clave);
}
```

Para los valores:

```
for(let valor of provincias . values()) {
    consolé.log(valor);
}
```

Son muchas posibilidades, lo que otorga una gran versatilidad a los programadores a la hora de utilizar mapas.

RECOMENDACIÓN: FUNCIONES CALLBACK EN ARRAYS, CONJUNTOS Y MAPAS

- En el tema siguiente se estudiarán más métodos sobre arrays, mapas y conjuntos que realizan tareas muy avanzadas, pero que requieren del uso de funciones callback. Como este tipo de funciones se explican en el tema siguiente, será en ese momento cuando veamos estos métodos (véase 5.4.4 *"Uso de métodos avanzados para manipular estructuras de datos",* en la página 185

4.7 PRÁCTICAS RESUELTAS

Práctica 4.1: Cifrado estilo César

- El cifrado César consiste en tomar cada letra de un mensaje y desplazarla en el alfabeto el número que diga una clave. Por ejemplo, la letra A desplazada con clave 2, se convertirá en C.
- Aunque el cifrado César funciona con un alfabeto completo y por desplazamiento (en el ejemplo anterior la letra Z se suele convertir en B) crea una página que pida al usuario un texto y una clave y escriba el mismo texto pero desplazando los caracteres en la tabla Unicode el número que indique la clave

SOLUCIÓN: PRÁCTICA 4.1

El código de la solución podría ser el siguiente:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Cifrado estilo César</title>
</head>
<body>
<script>
let texto=prompt("Escriba el texto que desea cifrar");
let clave;
do{
        clave=prompt("Escriba la clave (un número entero)")
        i f(isNaN(Number(clave))){
                alert("No has escrito un número");
        }
}while(isNaN(Number(clave)));
clave=parselnt(clave); //si hay decimales los quitamos
for(let letra of texto){
        let ci fra=letra.charCodeAt(letra)+clave;
        document.write(String.fromCharCode(ci fra));
}
</script>
</body>
</html>
```

Práctica 4.2: Cifrado estilo polialfabético

- Se trata de mejorar el cifrado anterior en el que cada letra siempre se va a corresponder con la misma letra cifrada (lo que facilita su descifrado)

- Por eso haremos una aplicación similar, pero ahora como clave pediremos un texto para cifrar que deberá contener números. La clave la cifraremos de forma cíclica:
- Ejemplo:

```
Mensaje:                Este es el mensaje
Clave:                  12345
Mensaje cifrado: Fuwi%fu#iq!ohrxblh
```

- El mensaje cifrado se forma de esa manera:
  - E+1=F
  - s+2=u
  - t+3=w
  - e+4=i
  - (espacio en blanco)+5=%
  - e+l=f
  - s+2=u
  - etc.

SOLUCIÓN: PRÁCTICA 4.2

En el código lo más llamativo es recorrer ambos textos (clave y texto original) e ir sumando sus códigos.

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Cifrado estilo polialfabético</title>
</head>
<body>
<script>
let texto=prompt("Escriba el texto que desea cifrar");
let clave=prompt("Escriba la clave (una serie de números");
let i=0; //i se encarga de recorrer las cifras de la clave
for(let letra of texto){
        let cifra=letra.charCodeAt(letra)+Number(clave.charAt(i));
        document.write(String.fromCharCode(ci fra));
        i++;
        if(i==clave.length) i=0;
}
</script>
</body>
</html>
```

Práctica 4.3: Lotería primitiva

- Crear una aplicación web que muestre cincuenta combinaciones para jugar a la lotería primitiva
- Las combinaciones son seis números del 1 al 49, pero hay que tener en cuenta que no se pueden repetir los números

SOLUCIÓN: PRÁCTICA 4.3

Como los números de la primitiva no se pueden repetir, sería un error resolver este problema con un código que, simplemente, muestre seis números aleatorios del uno al 49.

En su lugar crearemos un array de 49 elementos con cada uno de los números del 1 al 49. Después moveremos los elementos de forma aleatoria en el array. Lo haremos intercambiando 50 veces (el 50 es un número quizá suficiente, pero podemos realizar más intercambios) los números del array de forma aleatoria. Finalmente, mostraremos los seis primeros números del array cada vez que revolvamos el array.

Mostramos solo el código JavaScript de la solución (la página web contendría simplemente la etiqueta script invocando al archivo JavaScript que contenga este código):

```
//constantes, buena práctica por si queremos
//facilitar el mantenimiento del código
const N_COMBINACIONES=50;
const N_INTERCAMBIOS=50;
const RANGO JT0TAL=49;
const RANG0_S0LUCI0N=6;
//array que conmtendrá los números de la primitiva
const primitiva^ [];

//relleno del array original
for(let i=0;i<RANGO_TOTAL;i++){
        //el índice 0 contendrá el número 1
        //el 1 el 2 y así sucesivamente hasta
        //los 49 números
        primitiva[i]=i+l;
}

//bucle principal, en cada vuelta se
//muestra una combinación
for(let i=l;i <=N_COMBINACIONES;i++){
        //barajeo del array
        for(let j=l;j<=N_INTERCAMBIOS;j++){
                //il e i2 toman un índice
                //del array de forma aleatoria
                let il=parseInt(Math.random()*primitiva.length);
                let i2=parse!nt(Math.random()*primitiva.length);
```

```
        //intercambio de los valores de dos
        //elementos aleatorios
         [primitiva[il],primitiva[i2]] = [primitiva[i 2],primitiva[il]];
    }
    document.write('<h2>Combinación número ${i}</h2>');
    //mostrar seis primeros elementos del array
    document.write("<p>");
    for(1et j =0;j <RANGO_SOLUCION;j ++){
        document.write ('${primitiva[j]}          `);
    }
    document.write("< /P>");
}
```

Práctica 4.4: Frecuencias de números

- Queremos realizar una aplicación que nos permita comprobar hasta qué punto la función **Math.random** es realmente aleatoria.
- Para ello, calcularemos 10.000 veces números aleatorios del 1 al 10.
- Mostraremos en una página web la frecuencia con la que ha salido cada número.
- Ejemplo de resultado:

Frecuencias

- **Número 1**: 1016
- **Número 2**: 1019
- **Número 3**: 1059
- **Número 4**: 992
- **Número 5**: 995
- **Número 6**: 969
- **Número 7**: 977
- **Número 8**: 1003
- **Número 9**: 977
- **Número 10:** 993

Figura 4.2: Ejemplo de resultado de análisis de 10.000 números aleatorios del 1 al 10

SOLUCIÓN: PRÁCTICA 4.4

En este caso, aunque se podría solucionar con un array, solucionaremos el problema usando mapas. Haremos un mapa donde cada elemento represente a un número del 1 al 10. Esos valores serán las claves. Cada clave se relacionará con la frecuencia obtenida por el cálculo.

Mostramos solo el código JavaScript de la solución:

```
const N_REPETICIONES=10000;
//preparación del mapa inicial con las frecuencias a cero
let mapa=new Map();
for(let i=l;i <=10;i++){
        mapa.set(i,0);
}

//10.000 veces calculamos números aleatorios del 1 al 10
for(let i=l;i<=N_REPETICIONES;i++){
        let n=parselnt((Math.random()*10)+1);
        let frecActual=mapa.get(n);
        mapa.set(n,frecActual+l);
}

//mostramos los resultados
document.write("<hl>Frecuencias</hl>");
document. write( " <ul >    " ) ;
for(let i=l;i <=10;i++){
        document.write(
                '<li><strong>Número ${i}:</strong> '+
                '${mapa.get(i)}</li>'
        );
}
document.write("</ul>");
```

Práctica 4.5: Validación de contraseña

- Queremos realizar una aplicación web que pida al usuario un nombre y una contraseña. Pediremos ambos datos con cuadros de entrada de texto (aunque no sea el mejor método).
- Suponemos que el usuario está introduciendo esta información para darse de alta en algún servicio. La contraseña será visible (no es lo ideal, lógicamente, pero estamos aun empezando a trabajar con JavaScript) por lo que no pediremos repetirla.
- El nombre de usuario solo puede estar formado por letras minúsculas y números, de otro modo, indicaremos el error y pediremos de nuevo el nombre.
- Lo que validaremos de la contraseña es que al menos tenga un elemento de los siguientes:
  - Una letra mayúscula
  - Una letra minúscula
  - Un número
  - Un carácter que no sea ni letra ni número

- Aunque sería más fácil resolver esta práctica con expresiones regulares (véase 6.4.3 *"Expresiones regulares",* en la página 222) lo haremos sin ellas como oportunidad para practicar con los st rings.
- Si la contraseña no cumple lo indicado en el punto anterior, pediremos al usuario que la indique de nuevo. Cuando lo cumpla, indicaremos con un mensaje que se ha almacenado correctamente la información.
- Ante cualquier cancelación en los cuadros, la aplicación termina. Lo indicaremos con un mensaje.

SOLUCIÓN: PRÁCTICA 4.5

Para solucionar esta práctica crearemos un string con las letras minúsculas válidas (tomaremos las de las lenguas españolas) y otro con los números. Iremos validando carácter a carácter si se encuentran en los strings creados. No creamos un string con las mayúsculas, bastarán con pasar el de las minúsculas con ayuda del método **toUpperCase.**

Hay varias variables centinelas que ayudan a tomar decisiones, una avisa de si el usuario ha cancelado.

Se muestra el código JavaScript:

```
let nombre, contraseña;
//centinela para saber si el usuario cancela un cuadro
let seguir=true;
//centinela que avisa de la corrección del nombre
let nombreOK;
//centinela que avisa de la corrección de la contraseña
let conmtraseñaOK;
//centinelas de cada premisa para la contraseña
let hayMinúsculas=false;
let hayMayúsculas=false;
let hayNúmeros=false;
let hayOtros=false;

//strings con las letras preparadas
const minúsculas^
     "aáábcqdeééfghi i ijklmnñoóóppqrstuúüuvwxyz"     ;
const números="0123456789";

//petición de nombre de usuario
do{
       nombre=prompt("Escriba su nombre de usuario deseado");
       if(nombre==nul1) seguir=false;
       if(seguir){
              //validación del usuario
              nombreOK=true;
```

```
let i=0;
while(nombreOK && i<nombre.length){
        let caracter=nombre.charAt(i);
        //validamos si alguna letra minúscula del
        //español
        i f(minúsculas.indexOf(carácter)==-l){
                //no se encontró el carácter en las minúsculas
                nombreOK=false;
                alert("El usuario es incorrecto\n" +
                        "Solo se admiten minúsculas");
        }
        i++;
        }
    }
}while(seguir==true && nombreOK==false);
if(seguir){
    //petición de nombre de contraseña
    do{
        hayMinúsculas=false;
        hayMayúsculas=false;
        hayNúmeros^ fa1se;
        hayOtros=false;

        contraseña=prompt("Escriba su contraseña deseada");
        if(contraseña==nul1) seguir=false;
        if(seguir){
            //validación de contraseña
            contraseñaOK=true;
            let i=0;
            whi le(contraseñaOK && i «¿contraseña . length) {
                let caracter=contraseña.charAt(i);
//validamos si hay alguna letra minúscula
if(minúsculas.indexOf(carácter)!=—1){
    hayMinúsculas=true;
}
//validamos si hay alguna letra mayúscula
el se i f (minúsculas . tollpperCase(). indexOf (carácter)! =-l) {
    hayMayúsculas=true;
}
//validamos si hay algún número
else if(números.indexOf(carácter)!=-l){
    hayNúmeros=true;
}
else {//es otro tipo de carácter
    hayOtros=true;
}
```

```
                i++;
            }
            contraseñaOK=(
                hayM i núscu1as==true    &&  hayMayúsculas==true  &&
                hayNúmeros==true &&   hayOtros==true
            );
            if(contraseñaOK==false){
                alert("La contraseña es incorrecta\n" +
                "Debe contener minúsculas, mayúsculas," +
                " números y otros símbolos");
            }
        }
            //Este     es un código de comprobación
            //es interesante para comprobar
            //como va la validación por consola
            consolé.log(
                "Mayúsculas +:hayMayúsculas +
                " , Minúsculas + hayMinúsculas +
                " , Números :" + hayNúmeros +
                " , Símbolos :" + hayOtros
            );
        }while(seguir==true && contraseñaOK==false);
}
//mensajes finales
i f(seguir==true){
    document.write("<p>Datos almacenados</p>");
}
else{
    document.write("<p>El usuario canceló la operación</p>");
}
```

4.8 PRÁCTICAS RECOMENDADAS

Práctica 4.6: Mostrar asteriscos para un array

- Genera un array con 20 números aleatorios del 1 al 50.
- Recorre el array mostrando tantos asteriscos como se indique en el número de cada elemento.
- Ejemplo de resultado (con 9 elementos) con el array:

2	3	5	7	4	6	3	2	9

Resultado:

```
**
***
**
****
*
***
**
```

Práctica 4.7: Palíndromos

- Crea una aplicación web que lea un texto e indique si el mismo es un palíndromo.
- Un palíndromo es un texto que se lee igual de derecha a izquierda que de izquierda a derecha
- Ejemplos de palíndromos:
  - Dábale arroz a la zorra el abad
  - Logré ver gol
  - Luz azul
  - ¿Son muías o cívicos alumnos?
  - Lo sé, Dama de Sol.
- Hay que tener en cuenta que para que se consideren bien los palíndromos, se ignoran los signos de puntuación (espacios, interrogaciones, comas, puntos, etc.) también se ignoran tildes y diéresis (se considera igual el carácter ***á*** que el carácter ***a)*** y no se distingue entre mayúsculas y minúsculas.

Práctica 4.8: Media de números

- Crea una aplicación web que pida continuamente números hasta que el usuario indique el número cero.
- Al final indicará la media de los números introducidos.
- Si en algún momento no es un número lo que escribe el usuario, se le pide de nuevo.
- Si cancela cualquier cuadro de lectura, el programa termina y se indica que el usuario ha cancelado y no se mostrará la media.

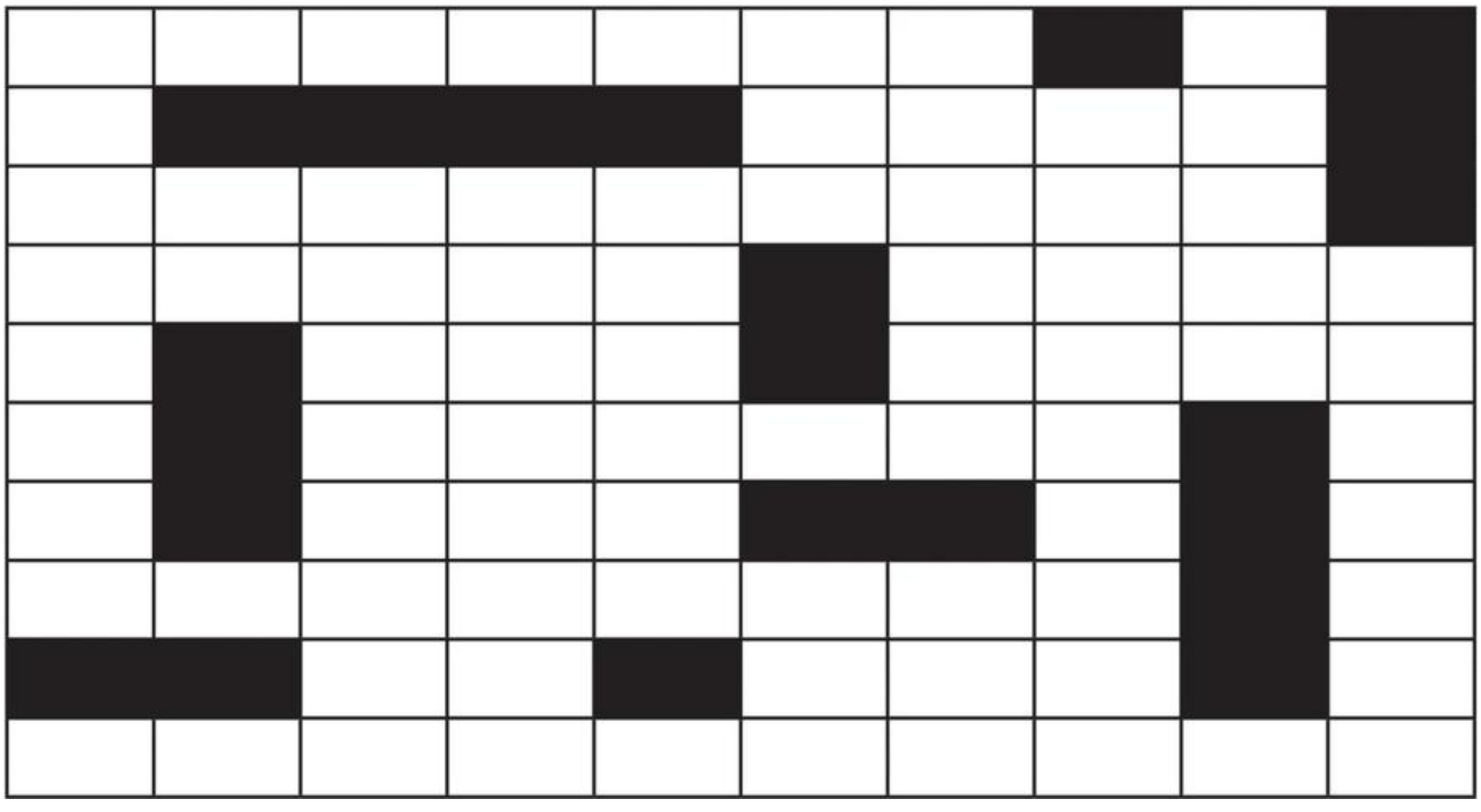
Práctica 4.9: Agrupar alumnos

- Crea una aplicación web que lea nombres de supuestos alumnos (valdrá cualquier texto introducido).
- La introducción de alumnos termina cuando indicamos la palabra **fin.**
- Los nombres no se pueden repetir, si se repite se indica que el nombre está repetido y se pide otro.
- La aplicación agrupará los usuarios de 3 en 3 en grupos aleatorios.
- Si el número de alumnos no es múltiplo de tres los alumnos sobrantes se mueven uno a uno de forma aleatoria en los grupos que indiquemos.
- Ejemplo de resultado:
  - Grupo 1: Adela, Maite, Javier
  - Grupo 2: Pedro, ltziar, Leire, Marco
  - Grupo 3: Josune, Andrés, Xuso
  - Grupo 4: Jordi, Antón, Neus, Marta
- Hay dos grupos de 4 alumnos al haber más alumnos, es importante que los dos alumnos no se coloquen en mismo grupo formando un grupo de 5

Práctica 4.10: Hundir la flota

- Crear un programa que dibuje en pantalla el tablero del juego **Guerra de barcos** (también como Hundir la flota gracias a un producto comercial).
- Tiene que dibujar aleatoriamente en el tablero un barco de tamaño cuatro (portaviones), tres tamaño tres (acorazados), tres de tamaño dos (destructores) y dos de tamaño uno (fragatas).

- El tablero será de 10 por 10.
- Los barcos no se pueden solapar ni tocar y se pueden dibujar tanto en horizontal como en vertical.
- Los barcos se situarán aleatoriamente en el tablero de forma automática por parte de la alicación.
- Ejemplo de resultado posible:



4.9 RESUMEN DE LA UNIDAD

- JavaScript aporta varios tipos de estructuras de datos para poder gestionar datos de forma más cómoda.
- Aunque los strings se consideran habitualmente un tipo de datos simple, realmente tienen posibilidades muy avanzadas para manipular textos.
- Los strings tienen métodos asociados que permiten, entre otras tareas, convertir a mayúsculas el texto, extraer el texto que deseemos, buscar un texto dentro de otro, obtener el tamaño del texto, etc.
- Los arrays permite asociar una colección de datos, incluso heterogéneos, a una misma estructura. El acceso al array se realiza mediante una variable que actúa como referencia del array.
- Para acceder a un elemento individual de un array se utilizan los corchetes y el número de índice del elemento. El primer elemento tiene índice cero, el siguiente el uno y así sucesivamente.
- Podemos asociar un array a una constante ya que aunque modifiquemos el contenido del array, el valor de la constante no cambia puesto que se considera que sigue siendo la referencia del mismo array.
- Los arrays son realmente objetos que poseen métodos ya preparados para realizar tareas sobre ellos como ordenar, devolver el tamaño del array, eliminar elementos, añadir elementos, etc.
- El recorrido de los arrays se puede realizar usando un contador que recorra cada índice a través de un bucle **for** normal. Pero disponemos de bucles más apropiados para recorrer arrays, concretamente **for..in** y **for...of.**
- Los conjuntos, también conocidos como ***sets,*** son una estructura que sirve para almacenar colecciones de datos. A diferencia de los arrays, no se permite repetir el mismo valor en un mismo conjunto.
- El método **add** es el encargado de añadir elementos a un conjunto. También podemos crear un conjunto a partir de un array o incluso convertir un conjunto en un array.
- Los conjuntos también disponen de métodos para añadir elementos, borrar elementos, buscar valores u obtener el tamaño del conjunto.
- El recorrido por los elementos de un conjunto se realiza con el bucle **for..of.**
- Los mapas son una estructura que permite asociar una clave (que no se puede repetir) a un valor.
- La forma de añadir elementos a un mapa es a través del método set. Otros métodos permiten buscar claves y valores, convertir las claves o los valores en arrays o eliminar elementos. También el bucle for..of es el idóneo para recorrer mapa.

4.10 TEST DE REPASO

1.- **¿Qué escribe este código por consola?**

```
consolé.log(typeof "Hola");
```

a) string
b) object
c) text
d) array
e) Nada, ocurre un error

2.- **¿Qué escribe este código por consola?**

```
consolé.log(typeof
    ["Hola","Adiós"]);
```

a) string
b) object
c) text
d) array
e) Nada, ocurre un error

3.- **¿Qué escribe este código por consola?**

```
consolé.log(typeof
   new Set(["Hola","Adios"]));
```

a) string
b) object
c) text
d) set
e) Nada, ocurre un error

4.- **Qué escribe este código por consola?**

```
consolé.log(typeof new
   Map([[1,"Hola"],[2,"Adiós"]]));
```

a) string
b) object
c) text
d) set
e) Nada, ocurre un error

5.- Un mapa...

a) Admite valores duplicados en las claves y en los valores
b) Admite valores duplicados solo en los valores
c) Admite valores duplicados solo en las claves
d) No admite valores duplicados ni en los valores ni en las claves

6.- **¿Qué escribe este código por consola?**

```
consolé.log(
   "España".toUpperCase()
);
```

a) ESPAñA
b) españa
c) España
d) ESPAÑA

7.- **¿En qué tipo de textos JavaScript podemos usar plantillas con expresiones como ${x}?**

a) En cualquier texto
b) Solo en los que se delimiten con comillas dobles
c) Solo en los que se delimiten con comillas simples
d) Solo en los que se delimiten con comillas dobles o simples
e) Solo en los que se delimiten con comillas invertidas

8.- Observa este código:

```
let a=[1,2,3,4];
delete a[0];
consolé.log(a[0])
```

¿Qué sale por consola?

a) 2
b) undefined
c) NaN
d) **1**
e) Nada, ocurre un error

9.- **¿Con qué tipos de datos podemos usar el bucle for..of?**

a) Con arrays
b) Con conjuntos
c) Con mapas
d) Con strings
e) Con todos los anteriores
f) Con todos los anteriores salvo con los string

10.- **Observa este código:**

```
let a=[l,2,3,4];
a.shi ft();
consolé.log(a);
```

¿Qué sale por consola?

a) **[1,2,3]**

b) **[2,3,4]**

c) **[1,2, 3, 4]**

d) **nuil**

11.- **¿Qué método permite fácilmente convertir un array en un string?**

a) typeof

b) to String

c) textify

d) join

e) toText

12.- **¿Cuál es el método que permite añadir elementos en un mapa?**

a) add

b) set

c) new

d) concat

13.- **¿Qué método permite saber si un elemento está en un conjunto?**

a) **get**

b) **has**

c) **indexOf**

d) **search**

14.- **¿Qué método permite saber si un elemento está en un mapa?**

a) **get**

b) **has**

c) **indexOf**

d) **search**

15.- **¿Qué escribe por consola este código?**

```
let c=new Set();
c.add(l).add(2).add(3).add(4);
consolé.log([...c])
```

a) **[1, 2 ,3, 4]**

b) **Set{}**

c) **Set{l, 2, 3, 4}**

d) Nada, ocurre un error

16.- **¿Qué escribe por consola este código?**

```
let array=[”a",”b",”c”,”d”];
for(let h in array)
    consolé.log(h);
```

a) **"a”**
"b"
"c"
"d"

b) **[0,"a"J**
[l,"b"]
[2,"c"]
[3,"d"]

c) **0**
1
2
3

d) Nada, ocurre un error

17.- **Qué escribe por consola este código?**

```
array=[2,3,2,3,2,3,3];
consolé.log(
   array.indexOf(2,-3);
);
```

a) **0**

b) **1**

c) **2**

d) **3**

e) **4**

UNIDAD 5

FUNCIONES

OBJETIVOS

— Valorar la importancia de las funciones en el desarrollo de aplicaciones

— Identificar los elementos y utilidad de los elementos de una función

— Reconocer la sintaxis de creación de funciones

— Crear funciones usando notación declarativa, notación anónima y notación de flecha

— Aplicar el uso de funciones en la creación de aplicaciones web

— Asimilar el funcionamiento de la pila de funciones

— Interpretar el aporte de la recursividad en la resolución de problemas mediante funciones

— Manejar funciones callback tanto para resolver problemas propios como para utilizar métodos de la librería estándar de JavaScript

CONTENIDOS

5.1 INTRODUCCIÓN A LAS FUNCIONES

5.2 CREACIÓN DE FUNCIONES
- 5.2.1 ELEMENTOS DE UNA FUNCIÓN
- 5.2.2 DECLARAR E INVOCAR FUNCIONES
- 5.2.3 ASIGNAR FUNCIONES A VARIABLES
- 5.2.4 FUNCIONES FLECHA

5.3 DETALLES SOBRE VARIABLES Y PARÁMETROS
- 5.3.1 ÁMBITO DE LAS VARIABLES
- 5.3.2 PASO POR VALOR Y PASO POR REFERENCIA
- 5.3.3 ARGUMENTOS CON VALORES POR DEFECTO
- 5.3.4 NÚMERO VARIABLE DE PARÁMETROS

5.4 USO AVANZADO DE FUNCIONES
- 5.4.1 LA PILA DE FUNCIONES
- 5.4.2 RECURSIVIDAD
- 5.4.3 FUNCIONES CALLBACK
- 5.4.4 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS

5.5 PRACTICAS RESUELTAS

5.6 PRÁCTICAS RECOMENDADAS

5.7 RESUMEN DE LA UNIDAD

5.8 TEST DE REPASO

5.1 INTRODUCCIÓN A LAS FUNCIONES

En la programación clásica, las funciones fueron la base de la llamada **programación modular.** La idea de programar aplicaciones de forma modular se basa en el paradigma "divide y vencerás" que, aplicado al desarrollo de aplicaciones, implica en no abordar el problema completo, sino en descomponerlo en problemas más pequeños, más fáciles de solucionar uno a uno.

La programación modular permite la posibilidad de que un programa se divida en un conjunto de módulos cada uno de los cuales se programaba de manera independiente. Cada módulo se encarga de una determinada tarea o función de modo que, a partir de una serie de datos de entrada produce unos resultados. La aplicación completa se convierte en una serie de módulos que se interconectan consiguiendo solventar el problema completo.

Pues bien, en el caso de JavaScript se dispone de funciones y objetos para implementar este paradigma. Indudablemente las funciones son un mecanismo para programar de forma modular, pero en JavaScript las funciones adquieren una nueva dimensión gracias a su versatilidad y a su capacidad de ser usada de forma absolutamente dinámica.

Una función es código JavaScript que realiza una determinada tarea a partir de unos datos de entrada **(parámetros).** Dicha tarea consiste en devolver un determinado resultado a partir de los datos de entrada, o bien, realizar una acción determinada. Por ejemplo, podemos programar una función para que calcule el factorial de un número entero y devuelva dicho factorial. También, podemos crear una función más sofisticada como una, por ejemplo, que se encargue de enviar los datos que la indiquemos a un servidor de base de datos.

Lo fundamental es que la misma función puede ser invocada una y otra vez. Incluso podemos usar las mismas funciones en diferentes aplicaciones. Es más legible un código que usa funciones y, además, facilita la detección de errores, ya que podemos ir mejorando y mejorando la función, para que todas las aplicaciones que la utilicen, vean reflejadas al instante esas mejoras.

Por otro lado, JavaScript es un lenguaje **asincrono** donde las instrucciones se ejecutan sin esperar a que la anterior termine. Eso dificulta ciertas tareas que requieren de un resultado anterior; es decir tareas que requieren un modo **síncrono** de trabajo. JavaScript utiliza las funciones para solucionar este, y otros problemas, gracias a que las funciones son un código que se puede asociar a cualquier variable o parámetro.

En realidad, ya hemos utilizado funciones en las unidades precedentes. Por ejemplo, **parselnt** es una función que hemos utilizado para poder convertir textos en números enteros.

5.2 CREACIÓN DE FUNCIONES

5.2.1 ELEMENTOS DE UNA FUNCIÓN

Normalmente las funciones requieren de los siguientes elementos:

- Un **identificador** (nombre) de la función. Que cumple las mismas reglas que los identificadores de variables. Como convención formal (pero no obligatoria), las funciones se

identifican con nombres en minúsculas. Hay que señalar que en JavaScript, como veremos más adelante, las funciones pueden incluso ser anónimas.

- Uno o más **parámetros** que son variables locales a la función que sirven para almacenar los datos necesarios para que la función realice su labor. Puede haber funciones que no utilicen parámetros.
- Un **resultado** que es un valor (simple o complejo) que se devuelve a través de la instrucción **retum.** Es posible que una función no devuelva un resultado, sino que, realice una determinada acción. Este tipo de funciones, que no usan la instrucción **return,** en otros lenguajes se las conoce como **procedimientos.**
- Las **instrucciones** de la función, que son las sentencias que se ejecutan cuando se invoca a la función. Es el código en sí de la función. Este código se ejecuta cuando invocamos a la función desde cualquier parte del código de la aplicación.

5.2.2 DECLARAR E INVOCAR FUNCIONES

Las funciones deben de ser declaradas antes de que se puedan usar. La sintaxis habitual para declarar una función, es:

```
function nombre([listaParámetros]){
   …cuerpo de la función…
}
```

Hay más maneras (como veremos más adelante) de declarar funciones, pero esta es la más clásica (es similar en casi todos los lenguajes de programación) y se la conoce en JavaScript como **notación declarativa.**

Ejemplo de declaración usando la notación declarativa:

```
function saludo(){
   consolé.log("Hola");
}
```

Esta función no tiene ni parámetros ni retorna ningún valor. Pero sí realiza una acción (aunque muy sencilla), escribe ***Hola*** por consola. Para utilizar esta función debemos invocarla de esta forma:

```
saludo(); //Escribe Hola por consola
```

La forma de invocar a una función es indicar su nombre y después, entre paréntesis, indicar los parámetros de la función. Aunque la función, como ocurre con el anterior ejemplo, no requiera de parámetros, debe de utilizar los paréntesis.

Podemos mejorar esta función si permitimos que use parámetros. En lugar de escribir ***Hola,*** vamos a conseguir que escriba el texto que la enviemos. La declaración de la función será:

```
function saludo(mensaje){
    consolé.log(mensaje); }
```

Ahora podremos invocar de esta forma:

```
saludo("Hasta la vista"); //Escribe Hasta la vista por consola
```

Como hemos comentado, las funciones pueden devolver un resultado,

```
function triple(n){
    return 3*n;
}
```

Esta función recibirá un número al que llamamos n, y retorna el resultado de multiplicar por tres a ese número. Un ejemplo de uso de esta función:

```
let x=6, y=4, z="Hola";
consolé.log(triple(9)); //Escribe 27
consolé.log(triple(x)); //Escribe 18
consolé.log(triple(x+y)); //Escribe 30
consolé.log(triple(x)+triple(y)); //Escribe 30
consolé.log(triple(triple(9))); //Escribe 81
consolé.log(triple(z)); //Escriba NaN
```

Podemos observar cómo al invocar la función, se admite cualquier expresión para indicar sus parámetros. En el código anterior se muestran ejemplos de esta idea:

- En la primera invocación se usa la expresión: **triple(9).** Dentro de la función, el parámetro ***n*** recoge este valor y le devuelve multiplicado por tres.
- En el caso de **tr i p** 1 **e (x),** el parámetro ***n*** recoge el valor de la variable ***x*** que es 6, por eso la función retorna 18.
- La invocación: **triple(x+y),** hace que primero se resuelva la expresión y el valor resultante (en este caso el resultado es ***10)*** se pase como parámetro a la función. Por eso devuelve ***30*** (triple de 10).
- Más llamativo es el código: **triple(x)+triple(y).** En este caso se debe resolver primero el código **triple(x)** que devuelve 18. Después se resuelve la expresión **triple(y)** que, como la variable y vale 4, devuelve el valor 12. Se suman ambos valores (30).
- Incluso un parámetro puede ser el resultado de invocar primero a la función como en: **triple(triple(9)) Lo** primero será resolver la llamada **más** interior **(triple(9))** que, en este caso, retorna el valor 27. Este valor se pasa como parámetro a la función que lo devolverá multiplicado por tres, es decir 81.
- Lógicamente como la variable ***z*** es un string. La invocación: **triple(z)** intentará multiplicar por tres ese texto, produciendo el valor no numérico **NaN.**

La función ***triple*** del código anterior es tan trivial que, evidentemente, no es necesario crear una función así. Se ha usado un ejemplo tan sencillo para facilitar la explicación sobre cómo crear y utilizar funciones.

Esta otra función sirve para contar la cantidad de números pares que hay en un array. No es ya tan trivial, es una función más elaborada.

```
function pares(array){
        let nPares=0;
        if(array instanceof Array){
                for(n of array){
                        if(n%2===0){
                                nPares++;
                        }
                }
        }
        return    nPares;
}
```

Un posible uso sería:

```
consolé.log(pares([1,2,3,4,5,6,7,8,9])); //Escribe 4
```

El resultado indica que hay cuatro números pares en el array que se pasa a la función.

Hay que tener en cuenta que las funciones se pueden invocar en cualquier parte, por ejemplo, al indicar los valores de un array:

```
let array=[triple(l),triple(2),triple(3)];
consolé.log(array);
```

El resultado será:

```
[  3, 6, 9      ]
```

Las expresiones, siempre que sean válidas, ahora pueden ser muy complejas, aunque, a la vez, simplifiquen mucho nuestro trabajo:

```
consolé.log(pares([triple(l),triple(2),triple(3)])); //Escribe 1
```

Solo el triple de dos es par, por eso el resultado de esa expresión es el número 1.

ACTIVIDAD 5.1: PRIMERA FUNCIÓN

- En este punto conviene realizar la Práctica 5.1 ***"Función de pares e impares",*** en la página 190, es muy sencilla y facilita entender en qué nos ayudan las funciones.

5.2.3 ASIGNAR FUNCIONES A VARIABLES

Hay un tipo especial de función en JavaScript, se llama **función anónima** y nos lleva de pleno a la forma que tiene JavaScript de entender las funciones.

Como hemos comentado anteriormente, realmente las funciones son, simplemente, un código que se puede invocar una y otra vez. El acceso a ese código se puede hacer, como hemos visto, con el nombre de la función. Pero en JavaScript no es la única manera de acceder a dicho

código. Cualquier mecanismo de javaScript que permita acceder a ese código es válido. Por ello, este código es correcto:

```
const trip=function(x){
    return 3*x;
}
```

Si nos concentramos solo en el código tras la palabra **function** veremos un código que, recibiendo un parámetro que hemos llamado x, lo devuelve multiplicado por tres. La novedad es que esa función no tiene nombre. La función es anónima, pero el código de la misma es accesible porque se le estamos asignando a la variable ***trip.*** Aunque ***trip*** es una variable, no es una función (aunque nos parezca un pequeño matiz, es importante), puede acceder al código de la función de la misma forma que si hubiera puesto ese nombre a la función:

```
consolé. log(trip(3)); //Escribe 9
```

Es decir, la variable es una referencia a la función. En JavaScript tenemos, como en muchos lenguajes, referencias a arrays y a otros objetos, y también, algo que ocurre en menos lenguajes, referencias a funciones.

El hecho de declarar ***trip*** como constante **(const)** tiene sentido si esa variable siempre se asocia con la función a la que se asigna en la declaración, si asignamos otra función ocurriría un error (porque la referencia cambia).

Es posible incluso que dos variables hagan referencia a la misma función. Si añadimos este código al anterior:

```
let x=trip;
consolé.log(x(8)); //Escribe 24
```

Ambas variables ***(trip*** y x) utilizan la misma función.

5.2.4 FUNCIONES FLECHA

Hay otra manera de declarar funciones que se ha convertido en muy popular debido a su facilidad de escritura. Solo sirve para funciones anónimas y consiste en que no aparece la palabra **function** y en que una flecha separa los parámetros del cuerpo de la función. Ejemplo:

```
const triple=x=>3*x;
consolé.log(triple(20));
```

La definición de la función anónima es la sorprendente expresión:

```
x=>3*x
```

Que resulta ser equivalente a:

```
function(x){
   return 3*x;
}
```

El símbolo de la flecha separa los argumentos del cuerpo de la función, en el que, además, se sobrentiende la palabra **return.**

Evidentemente, es una notación para escribir más rápido. Si hay más de un parámetro, se deben colocar entre paréntesis. Ejemplo:

```
const media=(x,y)=>(x+y)/2;
consolé. log(media(10,20)); //Escri be 15
```

Esta función es un poco más compleja y requiere que los dos parámetros que utiliza la función estén entre paréntesis, si no la expresión fallaría. Por otro lado, si el cuerpo de la función es más complejo, requiere ser incluido entre llaves:

```
const sumatorio = (n)=>{
        let acu=0;
        for(let i=n;i>0;i—){
                acu+=i;
        }
        return acu;
}
consolé.log(sumatorio(3)); //Escribe 6, resultado de 3+2+1
```

También son necesarias las llaves cuando no hay **return:**

```
const saludo = mensaje=>{
    consolé.log(mensaje);
}
saludo("Hola");
```

Si en la función no hay parámetros, hay que colocar paréntesis vacíos en la posición que ocuparían los parámetros:

```
const hola = ()=>{
    consolé.log("Hola");
}
hola(); //Escribe Hola
```

Pero hay que tener en cuenta que, ante funciones complejas, las ventajas de las funciones flecha se diluyen:

```
const  pares=(array)=>{
        let nPares=0;
        if(array instanceof Array){
                for(n of array){
                        if(n%2===0){
                                nPares++;
                        }
                }
        }
        return nPares;
}
```

No parece un código que ahorre mucho respecto a forma clásica de escribir funciones anónimas:

```
const  pares= function(array){
        let nPares=0;
        if(array instanceof Array){
                for(n of array){
                        if(n%2===0){
                                nPares++;
                        }
                }
        }
        return nPares;
}
```

De ahí que lo habitual es que los programadores solo usen funciones flecha para definir funciones sencillas.

ACTIVIDAD 5.2: CREAR UNA FUNCIÓN FLECHA

- Crea una función flecha que sirva para devolver verdadero si un número es par y falso si no lo es.
- Asigna esa función a una variable y prueba la función para ver que responde correctamente.

5.3 DETALLES SOBRE VARIABLES Y PARÁMETROS

5.3.1 ÁMBITO DE LAS VARIABLES

Ya hemos comentado anteriormente (3.3.2.1 ***"Diferencia entre let y var",*** en la página 73) que las variables tienen una duración en el código dependiendo de cómo se han declarado. Sin embargo, es en este punto en el que podemos aclarar más los aspectos sobre el ámbito de uso de las variables. Las variables definidas en una función tanto con **const,** como con **let,** como con **var** no se pueden usar fuera de la función en la que se declaran:

```
function f(){
        const a=9;
        let b=9;
        var c=9;
        consolé.log("soy la función f");
}
f();
consolé.log(a); //error
consolé.log(b); //error
consolé.log(c); //error
```

Aunque hemos invocado a la función mediante f () lo cierto es que las variables ***a, b*** y ***c*** no se pueden utilizar fuera de la función.

Veamos este otro código:

```
function f(){
   i f(true){
      const a=9;
      let b=9;
      var c=9;
   }
   consolé.log(a);          //error
   consolé.log(b);          //error
   consolé.log(c);          //i ¡Esta sería correcta!!
}
f();
```

Podemos utilizar la variable c fuera de la estructura ***if*** aunque se declaró en ella. Sin embargo, las dos líneas anteriores provocan un error ya que las variables definidas con **const** y **let** no pueden usarse fuera del bloque en el que fueron definidas, en este caso no se pueden usar fuera del ***if.***

En cuanto a los parámetros, tampoco pueden usarse fuera de la función en la que se definen:

```
function g(x){
      x=19;
}
g(8);
consolé.log(x); //Error, x no se puede usar fuera de la función
```

Se pueden usar solamente en la función. Es decir, su ámbito es el mismo que el de las variables declaradas mediante la palabra ***var.***

5.3.2 PASO POR VALOR Y PASO POR REFERENCIA

Este título de apartado le será muy familiar a todos los programadores de lenguajes clásicos. Pero para los nuevos programadores vamos a iniciar este apartado con un ejemplo:

```
var x=19;
function f(x){
      x++;
}
f(x);
consolé.log(x);
```

Este código es un poco enrevesado, se ha forzado a que sea así para explicar un detalle muy importante. El código parece que anima a pensar que cuando se escribe en pantalla el valor de ***x*** aparece el número 20. Sin embargo, aparece el 19.

Para entender las razones de este efecto veamos cómo se interpreta realmente este código:

- Inicialmente se crea una variable llamada ***x*** que como se declara con la palabra **var** y está declarada fuera de toda función, su ámbito es todo el código.
- La función/*se declara como una función que tiene un parámetro que también se llama ***x.*** Pero esta ***x*** no es la ***x*** del punto anterior. Son variables diferentes, aunque tengan el mismo nombre. De hecho, por culpa de esa coincidencia, la función no puede acceder a la variable ***x*** declarada en la primera línea.
- Tras la llave de cierre de la función se invoca a la función/pasando el valor de lax que se declaró fuera de la función. Es decir, se pasa el número 19 a la función.
- Ese valor se copia al parámetro ***x*** de la función.
- Después se ejecuta el código de la función que modifica el valor del parámetro ***x*** ya que incrementa su valor. Valdrá 20. Pero la variable ***x*** original no se ha modificado.
- Cuando termina la función, el parámetro ***x*** se elimina. Perderemos su valor que era 20.
- Cuando ***console.log*** escribe el valor de x, la variable a la que se refiere es la global, la que se declaró en la primera línea que sigue valiendo 19.

Indiscutiblemente es un poco enrevesada la explicación. Pero la moraleja es muy sencilla, si pasamos una variable a una función como parámetro, se recoge una **copia** de su valor. La variable original no se modifica. Por eso, lo lógico es que los parámetros no se llamen igual que las variables que se usan para pasar su valor. Sería más lógico este código:

```
var x=19;
function f(y){
        y=20;
}
f(x);
consolé.log(x);
```

Aún caben más tentaciones:

```
var x=19;
function f(){
        x=20;
}
f();
consolé.log(x);
```

Ahora sí aparece 20. Se ha modificado la variable x original dentro de la función. No hay ambigüedad porque la función no declara ningún parámetro o variable interna con ese mismo nombre.

No obstante, **no es recomendable usar variables globales dentro de las funciones.** La modularidad que aportan las funciones se pierde si hacemos uso de esta técnica, ya que la función no

sería transportable a otro archivo. Las funciones deben de crearse de la forma más independiente posible respecto al código que las rodea.

Para complicar más el asunto veamos este otro código:

```
var array=[1,2,3,4,5];
function g(a){
        a[0]=9;
}
g(array);
consolé.log(array[0]);
```

Tras lo explicando antes, ahora veremos, seguramente, con sorpresa que se escribe un 9 y no un ***1.*** Es decir, la función ha modificado el valor del array original. Nuevamente, para saber por qué ocurre este hecho, veremos el procesamiento de este código paso a paso:

- En la primera línea se crea una variable global llamada ***array*** que es una referencia a un array que almacena los valores ***1,2,3, 4 y 5.***
- Se declara una función llamada ***g*** que tiene un parámetro llamado ***a.*** El cuerpo de la función sirve para modificar el primer elemento de ***a,*** ya que da por hecho que ese parámetro es un array, y le otorga el valor 9.
- Se invoca la función ***g*** pasando el array original. El parámetro ***a*** recogerá una referencia a ese array. Esta vez no se recibe una copia, sino una referencia al array original. Los arrays no se copian cuando se asignan. Es decir, ***array y a*** son una referencia al mismo array.
- Dentro de la función se modifica el primer elemento de ***a*** para que valga 9. Eso es lo mismo que modificar el primer elemento de la variable ***array.***
- La función termina, el parámetro ***a*** ya no estará disponible.
- Se escribe el primer elemento de la variable ***array y*** comprobaremos que dentro de la función se ha modificado realmente su valor.

Moraleja final: los tipos básicos (booleanos, números y strings) se pasan por valor, se envía una copia de su valor a los parámetros de las funciones, los tipos complejos: arrays, conjuntos, mapas,.... en definitiva cualquier objeto, pasan una referencia al objeto original; si en la función se modifica el parámetro relacionado, se modificará realmente el array original.

En definitiva, los datos simples (strings, números y valores booleanos) se pasan por valor, los objetos se pasan por referencia.

5.3.3 ARGUMENTOS CON VALORES POR DEFECTO

En JavaScript, los parámetros pueden tener un valor predeterminado. Eso convierte a dicho parámetro en opcional: es decir, podremos enviar o no valores para ese parámetro.

Ejemplo:

```
function saludo(texto="Hola"){
        console.1og(texto);
}
saludo();
saludo("Buenos días");
```

Por pantalla aparece:

```
Hola
Buenos días
```

La primer invocación a la función es: **sa 1 udo()** lo que hace que el único parámetro de la función ***(texto)*** tome el valor ***"Hola"*** que es lo que aparece por pantalla.

Las funciones pueden utilizar tantos parámetros por defecto como se desee.

5.3.4 NÚMERO VARIABLE DE PARÁMETROS

Para entender este apartado, vamos a ver este ejemplo:

```
function media(x,y){
        return (x+y)/2;
}
consolé.log(media(10,20));
consolé.log(media(10,20,30));
```

El resultado de este código es:

```
15
15
```

La primera invocación **(media(10,20))** hace que la función use el valor 10 para el parámetro ***x*** y 20 para el parámetro y. Pero en la segunda se pasa un tercer número. No hay error, pero ese tercer número simplemente es ignorado.

Lo interesante es que podemos pasar tantos valores como nos apetezca, lo que permite crear funciones con un número variable de parámetros. El problema es cómo recoger esos valores. Para eso nos sirve el operador de propagación, del que ya hemos hablado en otros apartados (véase 4.4.9 ***"Desestructuración de arrays",*** en la página 140). Este operador, utilizado en los parámetros de una función, permite almacenar una serie indefinida de parámetros en un array. Veamos una idea de cómo funciona:

```
function f(x,y,...mas){
        consolé.log('x=${x} y=${y} mas=${mas}');
}

f(10,20);
```

```
f(10,20,30) ;
f(10,20,30,40);
Resultado:
x=10 y=20     mas=
x=10 y=20     mas=30
x=10 y=20     mas=30,40
```

En la primera línea se resuelve la invocación f(10,20) el parámetro ***mas*** se queda sin resolver (será vacío). En la segunda línea vemos el valor 30 asociado al parámetro ***mas.*** En la tercera línea observamos que son ambos números 30 y 40 los que se asocian este parámetro.

En resumen, lo que hace el operador de propagación en este contexto es convertir una lista de parámetros en un array. Esto permite revisar nuestra función para el cálculo de la media, de modo que pueda utilizar cualquier número de argumentos. Deberemos resolverla pensando que lo que el usuario envía es un array de números (aunque no lo sea). El cálculo de la media sumará los elementos del array y los dividirá entre el número de elementos del array para calcular la media:

```
function media(... números){;
        let acu=0;
        for(let n of números){
                acu+=n;
        }
        return acu/numeros.length
}
consolé.log(media(10,20));
consolé.log(media(10,20,30));
consolé.log(media(10,20,30,40));
consolé.log(media(10,20,30,40,50));
```

Como vemos, las invocaciones a la función usan los números que queramos. Si la función está bien resuelta, la media siempre es correcta. En este caso, lo es a tenor de los resultados:

```
15
20
25
30
```

Lo que no podremos hacer es enviar un array a esta función:

```
consolé.log(media([10,20,30,40,50])); //Escribe NaN
```

Habría que hacer otra función si queremos pasar los números de esa forma o bien modificar el código de la función para que acepte esta posibilidad.

5.4 USO AVANZADO DE FUNCIONES

5.4.1 LA PILA DE FUNCIONES

Cuando se invoca a una función en una expresión, esta debe esperar a que la función finalice para poder completar la expresión. Ejemplo:

```
function f1(){
        consolé.log("Inicio fl");
        f2();
        consolé.log("Fin fl");
}

function f2(){
        consolé.log("Inicio f2");
        f3();
        consolé.log("Fin f2");
}
function f3(){
        consolé.log("En f3");
}

fi();
```

El resultado es:

```
Inicio fl
Inicio f2
En f3
Fin f2
Fin fl
```

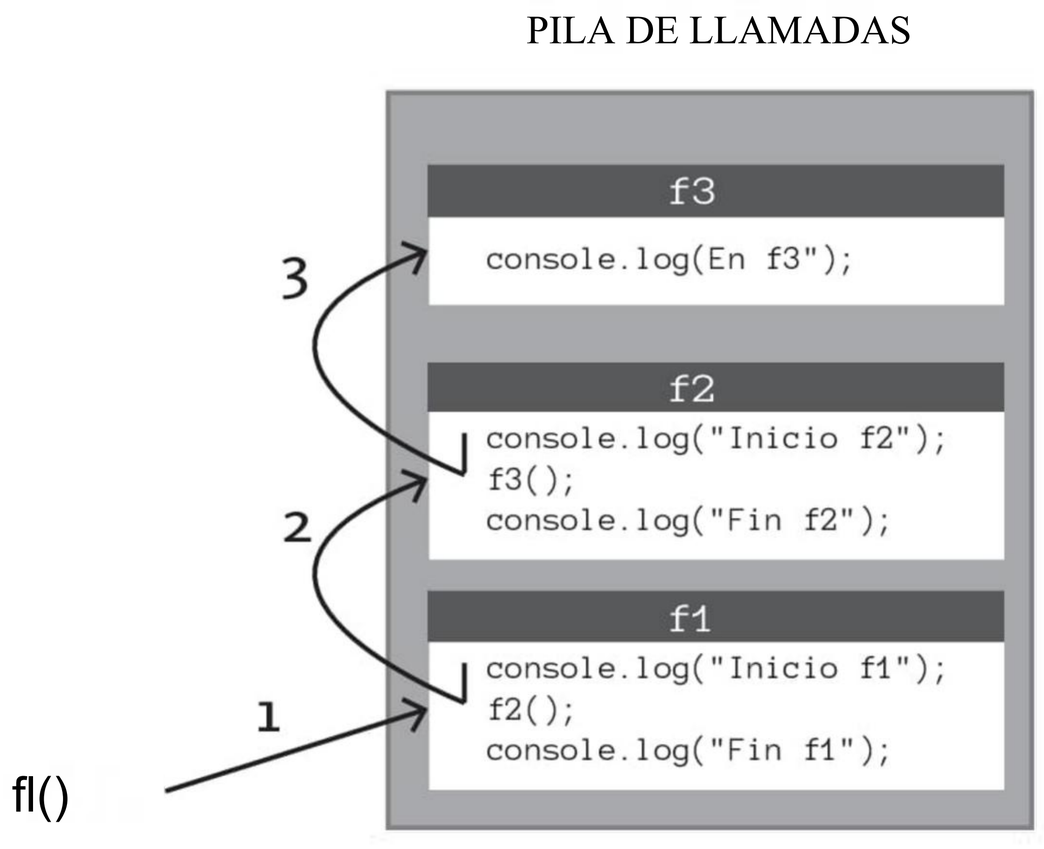


Figura 5.1: Ejemplo de pila de llamadas

Se observa claramente que la primera función (fl) no finaliza hasta que las otras llamadas han finalizado. Es decir, las funciones utilizan lo que se conoce como **pila de llamadas** que permite que, el intérprete JavaScript sepa qué funciones se deben de resolver antes.

En la Figura 5.1, podemos observar como se apilan las funciones en la pila de llamadas. La última función invocada queda en la cima. A medida que se resuelva el código de las últimas funciones se irán retirando de la pila a la vez que se devuelve el flujo a la función anterior.

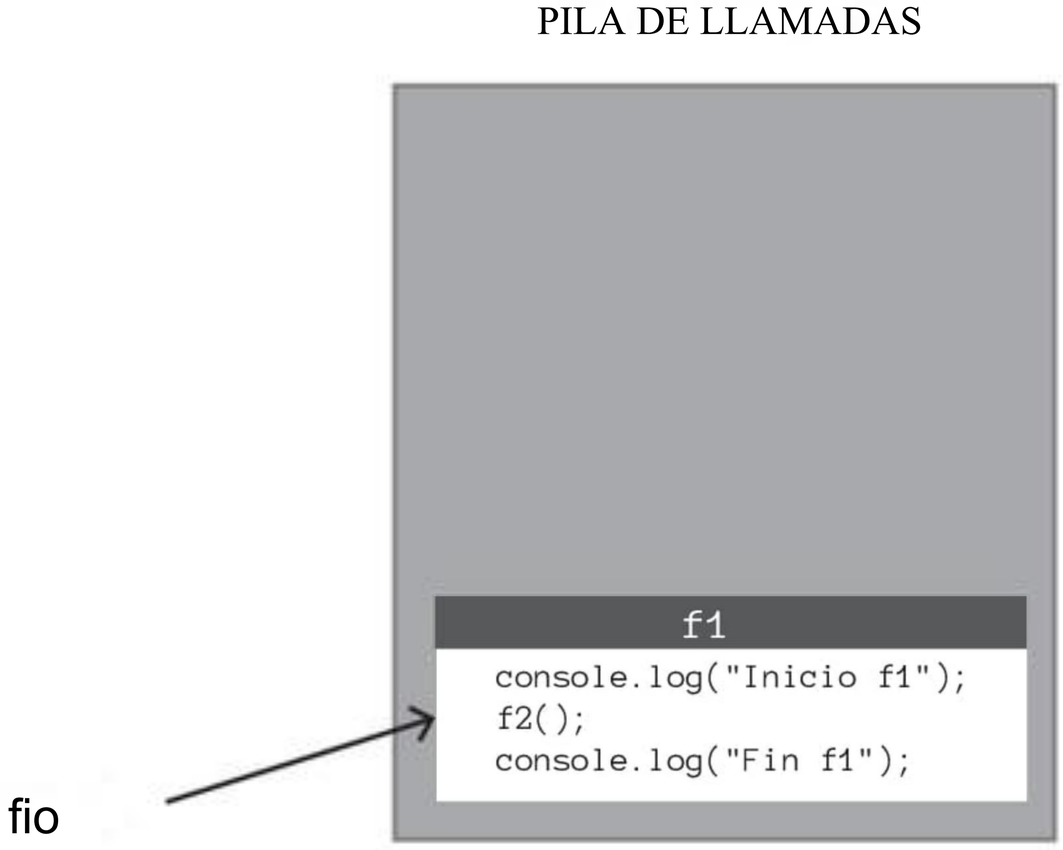


Figura 5.2: La función fl se coloca en la pila de llamadas

En el código anterior el proceso sería el siguiente:

[1] Se invoca a **fl.** El código de esta función se coloca en la pila de llamadas (Figura 5.2).

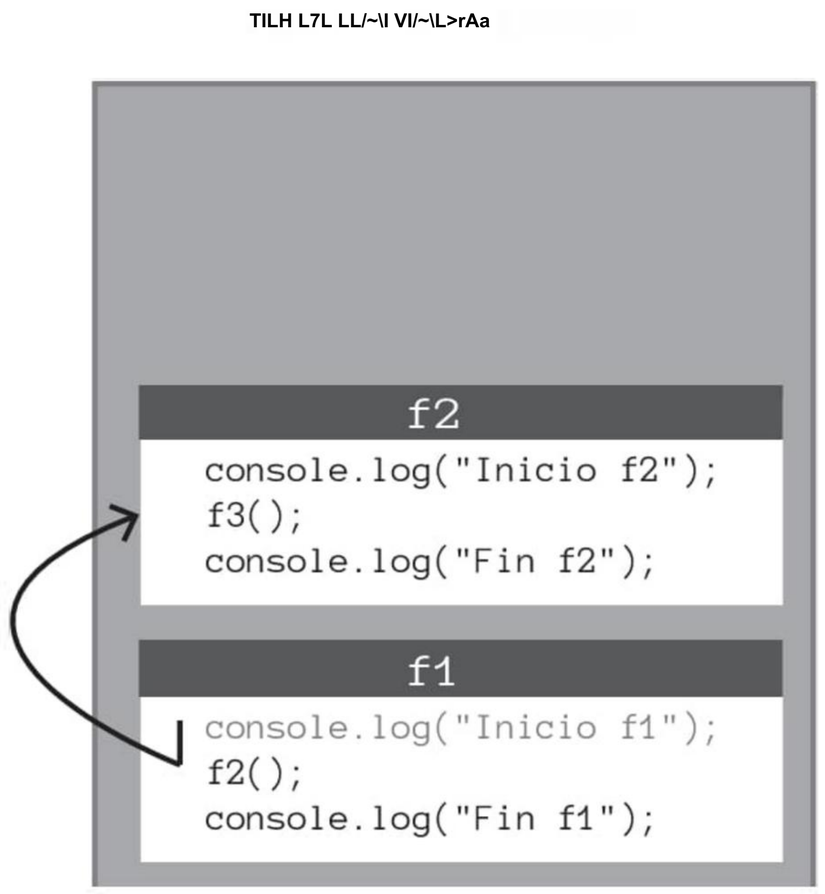


Figura 5.3: La función f2 se coloca en la pila de llamadas

[2] Se interpreta el código de fl. Se ejecuta la escritura del texto ***Inicio fl.***

[3] Se invoca a **f2.** Este código se pone en la cima de la pila (Figura 5.3) Se queda fl a la espera de que se resuelva f2.

[4] Se escribe ***Inicio J2.***

[5] Se invoca a £?. Su código pasa a ocupar la cima de la pila. f2 se queda esperando que se resuelva el código de £$.

PILA DE LLAMADAS

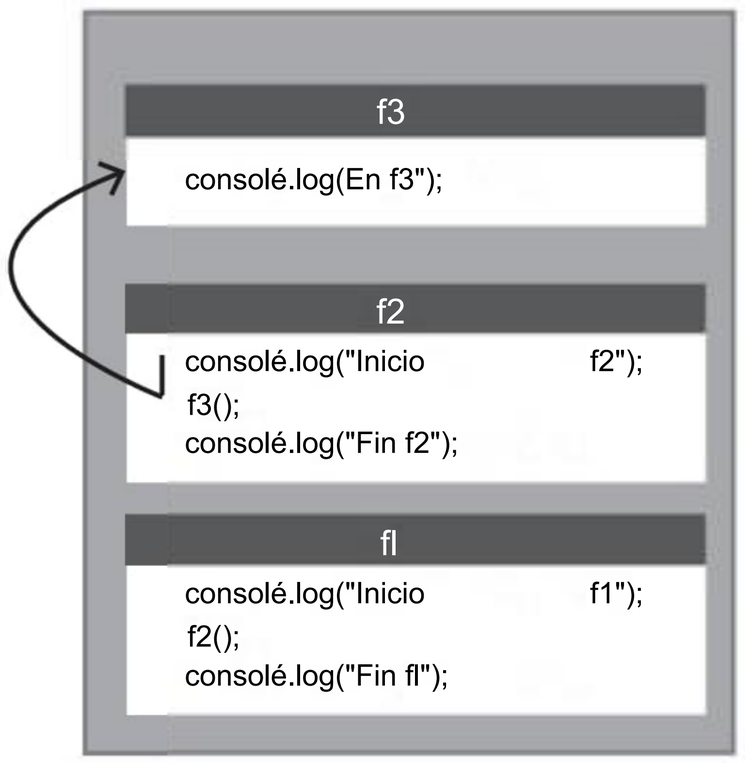


Figura 5.4: La función f3 se coloca en la pila de llamadas

[6] Se escribe ***En f3.***

[7] La función f3 finaliza, devuelve el control a **£2** y f3 se retira de la pila.

PILA DE LLAMADAS

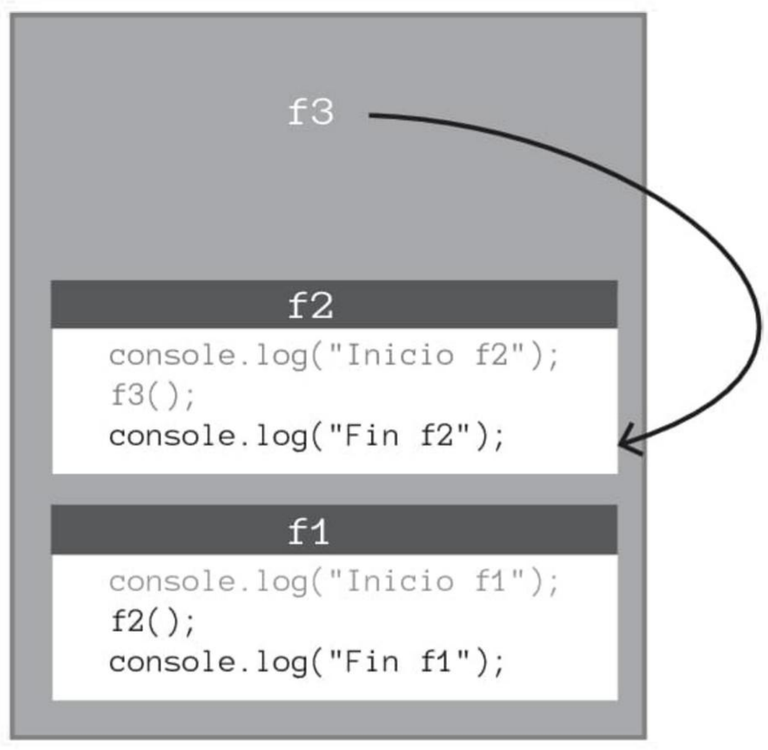


Figura 5.5: La función f3 finaliza y devuelve el control

[8] f2 recupera el control y escribe el mensaje ***Fin f2.***

[9] Como f2 ha finalizado, devuelve el control a fl y se retira de la pila.

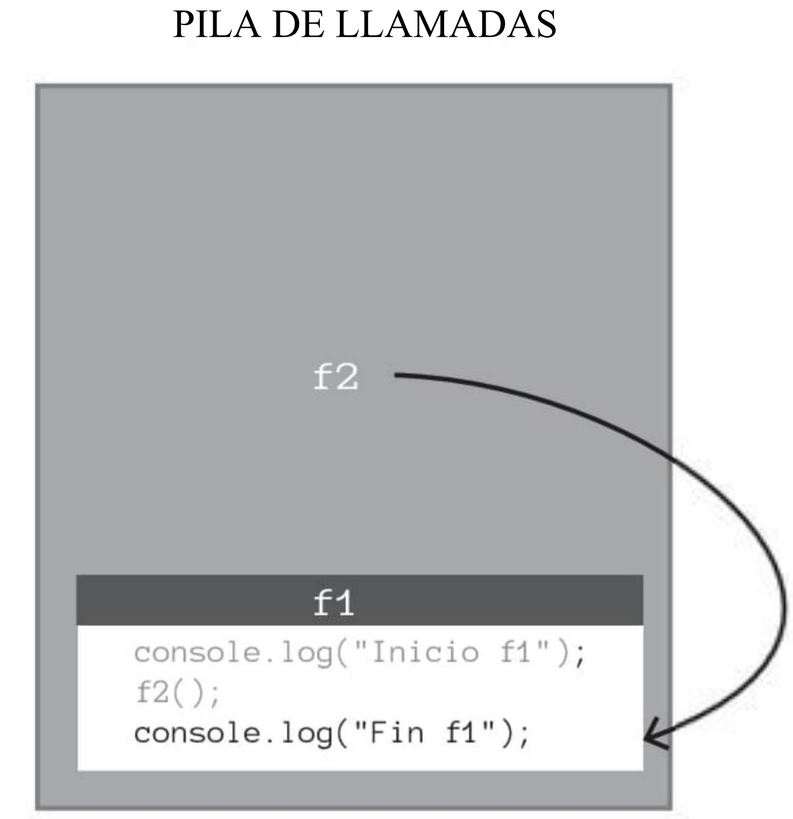


Figura 5.6: La función f2 finaliza y devuelve el control

[10] fl recupera el control y escribe **Fin f2.**

[11] fl se retira de la pila, el control se devuelve a la función principal

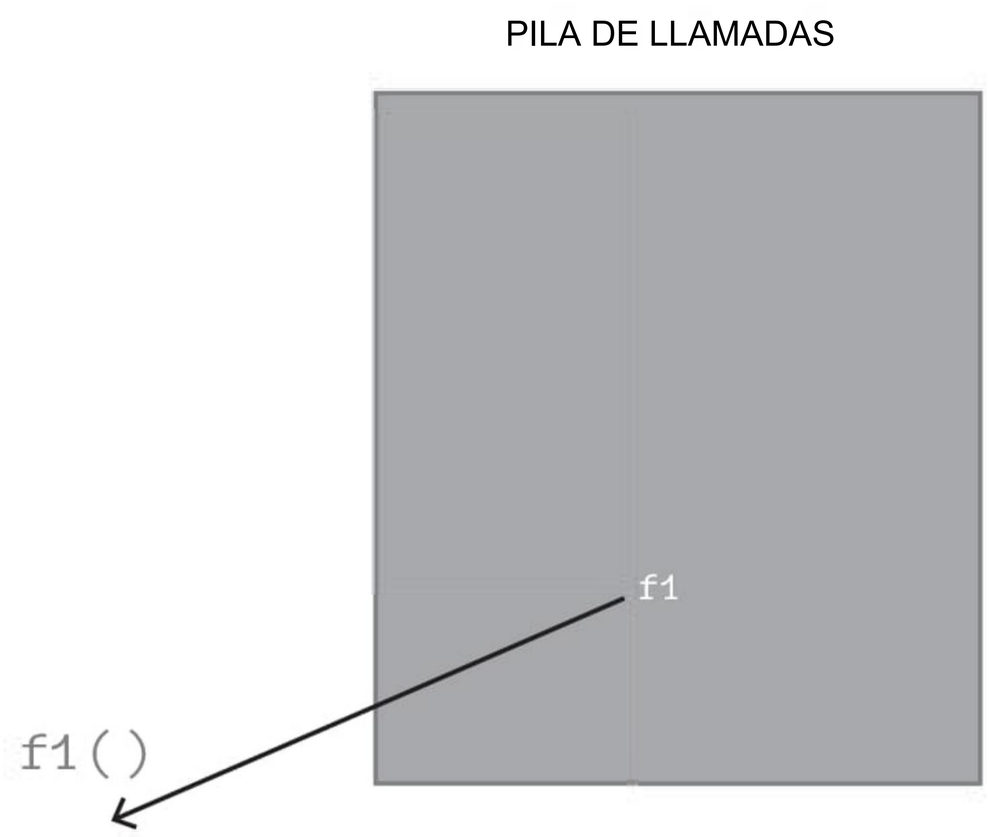


Figura 5.7: La función fl finaliza y devuelve el control

Ante una mala gestión de llamadas se puede llegar a provocar el famoso **desbordamiento de pila.** Veamos un ejemplo de código que lo produciría:

```
function saludo(){
        consolé.log("Saludo");
        despedida();
}
function despedida(){
        consolé.log("Despedida");
```

```
        saludo();
}
saludo();
```

Lo que ocurrirá es que de manera indefinida se llamarían una función a la otra en una especie de bucle infinito. Pero todos los motores de JavaScript (tanto node.js como los navegadores, por ejemplo) cortarán la ejecución del código tras una serie de llamadas debido a que se detecta que la pila de llamadas se va a llenar. La pila tiene un tamaño máximo y eso protege de problemas mayores, sin ese tope el código anterior provocaría efectos más devastadores.

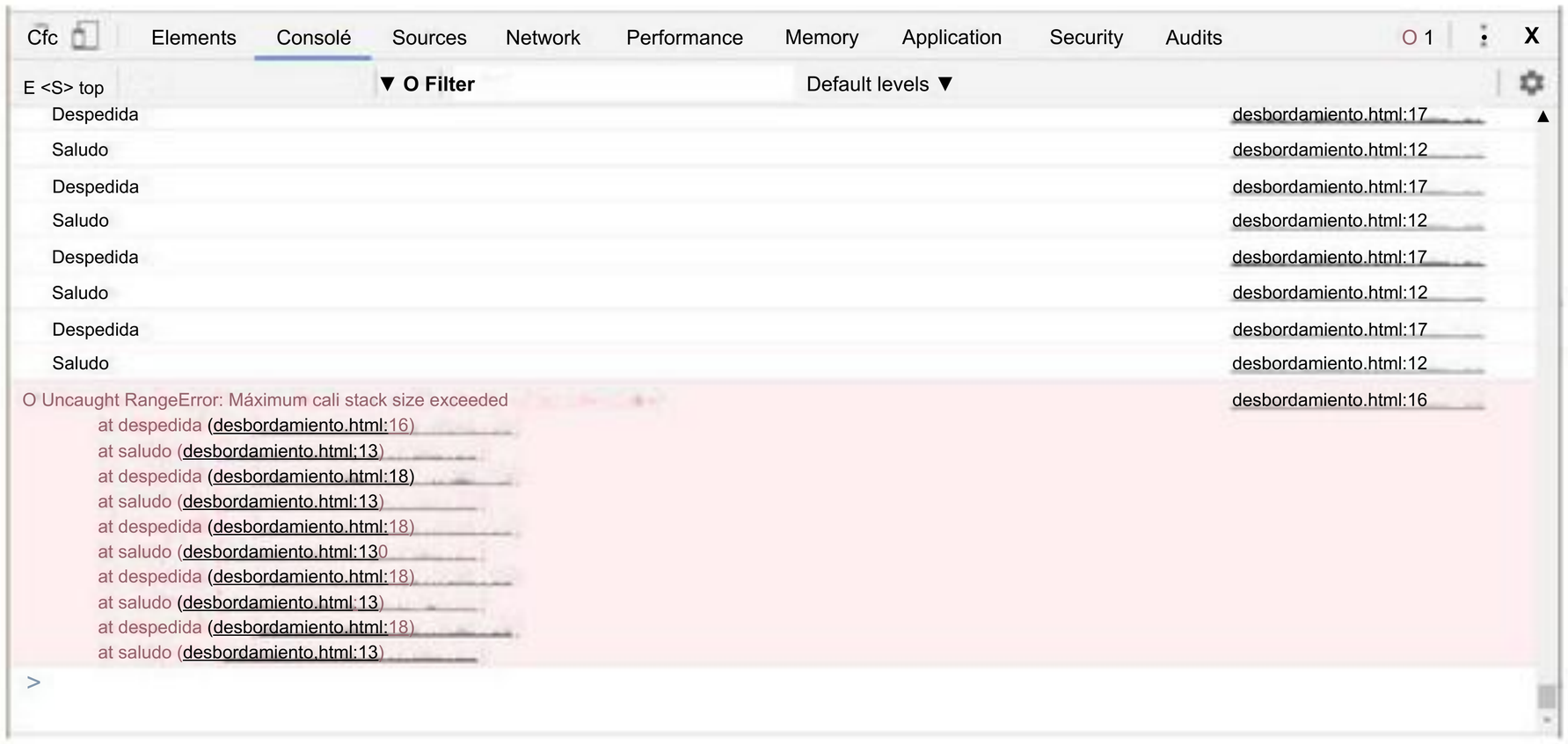


Figura 5.8: Ventana de depuración de Google Chrome informando del desbordamiento de pila que provoca el código anterior

5.4.2 RECURSIVIDAD

Hay una técnica de resolución de problemas complejos que se basa en la capacidad que tienen las funciones de invocarse a sí mismas. La pila de llamadas de JavaScript admite esta posibilidad y lo que ocurrirá es que aparecerá varias veces el código de la misma función en la pila.

Aunque esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas y propiciar un desbordamiento de pila, lo cierto es que es una técnica muy interesante que permite soluciones muy originales y que facilita la realización de aplicaciones sencillas para solucionar problemas muy complejos.

La idea es que cada invocación a la función resuelva parte del problema y se llame a sí misma para resolver la parte que queda del problema, y así sucesivamente. En cada llamada el problema debe ser cada vez más sencillo hasta llegar a una llamada, en la que la función devuelve un único valor. Es fundamental preparar bien esa última llamada ya que es la que cierra el bucle de llamadas ***y*** tras ella se irán resolviendo las anteriores hasta liberar la pila y conseguir el resultado final.

La recursividad se entiende mejor con ejemplos. En este sentido hay un ejemplo clásico: se trata del factorial. El factorial de un número entero se calcula multiplicando todos los números enteros anteriores. Así el factorial de 5 (que se denota matemáticamente como 5!) es el resultado

de 5-4-3-2-1. Pero podemos entender también que 5! es lo mismo que 5-4! (cinco por el factorial de cuatro) y eso permite una solución muy creativa al problema:

```
funetion factorial(n){
    if(n<=l) return 1;
    else return n*factorial(n-1);
}
```

La última instrucción **return n * factor i a1(n-1)** es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al uno, que devuelve uno.

Así si invocamos a esta función mediante el código: **factorial(4),** la ejecución del programa generaría los siguientes pasos:

[1] Se llama a la función factorial usando como parámetro el número 4 que será copiado en el parámetro ***n.***

[2] Se comprueba si ***n*** es mayor que uno. Como lo es, entonces se devuelve ***4*** multiplicado por el resultado de la llamada **factorial (3).** Hemos, por lo tanto, de resolver el factorial de ***3.***

[3] La invocación anterior añade un nuevo código a la pila en el que el parámetro ***n*** vale ***3,*** por lo que esta llamada devolverá ***3*** multiplicado por el resultado de la invocación a **factorial(2).**

[4] Se añade el código de esta función (con ***n*** valiendo 2) en la pila de llamadas. Y se ejecuta el código que intenta devolver 2 multiplicado por el resultado de la **factorial (1).**

[5] Habrá una nueva función en la pila en la que el parámetro ***n*** vale ***1.*** En este caso el código de esta invocación devuelve directamente 1.

[6] Se retirará esa función y el flujo pasa a la función del paso 4, la cual ya puede devolver: **2*factorial(l)** ya que ahora sabemos el resultado.

[7] La función devuelve el número 2 y el control se le queda la función del paso 3. Esta función se retira de la pila.

[8] Ahora podremos retornar el resultado de **3*factorial**(2), que será 6. Se devuelve el resultado a la función inicial y la actual se quita de la pila.

[9] Ahora ya podremos devolver el resultado de **4*factorial** (3), será 24. Esta función también se retira de la pila.

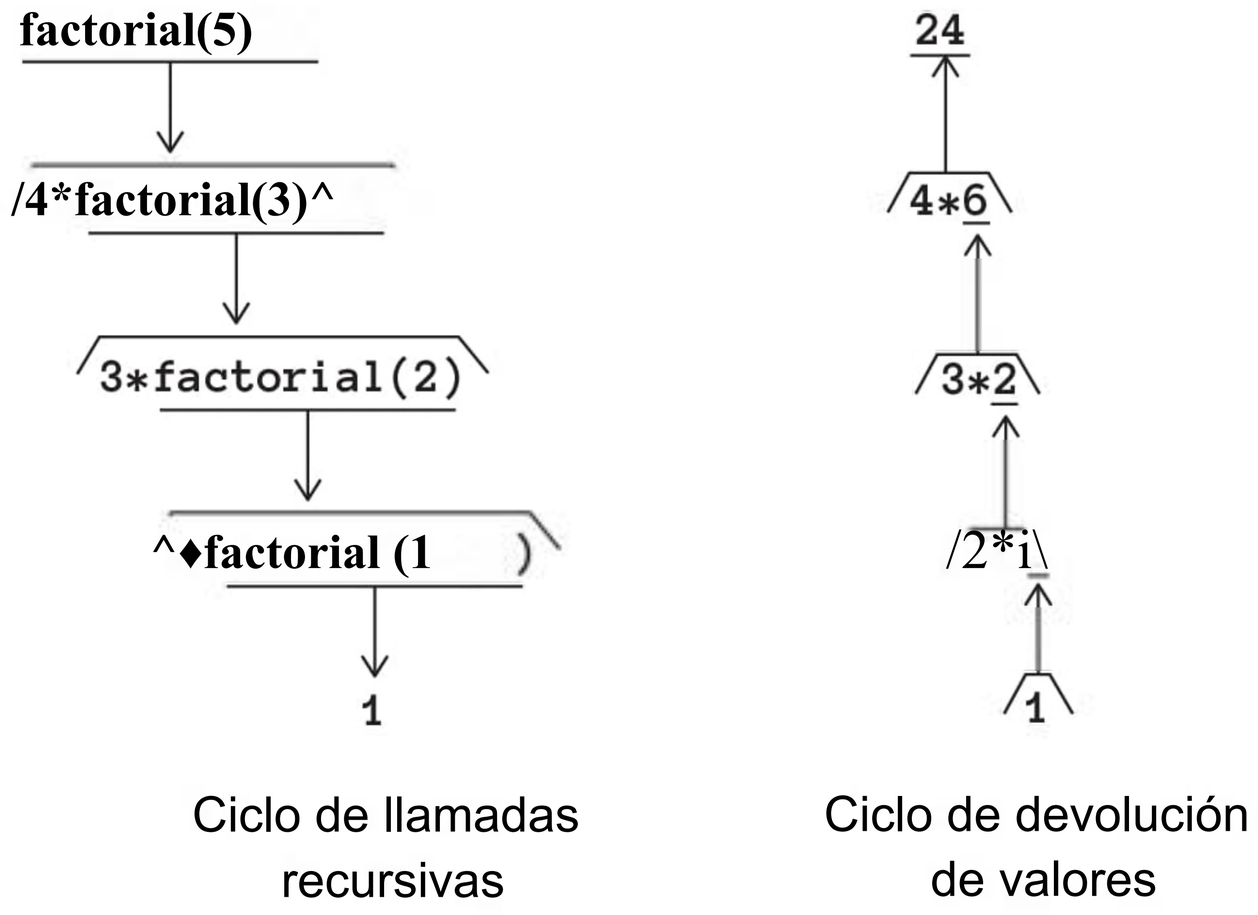


Figura 5.9: Reproducción de las llamadas recursividad en el ejemplo del factorial

---

ACTIVIDAD 5.3: RECURSIVIDAD

- La Práctica 5.6, permite practicar con la recursividad mediante la función de Fibonacci. Es una buena práctica para aplicar la recursividad y compararla con las soluciones iterativas.

5.4.2.1 ¿RECURSIVIDAD O ITERACIÓN?

Hay otra versión de la función factorial resuelta mediante un bucle ***for*** (solución iterativa) en lugar de utilizar la recursividad. Se trataría de esta:

```
function factorial(n){
        let res=l;;
        while(n>l){
            res*=n;
            n--;
        }
        return res;
}
```

La cuestión es ¿cuál es mejor?

Ambas implican ejecutar sentencias de forma repetitiva hasta llegar a una determinada condición que cierra el ciclo de repeticiones. En el caso de la solución iterativa es un contador el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta conseguir una invocación a la función que devuelva un valor sencillo.

En términos de rendimiento es más costosa la recursividad, ya que implica realizar muchas llamadas a funciones en cada cual se genera una copia del código de la misma, lo que sobrecarga

la memoria del ordenador y tiene una forma de ejecución más lenta. Es decir, es más rápida y menos voluminosa la solución iterativa.

¿De qué sirve la recursividad, entonces? Hay una realidad: si poseemos la solución iterativa a un problema, no deberíamos utilizar la recursividad. La recursividad se debería utilizar solamente si:

- No encontramos la solución iterativa a un problema.
- El código es mucho más claro en su versión recursiva y no implica mucha diferencia a nivel de rendimiento sobre la solución iterativa.

En todo caso, al final todo depende de la habilidad del programador ante un mismo problema, hay soluciones iterativas mucho más lentas que una solución recursiva, pero esto normalmente se debe a una mala habilidad del programador.

5.4.3 FUNCIONES CALLBACK

Si hay una característica de JavaScript que distingue mucho a este lenguaje de otros, es el manejo de las llamadas funciones **callback.** Han propiciado una forma de trabajar muy especial y facilitado el entendimiento, como veremos más adelante, de que JavaScript es un lenguaje asincrono y basado en eventos.

La idea en realidad es muy simple: si las funciones se pueden asignar a variables, también se pueden asignar a parámetros de las funciones. ¿Qué permite esta posibilidad? Conseguir que las funciones ejecuten otras funciones a través de los parámetros, es decir: las funciones pueden recibir datos y acciones a realizar. Veamos un ejemplo:

```
function escribe(dato,función){
        funcion(dato);
}
escribe("Hola",consolé.log);
```

Si ejecutamos el código, veremos por consola que se escribe la palabra ***Hola.*** El código puede ser muy difícil de entender inicialmente pero es muy interesante. La función ***escribe*** recibe dos parámetros: el primero es el texto a escribir. El segundo es el nombre de la función que se encargará de realizar la escritura. Hemos pasado como segundo parámetro la expresión **consolé. log** por lo que la expresión función (dato, consolé .log) es totalmente equivalente (en este caso) **a consolé.log(dato).**

No parece muy útil este código, pero si es fácil entender su versatilidad, ya que también podemos lanzar esta invocación:

```
escribe("Hola",consolé.error);
```

consolé.error es un método que permite escribir un error por consola. Normalmente la diferencia es que el texto sale coloreado en rojo. Pero lo interesante es que la función cambia su forma de escribir debido a la función que usamos.

Un último ejemplo sería:

```
escribe("Hola",alert);
```

Evidentemente, esta función no es muy útil, pero veremos más adelante las enormes posibilidades que dan este tipo de funciones.

Veamos este otro ejemplo:

```
function escribir(x,acción){
        consolé.log(accion(x));
}
function doble(y){
        return 2*y;
}
escribir(12,doble);
```

El resultado que se escribe es 24. Vayamos por partes:

- Al definir la función ***doble*** la damos la capacidad de devolver el parámetro que la enviemos multiplicado por dos.
- La función ***escribir*** recibe dos parámetros: ***x*** (un número) y una función. Con esos parámetros invoca a **console.log** haciendo que muestre el resultado de la función que indiquemos a la que pasaremos el parámetro ***x.***
- La invocación de **escribir(12,doble)** acabará produciendo en la función ***escribe*** el código **consolé.log(doble(12))**

Es muy habitual usar funciones callback usando funciones anónimas. Si observamos el siguiente código:

```
escribir(12,function(y){
        return 2*y;
});
```

Si suponemos que la función ***escribir*** es la misma que en el código anterior, esta llamada a la función escribir provoca el mismo resultado: 24. El segundo parámetro no es el nombre de una función, es una función anónima cuya definición es devolver el parámetro que reciba multiplicado por dos. Ese código se asociará al parámetro ***acción.***

Es más, incluso podríamos usar funciones flecha:

```
escribir(12,y=>2*y);
```

Inicialmente cuesta mucho crear funciones propias que usen funciones callback como parámetros. Pero lo útil es que hay muchos métodos de objetos básicos de JavaScript que requieren utilizar funciones callback. El uso de estos métodos facilita su aprendizaje. Por ello, en el apartado siguiente veremos algunas utilidades ya creadas que requieren usar funciones callback, y

que nos van a dar funcionalidades muy avanzadas sobre las estructuras de datos explicadas en la unidad anterior.

5.4.4 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS

5.4.4.1 ORDENACIÓN AVANZADA DE ARRAYS

Cuando explicamos el método ***sort*** para ordenar arrays (véase 4.4.8 ***"Modificar el orden de los elementos de un array",*** en la página 139) explicamos el problema de que, por defecto, esta función ordena el texto aplicando estrictamente el orden de la tabla Unicode. Y así, este código:

```
const palabras=["Ñu","Águila","boa","oso","marsopa","Nutria"];
palabras.sort();
consolé.log(palabras);
```

Produce este resultado, que en absoluto es el deseable:

```
[ 'Nutria', 'boa', 'marsopa', 'oso', 'Águila', 'Ñu' ]
```

Pero la función ***sort*** tiene la posibilidad de indicar un parámetro que es una función callback. Esa función debe recibir dos parámetros que sirven para explicar el criterio de ordenación. Por lo que debemos programar el código de esa función de modo que comparando, en la forma deseada, los parámetros:

- La función devuelva un número negativo si el primer parámetro es menor que el segundo
- Devuelva cero si son iguales
- Devuelva un número positivo si su segundo parámetro es mayor que el primero.

Un ejemplo de función personal para ordenar de modo que aparezcan primero los textos más cortos, sería esta:

```
function ordenPersonal(a,b){
    return a.length-b.length;
}
```

La función devuelve, dando por hecho que ha recibido dos parámetros de tipo string, un número negativo si el primer parámetro es más corto que el segundo, cero si los tamaños son iguales y un número positivo si el primer parámetro es más largo que el segundo. Si usamos esa función como función anónima (y en forma de flecha) que enviamos a sort el código sería:

```
const palabras^["Ñu","Águila","boa","oso","marsopa","Nutria"];
palabras.sort((a,b)=>a.length-b.length);
consolé.log(palabras);
```

Consigue el resultado:

```
[ 'Ñu', 'boa', 'oso', 'Águila', 'Nutria', 'marsopa' ]
```

Pero volvamos al problema de ordenar textos en la forma deseada respetando la ordenaciópn en idioma castellano. Es decir, dejando la ***eñe*** entre la ***ene*** y la ***o,*** olvidar la diferencia entre mayúsculas y minúsculas y el resto de problemas que aporta el orden estricto de la tabla Unicode. Para ello, afortunadamente, disponemos del método **localeCompare** de los strings (véase (4.1.2.1 ***"Método localeCompare",*** en la página 121). Por lo cual el problema se soluciona de esta forma:

```
const pa1abras=["Ñu","Águi 1a","boa","oso",”marsopa"/'Nutria"];
palabras.sort((a,b)=>a.localeCompare(b));
consolé.log(palabras);
```

Ahora la ordenación es perfecta:

```
[ 'Águila', 'boa', 'marsopa','Nutria', 'Ñu', 'oso' ]
```

Un detalle importante es que localeCompare sin indicar un segundo parámetro que indica el código del país, podría ordenar mal (podría aparecer el ***Ñu*** antes de la ***Nutria)*** porque usa la configuración nacional local del usuario. Por eso, es más acertado incluir el código del idioma sobre el que deseamos ordenar:

```
const palabras=["Ñu","Águila","boa","oso","marsopa","Nutria"];
palabras.sort((a,b)=>a.localeCompare(b ,"es"));
consolé.log(palabras);
```

La capacidad de enviar una función callback para ordenar permite realizar ordenaciones absolutamente personales en un array.

5.4.42 MÉTODO **FOREACH**

JavaScript nos ofrece una forma muy sofisticada de recorrer arrays, mapas y conjuntos. Se realiza mediante un método de los arrays que se llama **forEach** y que se incorporó al estándar **ES2015.** La sintaxis es la siguiente:

```
nombreArray.forEach(function(elemento,indice){
    instrucciones que se repiten por cada elemento del array
});
```

forEach requiere indicar una función que necesita dos parámetros: uno irá almacenando los valores de cada elemento del array y el segundo irá almacenando los índices. No es imprescindible usar ambos, el parámetro que almacena el índice es opcional.

Esa función permite establecer la acción que se realizará con cada elemento del array. Al igual que ocurría con el bucle **for...in** el método **forEach** no tiene en cuenta los elementos indefinidos. Así, el código que permite mostrar un array de notas es:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
notas.forEach(function(nota,i){
        consolé.log('La nota ${i} es ${nota}x);
});
```

Es una forma de recorrer arrays muy elegante, no necesitamos obtener los valores del array mediante el índice **(notas [i]),** el parámetro ***nota*** se encarga de ir recogiendo directamente cada valor del array.

En el caso de los conjuntos, el funcionamiento es semejante, pero a la función callback que recibe como parámetro ***forEach*** no se le indica más parámetro que la variable que recogerá cada elemento del conjunto:

```
let conjunto=new Set();
conjunto.add("Paúl").add("Ringo").add("George").add("John");
conjunto.forEach(function(valor){
        consolé.log(valor);
});
```

Escribirá los elementos del conjunto:

```
Paúl
Ringo
George
John
```

Finalmente decir que, en el caso de los mapas, el método ***forEach*** acepta una función donde se acepta un parámetro para almacenar cada elemento del mapa y otro para almacenar las claves.

```
const provineias=new Map();
provincias.set(l,"Álava").set(28,"Madrid").set(34,"Palencia")
        .set(41,"Sevilla");
provincias.forEach(function(valor,clave){
        consolé.log("Clave: ${clave}, Valor: ${valor}")
});
```

Se escribirá por pantalla lo siguiente:

```
Clave: 1, Valor: Álava
Clave: 28, Valor: Madrid
Clave: 34, Valor: Palencia
Clave: 41, Valor: Sevilla
```

5.4.43 MÉTODO **MAP**

Es otro método de recorrido de arrays que permite recorrer cada elemento y, a través de una función callback que se pasa como único parámetro, establecer el cálculo que se realiza con cada elemento.

El método map no modifica el array, sino que devuelve otro con los mismos elementos y al que se le habrá aplicado la acción que se pasa como parámetro

Si, por ejemplo, deseamos doblar el valor de cada elemento de un array, el código sería:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
```

```
const doble=notas.map(x=>2*x);
consolé.log(doble);
```

El nuevo array doble contiene los mismos valores que el array de notas, pero con los valores doblados. El código anterior escribe:

```
[ 10,
  12,
  <3 empty items>,
  18,
  <2 empty items>,
  16,
  <1 empty item>,
  18,
  <1 empty item>,
  14,
  16 ]
```

Solo los arrays disponen de método ***map.***

SAAA MÉTODO **REDUCE**

Se trata de un método que requiere de una función callback que está pensada para recorrer cada elemento del array. Sin embargo, a diferencia de los métodos ***map yforEach,*** la idea es devolver un valor, resultado de hacer un cálculo con cada elemento del array.

El método en sí tiene un segundo parámetro (el primero es la función callback) que sirve para indicar el valor inicial que tendrá la variable que sirve para acumular el resultado final. La función callback recibe dos parámetros: el primero es el acumulador en el que se va colocando el resultado deseado y el segundo sirve para recoger el valor del elemento del array que se va recorriendo en cada momento.

Así, podemos sumar todos los elementos de un array y devolver el resultado de esta forma:

```
const array=[1,2,3,4,5];
let suma=array.reduce((acu,valor)=>acu+valor,0);
consolé.log(suma);
```

Hemos usado una función flecha como función callback para el primer parámetro del método ***reduce.*** Esta función usa los parámetros ***acu*** para ir almacenando el total de las sumas y ***valor*** que es el que va recogiendo cada valor del array. En el segundo parámetro indicamos un cero para que el parámetro acu empiece valiendo cero (si no usamos ese parámetro, el parámetro ***acu*** empieza valiendo uno).

Vamos a ver, paso a paso, como se interpreta este código

[1] const array= [1,2,3,4,5]

Se crea el array con los valores 1, 2, 3, 4 y 5

[2] **let suma=array.reduce((acu,valor)=>acu+valor,0)**
Invocamos al método reduce, pasamos como primer argumento la función:
(acu,valor)=>acu+valor
el segundo parámetro es un cero

[3] El mecanismo de trabajo de la función callback es este:

[3.1] En la primera llamada el parámetro ***valor*** coge el valor uno (primer elemento del array). El parámetro ***acu*** valdrá cero (que es el valor inicial que hemos indicado). La función devuelve 0+1: es decir, uno.

[3.2] Se avanza al siguiente elemento, el parámetro ***acu*** vale uno (resultado de la llamada anterior), ***valor*** vale dos (valor del segundo elemento del array). La función devuelve 1+2, es decir: 3

[3.3] Se avanza al tercer elemento. El parámetro ***valor*** vale tres (valor del tercer elemento del array), el parámetro ***acu*** también vale tres (resultado de la llamada anterior). Se retorna 3+3, es decir: 6

[3.4] Avanzamos al cuarto elemento con ***acu*** valiendo 6 y ***valor*** valiendo ***4.*** El resultado de esta llamada es 6+4, por lo tanto: 10.

[3.5] El quinto elemento vale 5, el acumulador vale 10. El resultado de esta llamada, que es la última, es 10+5.

[3.6] El resultado de la función reduce será 15 (1+2+3+4+5), valor de la suma de todos los elementos del array.

5.4.4.5 MÉTODO **FILTER**

El método **reduce** es muy potente, pero a nivel práctico no se usa demasiado. El método ***filter,*** sin embargo, es muy utilizado. Este método utiliza una función callback que recibe un único parámetro. Gracias a ese parámetro se recoge cada valor del array. La fúnción retorna una condición que debe cumplir cada elemento. Este método obtiene un nuevo array que tendrá como elementos, aquellos que cumplan la condición de la función callback:

```
const array=[4,9,2,6,5,7,8,1,10,3];
const arrayF iltrado=array.filter(x=>x>5)
consolé.log(arrayFilirado);
```

El array llamado ***arrayFiltrado*** quedará de esta forma (el array original no se modifica):

```
[ 9, 6, 7, 8, 10 ]
```

Se quedan en este array los elementos que tengan un valor mayor que cinco.

5.5 PRACTICAS RESUELTAS

Práctica 5.1: Función de pares e impares

- En esta práctica crearemos una función muy sencilla que devuelve la palabra "par" si la enviamos un número par. En caso contraría retorna la palabra "impar".
- Conseguir que se escriban en una página web 500 números aleatorios del 1 al 10.000 y que a su lado se diga si es par o impar gracias a la función anterior.

SOLUCIÓN: PRÁCTICA 5.1

El código de la función se debería escribir en un archivo JavaScript separado, eso permitirá utilizar la función en las aplicaciones que deseemos. El archivo podría tener este código (archivo **par.js)**

```
function par(x){
      return (x%2==0) ? "par" : "impar";
}
```

La página web principal debe de cargar el archivo anterior en el apartado de la cabecera **(head),** es lo habitual cuando se usan archivos con funciones. Eso permitiría usar el código JavaScript en cualquier parte del documento.

Como siempre el código JavaScript que realmente modifica la página web se coloca justo antes de cerrar la etiqueta **body.** En ese código es donde usaremos la función creada en el archivo anterior para dibujar tablas.

Práctica 5.2: Función que dibuja una tabla

- Crea una función en JavaScript que nos permita dibujar una tabla en una página web
- Como parámetros indicaremos el número de filas y de columnas con dos números. Por defecto la función tomará 10 filas y 4 columnas
- La tabla se crea con un borde negro de 1 píxel entre cada celda, pero un tercer parámetro permite indicar el color (por defecto será negro). El borde exterior medirá 3 píxeles y siempre será del mismo color que el borde de las celdas.
- La tabla ocupará toda la anchura de la página
- Usa la función para crear una tabla con borde negro de 10 filas y 4 columnas
- Úsala de nuevo para generar una tabla de 20 filas y 10 columnas, con borde negro
- Finalmente, consigue dibujar 10 tablas de 5 filas y 4 columnas que tengan borde verde.

SOLUCIÓN: PRÁCTICA 5.2

Nuevamente conviene tener el código de la función en un arhivo separado (se llamará **tabla.js)**

```
funetion dibujaTabla(nFilas=10znCols=4,color="black"){
        document.write(
                '<table style=’border-collapse:collapse;' +
                        xborder:3px solid ${color};width:100%;'>'
        );
        for(let i=l;i<=nFi las;i++){
                document.write("<tr>");
                for(let j=l;j<=nCols;j++){
                        document.write(
                                '<td style=’border:lpx solid                ' +
                                        x${color}'> </td>'
                        );
                }
                document.write("</td>");
        }
        document.write("</table>");
}
```

Este sería el código de la página principal **(index.html):**

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta http-equiv="X-UA-Compatible" contenta"ie=edge">
        <title>Función de dibujo de tablas</title>
        <script src="tabla.js"></script>
</head>
<body>
<script>
        dibujaTabla();
        dibujaTabla(20,10);
        for(let i=l;i <=10;i++){
                dibujaTabla(5,4,"green");
        }
</script>
</body>
</html>
```

Práctica 5.3: Función que permite saber si un número es primo

- Crea una función que permita saber si un número es primo o no.
- Un número es primo si no se puede dividir de forma entera por otro número, sin contar el uno ni el propio número, dando un resto de cero.
- Aprovecha la función creada para crear una página web que escriba los números primos del lal 1000.

SOLUCIÓN: PRÁCTICA 5.3

Hay muchas posibles soluciones y algunas mucho más eficientes que la que se presenta aquí. Pero es interesante este ejercicio para plantear cuestiones algorítmicas sobre resolución de problemas complejos.

Sin conocer algoritmos matemáticos ya resueltos sobre esta cuestión, un primer planteamiento sería ir dividiendo el número sobre el que deseamos saber si es primo desde el número dos hasta llegar al número anterior. Pero eso significa que si el número fuera, por ejemplo, el 1000 deberíamos comprobar 998 divisiones.

La verdad es fácil entender que si a la mitad de divisiones ninguna da un resto cero, no hace falta seguir dividiendo. Es más, llegando a la raíz cuadrada del número sin que ninguna división sea exacta, ya sabemos que el número es primo. Por ejemplo, para el 11, basta dividir entre 2 y 3. No tiene sentido ir más allá, si fuera divisible por el cuatro, lo sería también por un número menor que el cuatro, ya que el cuadro supera la raíz cuadrada y así con todos los demás números.

Independientemente de estos planteamientos tan matemáticos, esta solución vuelve a probar lo interesante de programar de forma modular, porque el colofón es sacar los números primos del 1 al 1000 y aunque aquí hay otros algoritmos (como la criba de Eratóstenes), lo cierto es que podemos aprovechar nuestra función para obtener los primos. No es la solución más eficiente, pero, sí es muy interesante como aplicación de la utilidad de las funciones.

Este sería el código de la función que devuelve **true** si su parámetro es primo y **false** si no lo es. Es muy interesante esta función por el uso continuo de **return.**

```
function esPrimo(n){
        //No tiene sentido enviar como parámetro
        //números negativos o cero, por si acaso devolvemos
        //false en previsión de un bucle infinito
        if(n<l) return false;
        //El 1 le separamos, no hace falta dividir
        if(n==l) return true;
        //Para el resto de números empezamos a dividir entre 2
        //y terminamos cuando la raíz cuadrada del contador supere al número
        for(let i=2;i*i<=n;i++){
                //si el número se puede dividir por el contador
```

```
        //no tenemos un primo
        if(n%i==0) return false;
    }
    //si hemos salido del bucle sin ejecutar el return
    //tenemos un número primo
    return true;
}
```

La finalización de ejercicio es el código de la página web que muestra los primos del 1 al 1000. En ella cargamos la función anterior (el archivo será **primo.js)** y después ejecutamos un bucle en el que vamos comprobando cada número. Para que sea más rápido el bucle, hemos sacado primero los números 1 y 2 (no los comprobamos) y el bucle (sabiendo que el resto de pares no son primos) empieza por el 3 y va contando de 2 en 2 (3, 5, 7, 9,11, etc.). Así el bucle es el doble de rápido.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta          http-equiv="X-UA-Compatible"          content="ie=edge" >
    <title>Números primos</title>
    <script src="primo.js"></script>
</head>
<body>
<script>
    //escribimos el 1 y 2,sabemos que son primos
    document.write("<p>l</p> <p>2</p>");
    for(let i=3;i<l0000;i +=2){
        i f(esPrimo(i))
            document.write('<p>${i}</p>');
    }
</script>
</body>
</html>
```

Práctica 5.4: Ordenar palabras en orden inverso

- Crea una aplicación que pida al usuario palabras continuamente hasta que se acepte el cuadro sin texto o se cancele.
- Se eliminarán las palabras repetidas y además se ordenarán en español, pero en orden inverso (de la Z a la A).

SOLUCIÓN: PRÁCTICA 5.4

Este sería el código JavaScript a colocar en la aplicación web:

```
const setPalabras=new Set();
var arrayPalabras=[];
//hacemos una primera lectura adelantada
var palabra=prompt("Escriba una palabra (o nada si                    desea acabar");
while(palabra !=null && palabra !=""){
        setPalabras.add(palabra);
        palabra=prompt("Escriba una palabra (o nada si                    desea acabar");
}
//convertimos en array
arrayPalabras=[...setPalabras];
//ordenamos por lo contrario a localCompare
//por eso ponemos un signo "menos" por delante
arrayPalabras.sort((a,b)=>(-a.localeCompare(b,"es")));
//recorremos el array usando forEch
//(podríamos) usar cualquier otro bucle
arrayPalabras.forEach(function(pal){
        document.write('<p>${pal}</p>');
});
```

Práctica 5.5: Crear mapa con repeticiones de arrays

- Crea una función que reciba un array de palabras.
- La función devolverá un mapa que contenga como clave cada palabra y el valor es el número de veces que esa palabra aparece en el array.
- Haremos una página web que lea palabras hasta que el usuario cancele o deje el cuadro vacío y mostraremos las repeticiones de las palabras.

SOLUCIÓN: PRÁCTICA 5.5

El código de la función lo grabaremos en el archivo repeticiones.js

```
function mapaRepeticiones(array){
        if(array instanceof Array -- false){--
                //no es un array
                return nuil;
        }
        else{
                let mapa=new Map();
                //en este caso llamamos clave a cada valor del array
                //porque serán las claves del mapa
                for(let clave of array){
                        //comprobamos si el valor está en el mapa
```

```
                if(mapa.get(clave)Hunde fined){
                        //si lo está incrementamos el contador
                        mapa.set(clave,mapa.get(clave)+l);
                }
                else{
                        //si no, la añadimos con contador a 1
                        mapa.set(clave,1);
                }
        }
        return mapa;
    }
}
```

El código javaScript que se encarga de pedir al usuario palabras e indicar las veces que se repite cada una, es el archivo **accion.js.** Usa la función definida en el archivo anterior.

```
const arrayPalabras= [];
let mapa;
//hacemos una primera lectura adelantada
var palabra=prompt("Escriba una palabra (o nada si desea acabar");
while(palabra Hnull && palabra!=""){
        arrayPalabras.push(palabra);
        palabra=prompt("Escriba una palabra (o nada si desea acabar");
}
//ordenamos el array
arrayPalabras.sort((a,b)=>(a.localeCompare(b,"es")));
//convertimos en mapa
mapa=mapaRepeticiones(arrayPalabras);
//recorremos el mapa y mostramos las repeticiones
for([palabra,cont] of mapa){
        document.write( <p>$!palabra¡, ${cont repeticiones ):
}
```

Finalmente la página carga en la cabecera el código de la función para asegurar su disponibilidad en el resto de archivos. En el cuerpo carga la acción en sí de la página.

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Repeticiones de palabras</title>
        <script src="repeticiones.Js"> </script>
</head>
<body>
    ‹script src=accion.js› ‹/script›
</body>
</html>
```

Práctica 5.6: Función recursiva Fibonacci

- La función de Fibonacci es un clásico de la programación recursiva. Se trata de una función relacionada con una sucesión de elementos donde los dos primeros son el cero y el uno, y el resto son la suma de sus dos anteriores.
- Es decir, la sucesión es 0,1,1, 2, 3, 5, 8,13, 21, 34, 55, 89, etc.
- A la función le mandaríamos el número del que queremos saber el valor Fibonacci y nos devolvería dicho valor. Así si pasamos el número 10, devolvería 55.
- Es mucho más fácil la solución recursiva, pero sería muy interesante hacer también la solución no recursiva.

SOLUCIÓN: PRÁCTICA 5.6

La idea recursiva para del hecho de que la función de Fibonacci de un determinado número es la suma de los dos anteriores. Si no tenemos cuidado este genera una pila de llamadas infinita, pero las llamadas finalizan en el número uno y en el cero, ya que de estos enviamos los valores uno y cero respectivamente. Este sería el código de la función:

```
function fibonacci(x){
        if(x==0) return 0;
        else if(x==l) return 1;
        else return fibonacci(x-1)+fibonacci(x-2);
}
```

Como se puede apreciar el código es muy sencillo. Para la solución sin recursividad debemos ir recorriendo los elementos y acumulando las sumas, es algo más difícil de programar, pero es más eficiente:

```
function fibonacci 2(x){
        //acumuladores de los dos elementos anteriores
        let acul=l;
        let acu2=0;
        //acumula el resultado final
        let fibo=l;

        if (x==0) return 0;
        else if (x==l) return 1;
        else{
                for(let i=2;i <=x;i++){
                        //se actualizan los valores para la siguiente suma
                        [acul,acu2]=[acu2,fibo];
                        //resultado
                        fibo=acul+acu2;
                }
                return fibo;
        }
}
```

5.6 PRÁCTICAS RECOMENDADAS

Práctica 5.7: Función para detectar palíndromos

- Crea una función para resolver lo que se pedía en la Práctica 4.7 ***"Palíndromos",*** en la página 157.
- La función recibe un texto y devolverá verdadero si es un palíndromo y falso si no lo es.
- Hay que tener en cuenta que para que se consideren bien los palíndromos, se ignoran los signos de puntuación (espacios, interrogaciones, comas, puntos, etc.) también se ignoran tildes y diéresis (se considera igual el carácter ***á*** que el carácter ***a)*** y no se distingue entre mayúsculas y minúsculas.

Práctica 5.8: Función para detectar anagramas

- Crea una función a la que se le pasen una serie de textos (mínimo dos) y detecte si los mismos son anagramas o no.
- Un anagrama es una palabra que resulta de trasponer las letras de otra: por ejemplo ***ESTANCO y ACENTOS.***
- La función devuelve verdadero si todas las palabras pasadas son anagramas de las mismas letras.

Práctica 5.9: Función Tribonacci

- Basada a la sucesión de Fibonacci, tenemos una sucesión conocida como **Tribonacci.** En ella cada elemento es la suma de los tres anteriores.
- La sucesión es 1,1, 2,4, 7,13,24,44, 81,149,274, etc.
- Resuelve la función de forma recursiva y de forma iterativa.

Práctica 5.10: Crear función "Filtro"

- El método ***filter*** de los arrays visto en este tema, permite indicar una función **callback,** para aplicar un filtro a los elementos de un array.
- Es buena práctica para aprender a implementar funciones **callback** tratar de crear nuestras propias funciones

- Crea una función llamada **filtro** que reciba un array y una función callback. La función callback se entenderá que tendrá un solo parámetro y que devuelve verdadero o falso dependiendo del criterio que establezcamos.
- Nuestra función ***filtro*** retornará un nuevo array con los elementos que cumplan los criterios establecidos en la función callback.
- Pruébala con varios arrays y funciones de filtro que establezcas a medida.

Práctica 5.11: Mapa buscaminas

- Crear una aplicación web que muestre un mapa del popular juego buscaminas en el que aparezcan las minas dibujadas y también que se indique en las casillas sin minas, las minas que hay alrededor.
- Hacerlo de forma modular de manera que dividamos la aplicación en una serie de funciones. Concretamente recomendamos:
  - Una función a la que le pasemos el tablero buscaminas (sería un array de dos dimensiones) y coloque en él de manera aleatoria las minas. Esta podría dividirse en dos, siendo una más sencilla la que recibe el array del tablero, una posición en él y devuelve las minas alrededor de esa posición. La función principal simplemente invoca a esa segunda recorriendo cada casilla
  - Otra función que recorra el tablero marcando en cada casilla las minas que hay alrededor. Finalmente una tercera función que dibuje el tablero en una página web
- Se pedirá al usuario el tamaño del tablero y las minas a colocar.
- Ejemplo de resultado final (tablero de 9x9 y 16 minas):

1	MINA	1		1	MINA	2	1	MINA
	1	2	1	2	2	MINA	3	2
		1	MINA	2	2	2	MINA	1
1	1	2	2	MINA	2	2	1	1
1	MINA	2	3	4	MINA	1	1	1
1	2	3	MINA	MINA	3	3	MINA	1
	1	MINA	3	4	MINA	4	2	2
1	2	1	1	2	MINA	3	MINA	1
1	MINA	1		1	1	2	1	1

5.7 RESUMEN DE LA UNIDAD

Las funciones facilitan la modularidad a la hora de programar aplicaciones. Cada función será capaz de resolver una tarea sencilla a partir de una serie de datos que la función requiere para poder ejecutar su tarea.

Las funciones JavaScript se declaran con la palabra clave **function** a la que sigue el nombre de la función, los parámetros de la misma (que son los datos que requiere para poder resolver la tarea) y el cuerpo de la función (que es el código que la función ejecute cuando se la invoque).

En el cuerpo de la función puede aparecer la palabra **return** que es la que permite que la función devuelva un valor concreto. No es obligatorio que aparezca, hay funciones que no retornan un valor concreto.

Hay funciones anónimas cuyo cuerpo se puede asignar a variables o parámetros de funciones, que pasan a ser referencias a la propia función.

Las funciones flecha permiten definir funciones de forma más cómoda sin tener que utilizar la palabra **function** ni, en muchas ocasiones, tener que indicar explícitamente la palabra **return.** Son cómodas para funciones simples, pero no tanto para funciones complejas.

Cuando una función se invoca indicando tipos simples de datos (números, booleanos, strings, etc.) en los parámetros, estos reciben una copia de esos valores. Pero si enviamos arrays u otros objetos, entonces los parámetros obtendrán una referencia al objeto original. En este último caso, si se modifica el objeto a través del parámetro, se estará modificando el objeto original y no una copia.

JavaScript admite parámetros de funciones con valores predeterminados y, también definir parámetros con el operador de propagación para un número indeterminado de parámetros.

Cada vez que invocamos a funciones, estas se colocan en una pila que tiene un tamaño finito y que podríamos colapsar en caso de excesivas llamadas a funciones sin resolver.

La recursividad es una técnica que permite que una función se invoque a sí misma. Con habilidad permite resolver de forma sencilla, problemas complejos.

Las funciones **callback,** son funciones que recogen los parámetros de otras funciones. Así, las propias funciones reciben datos y, además, acciones que pueden realizar.

El uso práctico de funciones callback queda de manifiesto en tareas como ordenar arrays **(sort),** recorrer elementos de objetos iterables **(forEach),** hacer cálculos con todos los elementos de un array **(map)** o filtrar los elementos de un array en base a una condición **(filter).**

5.8 TEST DE REPASO

1.- **¿Qué modelo de programación de aplicaciones está relacionado con el uso de funciones?**

a) Programación estructurada
b) Programación modular
c) Programación orientada a objetos
d) Programación orientada a eventos

2.- **¿Qué muestra este código por pantalla?**

```
function fl(...a){
    consolé.log(a)
}
fl(1,2,3,4);
```

a) Set { 1, 2, 3, 4 }
b) Map { 1=>1,2=>2,3=>3,4=>4 }
c) 1, 2, 3, 4
d) [1, 2, 3, 4]

3.- **¿Qué muestra este código?**

```
function f2(a,b,c=10){
  consolé.log('a=${a}, b=${b},' +
                    'c=${c}');
}
 £2(1)
```

a) a=l
b) a=l, b=, c=10
c) a=l, b=null, c=10
d) a=l, b=NaN, c=10
e) a=l, b=undeñned, c=10
f) Ocurre un error, no se muestra nada

4.- **¿Qué es una función flecha?**

a) Una función anónima declarada con una sintaxis más sencilla
b) Una función que aplica técnicas de recursividad
c) Una función donde uno de sus parámetros es una referencia a otra función
d) Una función que no utiliza la palabra clave **return.**

5.- **¿Qué escribe este código?**

```
function f3(n){
  if(n<=l) consolé.log(n);
  else {
    consolé.log(n);
    consolé.log(f3(parseInt(n/2)));
  }
}
f3(19);
```

a) 19 19 19 19 19 1
b) 19 19 19 19 1
c) 19 9 4 2 1
d) 19 10 5 3 1

6.- **¿Qué muestra por pantalla este código?**

```
var x=19;
function f4(){
   consolé.log(x);
}
f4();
consolé.log(x);
```

a) undeñned 19
b) undeñned undeñned
c) 19 19
d) Ocurre un error

7.- **¿Qué muestra por pantalla este código?**

```
let x=19;
function f5(){
   consolé.log(x);
}
f5();
consolé.log(x);
```

a) undeñned
19
b) undeñned
undeñned
c) 19
19
d) Ocurre un error

8.-

9.- **¿Qué muestra por pantalla este código?**

```
function f6(){
    let x=19;
    consolé.log(x);
}
f6();
consolé.log(x);
```

a) undeñned
   19
b) undeñned
   undeñned
c) 19
   19
d) Ocurre un error

10.- **¿Qué muestra por pantalla este código?**

```
function f6(){
    var x=19;
    consolé.log(x);
}
f6();
consolé.log(x);
```

a) undeñned
   19
b) undeñned
   undeñned
c) 19
   19
d) Ocurre un error

11.- **¿Qué muestra por pantalla este código?**

```
function f7(a,b){
      return b(a);
}
consolé.log(f7(9,x=>x+2));
```

a) Ocurre un error
b) function
c) object
d) 11

12.- **¿Qué muestra por pantalla este código?**

```
function f8(s){
      s="no";
      return s;
}
let t="si";
f8(t)
consolé.log(t);
```

a) si
b) no
c) undeñned
d) Ocurre un error

13.- **¿Qué muestra por pantalla este código?**

```
function f9(a){
      a=["no"];
      return a;
}
let b=["si"];
f9(b);
consolé.log(b);
```

a) ['si']
b) ['no']
c) undeñned
d) Ocurre un error

14.- **¿Qué muestra por pantalla este código?**

```
function fl0(a){
      a[0]="no";
      return a;
}
let b=["si"];
fl0(b);
consolé.log(b);
```

a) ['si']
b) ['no']
c) undeñned
d) Ocurre un error

UNIDAD 6

PROGRAMACIÓN DE OBJETOS EN JAVASCRIPT

OBJETIVOS

— Reconocer las ventajas de utilizar objetos en la programación de aplicaciones web

— Identificar las características principales de la programación orientada a objetos en JavaScript

— Utilizar propiedades y métodos de objetos propios o predefinidos

— Crear objetos propios especificando sus propiedades y métodos

— Establecer mecanismos de herencia por prototipado

— Examinar objetos recorriendo sus propiedades y métodos

— Distinguir la utilidad y funcionamiento básico de la notación JSON

— Identificar la utilidad de los objetos predefinidos para manipularlos adecuadamente

— Realizar cálculos avanzados con ayuda del objeto Math

— Utilizar el objeto Date para manipular fechas y horas

— Distinguir el funcionamiento de las expresiones regulares en JavaScript

— Validar expresiones regulares con ayuda de los objetos RegExp

CONTENIDOS

6.1 JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS
- 6.1.1 ¿QUÉ ES LA POO?
- 6.1.2 CARACTERÍSTICAS DE LA POO
- 6.1.3 JAVASCRIPT COMO LENGUAJE ORIENTADO A OBJETOS

6.2 USO DE OBJETOS
- 6.2.1 ACCESO A PROPIEDADES Y MÉTODOS
- 6.2.2 OBJETOS LITERALES
- 6.2.3 OBJETO THIS
- 6.2.4 RECORRER LAS PROPIEDADES DE UN OBJETO
- 6.2.5 BORRAR PROPIEDADES DE OBJETOS

6.3 USO AVANZADO DE OBJETOS
- 6.3.1 CREAROBJETOS A TRAVÉS DE CONSTRUCTORES
- 6.3.2 OPERADOR INSTANCEOF
- 6.3.3 PROTOTIPOS
- 6.3.4 NOTACIÓN JSON

6.4 OBJETOS PREDEFINIDOS
- 6.4.1 OBJETO MATH
- 6.4.2 OBJETO DATE
- 6.4.3 EXPRESIONES REGULARES

6.5 PRÁCTICAS RESUELTAS

6.6 PRÁCTICAS RECOMENDADAS

6.7 RESUMEN DE LA UNIDAD

6.8 TEST DE REPASO

6.1 JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS

6.1.1 ¿QUÉ ES LA POO?

Las siglas **POO** se corresponden a **Programación Orientada a Objetos,** un término que ya tiene un largo recorrido dentro de la ciencia de programación de computadoras y que cobró una enorme dimensión a finales de los 80 hasta su total implantación en la década de los 90 del siglo pasado.

En definitiva, la POO es un modelo de programación. Su éxito se debe a que ha conseguido facilitar enormemente la creación de aplicaciones al conseguir que una aplicación sea un conjunto de objetos que se comunican. Cada objeto se programa de forma independiente, consiguiendo una modularización mayor que la propia programación modular. Los objetos son una estructura que aglutina datos y acciones (funciones).

La idea es que los objetos envían y reciben mensajes entre sí. Un objeto será un elemento de la aplicación capaz de recibir mensajes que provocan un procesamiento de dicho de mensaje hasta producir un resultado que, a su vez, determinará un envío de mensajes a otro objeto.

El mantenimiento de las aplicaciones es más sencillo ya que nos concentramos en hacer que cada objeto funcione debidamente, y, cuando dicho objeto funciona, lo podemos utilizar en todas las aplicaciones que deseemos.

La idea de objeto es la de una estructura que aglutina datos y acciones. Los datos son los atributos o propiedades de los objetos. Si por ejemplo, un objeto representara un ***coche,*** los datos (que se llaman **atributos** o **propiedades)** podrían ser la marca, el color, la potencia, etc. Las acciones (que se llaman **métodos)** podrían ser arrancar, parar, acelerar, repostar,... es decir, lo que un coche es capaz de hacer.

Una vez que un tipo de objeto (en terminología POO un tipo de objeto se conoce como **clase)** se ha definido, se puede utilizar las veces que queramos y ya no nos preocuparemos por él. Programar aplicaciones consistirá en analizar y diseñar los objetos que necesitamos y cómo se han de comportar y comunicar entre sí.

6.1.2 CARACTERÍSTICAS DE LA POO

Hay cierto debate sobre si JavaScript es realmente un lenguaje orientado a objetos. Ese debate se debe a que se considera que otros lenguajes (como C++, java o Python) implementan de forma más completa el paradigma de la POO.

La POO se dice que ha de tener estas características:

- **Abstracción.** Propiedad que permite reutilizar un objeto sin conocer su funcionamiento interno, nos bastará con que se nos proporcione lo que se conoce como su **interfaz externa.** La interfaz son las propiedades y métodos que permiten manipular y utilizar el

objeto. En definitiva, la abstracción en la POO nos permite reutilizar el código de forma óptima, al poder utilizar objetos sin conocer los detalles de su implementación.

- **Encapsulamiento.** Capacidad de que los objetos puedan ocultar algunos métodos y propiedades de modo que, podamos elegir solo los que nos interesan que queden visibles.

- **Polimorfismo.** Esta característica permite conseguir que diferentes tipos de objetos puedan tener métodos con el mismo nombre, pero que actuarán de forma diferente. Por ejemplo, los objetos de clase círculo podrían tener un método llamado suma que permite unir su superficie a la de otro círculo, pero la clase **númeroGigante** podría tener el mismo método, pero para hacer sumas aritméticas.

- **Herencia.** Los diferentes tipos de objetos se suelen relacionar con **clases** de objetos. La mayoría de lenguajes define clases y luego crea objetos a partir de esas clases, es decir: objetos que son de una determinada clase. La herencia permite que haya clases que hereden las características de otras clases. Por ejemplo, la clase ***Automóvil*** puede tener un método que se llame ***acelerar*** y otro que se llame ***frenar.*** Si definimos una clase llamada ***Coche,*** indicando que esa clase deriva de la clase ***Automóvil,*** los coches también podrán frenar y acelerar.

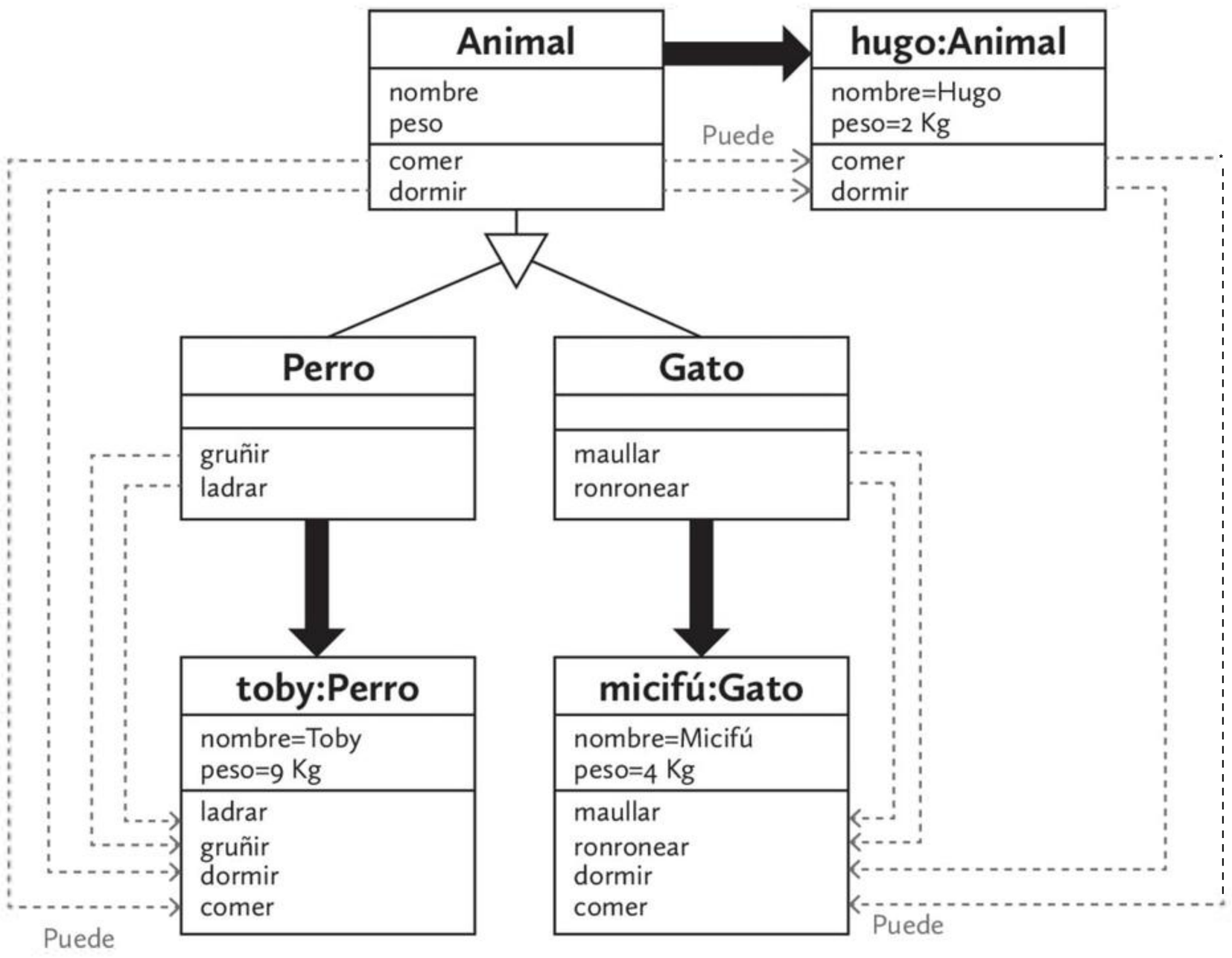


Figura 6.1: Ejemplo de uso de clases y objetos con herencia en la POO

En la Figura 4.1, **Animal, Perro** y **Gato** son clases de objetos. Mientras que **toby, hugo** y **micifú** son objetos ya concretos. Perro y Gato son clases que descienden de la clase **Animal** (son animales). El objeto ***toby,*** que es un perro, hace lo que hacen los perros (ladrar y gruñir), pero también

heredan lo que hace cualquier animal (comer y dormir). Lo mismo ocurre con ***micifú,*** hace cosas de gatos y también las cosas que comparte con cualquier otro animal. De ***hugo*** solo sabemos que es un animal genérico por ello solo puede hacer las cosas que hace cualquier otro animal.

6.1.3 JAVASCRIPT COMO LENGUAJE ORIENTADO A OBJETOS

Hay lenguajes que llevan a rajatabla las características vistas en los apartados anteriores, otros aportan cierta flexibilidad, o bien una reinterpretación de las mismas. Es todo un debate, a veces polémico, si JavaScript es un lenguaje realmente orientado a objetos o no.

Sin entrar a fondo en este debate, cierto es que JavaScript trabaja de forma diferente los objetos con respecto a lenguajes como java o C++, pero indudablemente los objetos son importantísimos en JavaScript e independientemente de cómo implementa el paradigma, toda aplicación JavaScript utiliza objetos y permite modular el código basándonos en interacciones entre objetos.

Sin duda, la gran diferencia de JavaScript está en el hecho de cómo maneja las clases y la herencia. Es posible programar objetos sin utilizar clases en JavaScript. En su lugar utiliza una orientación basada en prototipos. Ese mecanismo permite heredar características entre objetos gracias a considerar que cada objeto tiene un prototipo asociado que podemos modificar en cualquier momento para añadirle nuevas propiedades y métodos.

6.2 USO DE OBJETOS

6.2.1 ACCESO A PROPIEDADES Y MÉTODOS

Si tenemos un determinado objeto, el acceso a sus propiedades y métodos se consigue gracias al operador punto (.) Para acceder a una propiedad de un objeto, la sintaxis es:

```
E>b. jeto^propiedad
```

Cambiar el valor de una propiedad es posible, por ejemplo, así:

```
coche.co1or="Rojo";
```

Se ha modificado la propiedad ***color*** del coche. También podemos acceder usando esta forma:

```
objeto["propiedad"]
```

Por ejemplo:

```
coche[“color”]=”Rojo”;
```

Para utilizar métodos, la sintaxis es:

```
obj eto.métodof[parámetros!)
```

Ejemplo:

```
coche.acelerar(25);
```

También es posible con:

```
coche["acelerar"](25);
```

6.2.2 OBJETOS LITERALES

Los objetos más sencillos que podemos crear en JavaScript son los llamados **literales** o **instancias directas.** Son objetos en los que se pueden definir directamente sus propiedades y métodos sobre la marcha. Por ejemplo:

```
let punto=new 0bject(); //punto es un objeto, por ahora vacío
punto.x=5; //Definimos la propiedad x y le damos valor
punto.y=punto.x*2;
//Definimos la propiedad "y" haciendo que valga el doble de lo que
//vale "x", es decir: 10
punto.mostrarCoordenadas=function(){
   return '(${punto.x},${punto.y})';
}
consolé.log(punto.mostrarCoordenadas( ) ) ; //Escribe (5,10)
```

La primera línea del código anterior:

```
let punto=new 0bject();
```

Índica que la variable llamada ***punto*** es un objeto. Hay una instrucción más sencilla y equivalente a la anterior:

```
let punto={};
```

Las llaves permiten indicar que la variable punto es un objeto vacío. Tras esa instrucción podemos definir propiedades y métodos de la misma forma. Otra posibilidad es crear el objeto y directamente asignarle propiedades y métodos:

```
let punto={
      x: 19,
      y:36,
      mostrarCoordenadas:function(){
            return '(${punto.x},${punto.y})';
      }
};
consolé.log(punto.mostrarCoordenadas());
```

Esta forma de definir objetos se basa en la forma:

```
{
propiedad1:valor1,
propiedad2:valor2,
…
método1:function(…){…},
…
}
```

El nombre de las propiedades y métodos puede ir entrecomillado, por lo que se admiten incluso nombres de propiedades y métodos con guiones y espacios en blanco (aunque no es muy recomendable hacerlo). El orden de las propiedades y métodos no es estricto, podemos empezar por métodos, luego definir propiedades, luego definir más métodos, etc. Es decir, el orden lo decidimos a voluntad. Ahora bien, es recomendable, para una mejor legibilidad del código, definir primero las propiedades y luego los métodos.

```
let libro = {
    título : "Manual de UFOlogía",
    " n-serie"      :   "18702",     //Las propiedades pueden llevar espacios,
                //guiones, etc. por lo que en esos casos se
                //usan comillas
    autores : [ "Pedro Martínez","Ana León"],
                //Esta propiedad es un array
    editorial : { //La editorial es otro objeto
        nombre : "Inexistente S.A.",
        país : "España"
    }
};
consolé.log(1ibro.título); //Escribe: Manual de UFOLogía
consolé.log(libro["n-serie"]); //Escribe: 187C2
console.log (1ibro.editorial.nombre); //Escribe Inexistente S.A.
```

Los objetos pueden ser todo lo complicados que queramos, una propiedad de un objeto puede ser un array, un mapa un conjunto, otro objeto, etc.

6.2.3 OBJETO **THIS**

En alguno de los códigos de ejemplo anteriores ya hemos utilizado la palabra reservada **this.** Para entender mejor su uso, veamos el siguiente ejemplo de código:

```
let punto={
        x: 19,
        y: 36,
        mostrarCoordenadas:function(){
                return '(${x},${y})';
        }
};
consolé.log(punto.mostrarCoordenadas());
```

El resultado de ese código provoca un error de variable indefinida. La razón es que no hay una variable ***x,*** lo que hay es una propiedad ***x*** (lo mismo pasa cony). Por lo tanto, desde la función que se asigna a **mostrarCoordenadas** no se puede acceder a ninguna variable llamada ***x.***

```
let punto={
        x:19,
        y: 36,
        mostrarCoordenadas:function(){
```

```
            return '(${punto.x},${punto.y})';
        }
};
consolé.log(punto.mostrarCoordenadas());
```

Al hacer referencia a ***punto.x y punto.y,*** ya no hay indefinición, está claro que ahora, sí sabemos que nos referimos a las propiedades del punto.

Pero veamos, finalmente este código:

```
let punto={
        x: 19,
        y: 36,
        mostrarCoordenadas:function(){
            return '(${this.x},${this.y})';
        }
};
consolé.log(punto.mostrarCoordenadas());
```

El resultado es el mismo. ¿A quién hace referencia **this?** Al objeto actual, que en este caso es accesible a través de la variable ***punto,*** pero que con código más complejo no sería tan fácil llegar al objeto. **This,** nos permite abstraemos y llegar al objeto propietario de la propiedad o el método que estamos definiendo, sea el que sea.

Para entender mejor las posibilidades de this veamos este código que es un poco más complejo:

```
function doblarX(){
        this.x*=2;
}
let punto={
        x: 15,
        y:7,
        doble:doblarX
};
let incógnita={
        x: 5,
        dbl:doblarX
};
punto.doble();
incógnita.dbl();
consolé.log(punto.x) //Escribe 30, doble de 15
consolé.log(incógnita.x) //Escribe 10, doble 5
```

La función ***doblarX*** sirve para multiplicar por dos la propiedad ***x*** del objeto que sea propietario de esta propiedad. Cuando esta función se asigna al método ***doble*** del objeto ***punto*** lo que ocurrirá, cuando se la invoque, es que se doblará el valor ***x*** de la propiedad del punto. Lo mismo ocurre cuando asignamos la función ***doblarX*** al método ***dbl*** del objeto ***incógnita.*** Lo interesante

es que la palabra ***this*** permite llegar al objeto en cuestión y, por eso, podemos definir métodos, propiedades y funciones de forma más abstracta, y que funcionan incluso en objetos distintos.

6.2.4 RECORRER LAS PROPIEDADES DE UN OBJETO

Para poder recorrer las propiedades de un objeto el bucle idóneo ***esfor...in.*** Ejemplo:

```
for(let prop in punto){
        consolé.log('${prop} tiene el valor ${punto[prop]}');
}
```

Suponiendo que el objeto punto es el mismo que en los objetos anteriores, se obtendría este resultado:

```
x tiene el valor 19
y tiene el valor 36
mostrarCoordenadas tiene el valor function(){
                return ᵛ(${this.x},${this.y})\
        }
```

Si no queremos mostrar las funciones, el código podría ser:

```
for(let prop in punto){
        if(typeof punto[prop]!="function"){
                consolé.log('${prop} tiene el valor ${punto[prop]}');
        }
}
```

6.2.5 BORRAR PROPIEDADES DE OBJETOS

El operador **delete** permite eliminar propiedades de los objetos. Se usa indicando, detrás de la palabra **delete,** la propiedad a borrar. Ejemplo:

```
let objeto={
        x: 18,
        y:-10,
        z: -1
};

delete objeto.z;

consolé.log(objeto.x);          //Escribe 18
consolé.log(objeto.z);          //Escribe undefined
```

En el código anterior, la propiedad ***z*** no existe tras la instrucción **delete.**

6.3 USO AVANZADO DE OBJETOS

6.3.1 CREAR OBJETOS A TRAVÉS DE CONSTRUCTORES

Hay una forma de crear objetos que, sin duda, será más del agrado de los programadores acostumbrados a utilizar lenguajes orientados a clases. Se trata de utilizar una función constructora.

Los constructores son un elemento muy conocido por los programadores de lenguajes orientados a objetos. En estos lenguajes, estos métodos sirven para generar nuevos objetos de un tipo en concreto en base a datos que se envían a esa función constructora.

JavaScript utiliza esta idea, pero de forma un poco diferente. Realmente, lo que ocurre es que todas las funciones tienen capacidad de devolver un objeto y que además disponemos del método **this,** visto anteriormente, para hacer referencia al objeto actual al que pertenece la función. Ambos elementos combinados con un operador llamado **new** permite crear nuevos objetos utilizando funciones constructoras.

El operador **new** genera nuevos objetos a partir de esa función. A este proceso se le conoce como **instanciar** objetos, generar ejemplares concretos (en inglés ***instances,*** que se suele traducir literalmente como instancia, aunque quizá no es la traducción ideal).

```
//Función constructora
function Punto(coordX,coordY){
        this.x=coordX;
        this.y=coordY;
        this.mostrarCoordenadas=()=> '(${this.x},${this.y})\
}

let a=new Punto(10,20);
let b=new Punto(-3,6);

consolé.log(a.mostrarCoordenadas());
consolé.log(b.mostrarCoordenadas());
```

El resultado es:

```
(10,20)
(-3,6)
```

El hecho de usar como nombre de la función la palabra **Punto,** con mayúsculas en la primera letra, es una formalidad opcional que se utiliza por parte de los programadores para hacer notar que la función sirve para construir tipos de objetos. En realidad el nombre de la función podemos escribirlo como queramos, pero siempre es aconsejable seguir estas normas para que el código sea más legible.

La función lo que hace es manipular (y a la vez deñnir) las propiedades del objeto que se devuelve. De eso se encarga la palabra **this,** que es la que hace posible llegar a las propiedades

concretas del objeto que se está creando, sabiendo que el valor de esas propiedades podrán ser distintas en cada objeto. Se puede observar en el código anterior, que el constructor define propiedades (x ey) pero también métodos ***(mostrarCoordenadas)*** que serán propiedades y métodos que poseerán todos los objetos creados con este constructor.

Tras definir la función, la utilizamos para crear dos objetos distintos (a y ***b)*** que usan esa función gracias al operador **new** para crear dos objetos distintos, pero ambos del mismo tipo. Cuando usamos el método **mostrarCoordenadas,** la información que devuelve es distinta porque las coordenadas x e y son diferentes.

Es decir **Punto** es un tipo de objetos (evitamos usar la palabra clase, que en JavaScript es controvertida) mientras que ***a*** y ***b*** son objetos de ese tipo.

Un detalle que hay que saber, aunque es de conocimiento muy profundo del lenguaje JavaScript, es que realmente los constructores son funciones normales. Pasan a ser constructores cuando se usan con el operador new. Para que quede claro que es así, basta con ejecutar este código:

```
consolé.log(typeof Punto);
```

El resultado es la palabra **fiinction.** Es más, es perfectamente válido este código:

```
let r=Punto(10,20);
```

Al no usar la palabra new, no hemos creado un objeto, por lo que la variable r es indefinida porque la función en sí no devuelve valor alguno (no hay **return).** Dicho de otro modo, no tiene sentido usar una función constructora de esa forma.

6.3.2 OPERADOR INSTANCEOF

Existe un operador similar a **typeof,** pero pensado para ser usado con objetos, que se llama **instanceof.** La razón para usar este nuevo operador es que todos los objetos pertenecen al mismo tipo de datos: **Object.** A veces, necesitamos comprobar a qué tipo de objeto pertenece uno dado. Teniendo vigente el código del apartado anterior, si ejecutamos después este código:

```
consolé.log(b instanceof Punto); //Devuelve true, b es un Punto
consolé.log(b instanceof Object); //Devuelve true, b es un Objeto
```

La variable b hace referencia a un objeto creado con la función constructora Punto. Es decir, b es un punto. Pero todos los objetos, a su vez, pertenecen a un tipo básico llamado **Object** (son objetos). Gracias a ello, sabemos que los objetos de tipo ***Punto*** heredan todo lo que posee la clase ***Object.***

Hay otra posibilidad para obtener o comprobar el tipo de un objeto. Es una propiedad que se añadió a todos los objetos en la versión del estándar ES2015, es constructor. ñame. La propiedad constructor la tienen todos los objetos y permite obtener información sobre la construcción del objeto. En realidad, constructor es también un objeto que entre sus propiedades posee la propiedad ñame que nos devolverá el nombre de la función constructora y, por lo tanto, el tipo de objeto. Ejemplo:

```
consolé. log(b. constructor . ríame); //Escribe Punto
```

6.3.3 PROTOTIPOS

6.3.3.1 IDEA DE PROTOTIPO

Desde hace tiempo JavaScript utiliza un concepto muy interesante para conseguir implementar lo que en otros lenguajes se conoce como herencia.

En el lenguaje java (no JavaScript) y en otros lenguajes, todo objeto pertenece a una clase. Para definir un objeto es obligatorio primero crear una clase. Además, se puede indicar que una clase es **heredera** de otra clase, por lo que esa clase dispondrá de las propiedades y métodos definidos en la clase de la que hereda.

JavaScript no nació con esa idea. La idea es que todos los objetos procedentes del mismo tipo de función constructora, tienen un mismo prototipo con el que enlazan. El prototipo de un objeto es una serie de métodos y propiedades comunes.

En los lenguajes que usan clases, el código de los métodos se copia a los objetos de esa clase. Sin embargo en JavaScript lo que se hace es enlazar con su prototipo. El prototipo es la parte común de los objetos del mismo tipo. Lo interesante en JavaScript es que podemos modificar el prototipo sobre la marcha, y los objetos que enlazan con ese prototipo inmediatamente estarán al día porque el enlace con su prototipo es dinámico.

En el código anterior, la variable ***a*** es un objeto de clase ***Punto,*** al igual que la variable ***b.*** El acceso al prototipo de un objeto se puede hacer con la propiedad**proto**(hay dos guiones al principio y dos al final de la palabra ***proto).*** Si mostramos esa propiedad para la variable ***a:***

```
consolé.log(a.proto);
```

Se nos muestra:

```
Punto {}
```

Con ello se nos dice que ***a*** usa el prototipo de clase **Punto,** y que ese prototipo no tiene definido ningún método ni propiedad. Los objetos de tipo punto toman las propiedades y métodos definidos en la función constructora, pero no hay propiedades comunes.

Una forma equivalente de obtener el prototipo es mediante el método **getPrototypeof** que es un método de la clase genérica **Object.** Se usa de esta forma:

```
Object.getPrototypeOf(a)
```

El resultado será el mismo.

6.33.2 MODIFICAR PROTOTIPOS

Para modificar prototipos basta con indicar la propiedad **prototype** y después definir propiedades y métodos a voluntad. Esta propiedad, como es lógico, solo está disponible en las funciones

constructoras (en realidad está disponible sobre cualquier función, pero eso es otra cuestión). Así si escribimos este código:

```
consolé.log(Punto.prototype);
```

Obtendremos el prototipo de la función Punto, que será un objeto vacío porque no hemos definido nada en él.

Para definir, por ejemplo, un nuevo método y una nueva propiedad podemos indicarlos y darles valor. Por ejemplo:

```
Punto.prototype.sumaXY=function(){
      return this.x+this.y;
}
Punto.prototype.z=0;
```

Hemos definido un método llamado ***sumaXY***y lo hemos definido para el prototipo de los objetos basados en **Punto.** También hemos creado en ese mismo prototipo una propiedad llamada ***z*** con valor de cero. Si mostramos ahora el prototipo veremos:

```
Punto { sumaXY: [Function], z: 0 ’                }
```

Dará igual que lo hagamos con la expresión Punto, prototype o con (si ***p*** es un objeto de tipo ***Punto)*** p.proto

Ahora el prototipo de ***Punto*** ya tiene un método y una propiedad. Lo interesante es que todos los objetos basados en ***Punto*** disponen de esa propiedad y método:

```
let a=new Punto(10,20);
let b=new Punto(-3,6);
consolé.log(a.sumaXY()); //Muestra 30, resultado de sumar 10+20
consolé.log(b.sumaXY()); //Muestra 3, resultado de sumar -3+6
consolé.log(a.z);          //Muestra          0
consolé.log(b.z); //Muestra 0
```

Un detalle muy importante es lo ocurre si modificamos en un objeto la propiedad heredada. Por ejemplo:

```
a.z=7;
```

Ahora la variable «, ya no coge la propiedad ***z*** del prototipo, tiene un valor propio de esa propiedad. Aunque modifiquemos la propiedad ***z*** a través del prototipo, la variable ***a*** no lo reflejará porque su propiedad z ya es independiente de su prototipo. Sin embargo, todos los demás objetos usarán la propiedad z del prototipo.

Para aclarar este punto observemos este código:

```
consolé.log(a);
consolé.log(b);
```

El resultado de este código sería:

```
Punto { x: 10, y: 20, mostrarCoordenadas: [Function], z: 7 }
```

```
Punto { x: -3, y: 6, mostrarCoordenadas: [Function] }
```

La primera línea del resultado muestra las propiedades y métodos de ***a,*** la segunda los de ***b.*** El objeto ***a*** ha definido una propiedad que no tiene ***b.*** No obstante, ***b*** la tiene:

```
consolé.log(b.z); //Muestra 0
```

La propiedad ***z*** en el caso de ***b*** la obtiene del prototipo. Lo mismo pasaría con los métodos. Si un objeto redefine un método, entonces, usa su versión de forma prioritaria sobre la del prototipo.

Pero siempre podemos acceder a las propiedades y métodos de los prototipos de un objeto:

```
consolé.log(a.proto.z); //Escribe 0
```

Un detalle muy interesante es que podemos modificar el prototipo incluso de los objetos estándar. Ejemplo:

```
Array.prototype.obtenerPares=function(){
        return this.filter((x) = >(x%2==0));
}

let a= [1,2,3,4,5,6,7,8,9] ;
consolé.log(a.obtenerPares());
```

En el código anterior conseguimos que todos los arrays dispongan de un nuevo método llamado **obtenerPares,** el cual devuelve un nuevo array en el que solo quedan los números pares del array original (para ello usa internamente el método **filter** de los arrays).

6.3.4 NOTACIÓN JSON

JSON es el acrónimo de ***JavaScript Objects Notation*** **(Notación de Objetos de JavaScript) que se creó en 2001 por parte de Douglas Crockford** con la idea de aprovechar la forma que posee JavaScript de crear objetos estáticos para crear un formato documental que sea más cómodo y eficiente para los programadores que los lenguajes de marcado como XML.

Aunque inicialmente se creo para ser usado desde JavaScript, actualmente se considera un formato documental independiente de cualquier lenguaje. La información almacenada en formato JSON se puede manipular desde casi cualquier lenguaje de programación actual.

El formato JSON es muy parecido a la forma de crear objetos de JavaScript, se basa en indicar propiedades y valores separados por dos puntos. Los valores se indican igual que en JavaScript (textos entrecomillados, los números tal cual y usando el punto como decimal, etc.) Se permite el uso de arrays, anidar objetos, etc. Pero hay diferencias que hay que reseñar:

- JSON solo admite definir propiedades, no se pueden indicar métodos en el formato JSON.
- En JSON el nombre de las propiedades tienen que ir entre comillas dobles (no se admiten el resto de comillas).

Ejemplo:

```
{
    "título" : "Manual de UFOlogía",
    "n-serie" : "187C2", //Las propiedades pueden llevar espacios,
                //guiones, etc. por lo que en esos casos se
                   //usan   comí 1 las
    "autores" : [          "Pedro Martínez","Ana León"],
                    //Esta   propiedad es un array
    "editorial"          :   { //La editorial es otro objeto
          "nombre"    : "Inexistente S.A.",
            "país" :    "España"
    },
    "edición":2,
    "ensayo":true
};
```

Manipular datos en JSON es muy importante en la creación actual de aplicaciones web, por ello, JavaScript aporta un objeto global llamado **JSON** que permite manipular los datos en este formato.

Este objeto posee dos métodos:

- **stringify.** Sirve para convertir un objeto JavaScript a un string que contiene el formato JSON equivalente.
- **parse.** Recibe un texto en formato JSON y evalúa su corrección. Si es correcto, retorna el objeto equivalente y si no, devuelve una excepción de tipo **SyntaxError.**

Ejemplo de stringify:

```
const musicol={
        nombre:" Bob",
        apell ido:"Dylan",
        fecha_nacimiento: {
                dia:24,
                mes:5,
                año:1941
        },
        discos:['Highway 61 RevisitedBlonde on Blonde','Self Portrait']
}
consolé.log(JSON.stringi fy(musicol));
```

El resultado:

```
{"nombre":"Bob"     , "apellido":"Dylan"          , " fecha_nacimiento":{"día":24,"-
mes":5,"año":1941},"discos":["Highway 61 Revisited","Blonde on Blonde"                    ,"–
Self Portrait"]}
```

El método **parse** haría lo contrario, a partir del texto crea el objeto. Más adelante veremos las enormes funcionalidades que otorga JSON.

6.4 OBJETOS PREDEFINIDOS

El lenguaje javaScript proporciona objetos ya preparados para ser utilizados en nuestro código. En realidad, ya conocemos bastantes: **Array, Map** o **Set** son algunos de ellos. Veremos algunos otros que también nos serán útiles.

6.4.1 OBJETO MATH

Se trata de un objeto global que facilita la ejecución de algunas operaciones matemáticas de alto nivel.

6.4.1.1 CONSTANTES DE MATH

Permiten usar en el código valores de constantes matemáticas, por ejemplo, tenemos a **Math. Pl,** que representa el valor matemático de Pi. Podemos usarla en nuestro código de esta forma por ejemplo:

```
function Circulo(r){
    this.radio=r;
    this.calcularCircunferencia=function()=>(2*Math.PI*this.radio);
}
let c=new Circulo(10);
consolé.log(c.calcularCircunferencia()); //Escribe: 62.83185307179586
```

En la siguiente tabla se muestra la lista completa de constantes:

CONSTANTE	SIGNIFICADO
E	*Valor matemático* ***e***
LN10	*Logaritmo neperiano de 10*
LN2	*Logaritmo neperiano de 2*
LOGIOE	*Logaritmo decimal de* ***e***
LOG2E	*Logaritmo binario de* ***e***
Pl	*La constante TT (Pi)*
SQRT1_2]	*Resultado de la división de uno entre la raíz cuadrada de dos*
SQRT2	*Raíz cuadrada de 2*

6.4.1.2 MÉTODOS DE MATH

MÉTODO	SIGNIFICADO
abs(n)	Calcula el valor absoluto del número ***n.***
acos(n)	Calcula el arco coseno del número ***n.***
asin(n)	Calcula el arco seno del número ***n.***
atan(n)	Calcula el arco tangente del número ***n.***
ceil(n)	Redondea el número ***n*** (si es decimal) a su valor superior. Por ejemplo, si el número es el 2.3 se redondea a 3.
cos(n)	Coseno de ***n.***
exp(n)	Número ***e*** elevado a ***n.*** Es decir ***e".***
floor(n)	Redondea el número ***n*** (si es decimal) a su valor inferior. Por ejemplo, si el número es el 2.8 se redondea a 2.
log(n)	Calcula el logaritmo decimal de n.
max(a,b)	Siendo ***ayb*** dos números, esta función devuelve el mayor de ellos.
min(a,b)	Siendo ***ayb*** dos números, esta función devuelve el mayor de ellos.
pow(a,b)	Potencia. Devuelve el resultado de a^b.
random()	Devuelve un número aleatorio, decimal entre cero y uno.
round(n)	Redondea ***n*** a su entero más próximo, round(2.3) devuelve 2 y round(2.5) devuelve 3
sin(n)	Devuelve el seno de n.
sqrt(n)	Raíz cuadrada de n.
tan(n)	Tangente de n.

6.4.2 OBJETO DATE

Es el nombre de un tipo de objetos preparados para manejar fechas. Crear un objeto de fecha es usar el constructor de objetos **Date.** En la construcción más simple, la función **Date** no requiere parámetros:

```
let hoy=new Date();
consolé.log(hoy);
```

Escribiría, en la consola de un navegador, algo como:

```
Sat Jun 29 2019 00:58:47 GMT+0200 (Hora de verano romance)
```

Un objeto de tipo **Date** representa un momento concreto de tiempo. El texto que aparece por consola indica que la variable **hoy** hace referencia a un objeto de tipo Date que almacena como fecha el 29 de Junio de 2019 (Sábado) a las 0:58 de la madrugada y, además, se indica que esa fecha está dos horas por encima de la del meridiano de **Greenwich** y que el horario de esa zona en ese momento es el que se usa en verano.

Los objetos Date se pueden crear indicando una fecha concreta de esta manera:

```
hew Datefaño,mes,dia,hora,miñutos,segundos,ms);
```

Ejemplo:

```
let fecha=new Date(2013,5,27,18,12,0,0);
consolé.log(fecha);
```

Muestra:

```
Thu Jun 27 2013 18:12:00 GMT+0200 (Hora de verano romance) ;
```

Observando el resultado queda patente que el mes 5 es junio (en lugar de mayo), porque se considera que enero es el mes cero.

No es obligatorio indicar todos los datos, podemos indicar solo año, mes y día sin indicar la hora:

```
let fecha=new Date(2013,5,27);
```

Hay una tercera opción que es crear una fecha a partir de un número que simboliza el número de milisegundos transcurridos desde el 1 de enero de 1970. Ejemplo:

```
let fecha2=new Date(1000);
consolé.log(fecha2);
```

Saldría:

```
Thu Jan 01 1970 01:00:01 GMT+0100 (Hora estándar romance)
```

Ese resultado muestra un segundo tras el uno de enero de 1970. Con el horario de esa fecha en la franja horaria de España, sería la una de la mañana.

6.4.2.1 MÉTODOS DE LOS OBJETOS DATE

Los objetos de tipo Date disponen de numerosos métodos para realizar operaciones sobre fechas. Por ejemplo:

```
var fecha=new Date();
consolé. log( fecha . getFullYear()); //Escribe el año actual (p.ej. 2019)
```

La lista completa de métodos es la siguiente:

MÉTODO	SIGNIFICADO
getDate()	Devuelve el día del mes de la fecha (de 1 a 31).
getUTCDate()	Devuelve el día del mes de la fecha en formato universal.
getDay()	Obtiene el día de la semana de la fecha (de 0 a 6).
getUTCDay()	Obtiene el día de la semana de la fecha universal.

MÉTODO	SIGNIFICADO
getFullYear()	Obtiene el año (con cuatro cifras).
getUTCFullYear()	Obtiene el año (con cuatro cifras). La diferencia es que UTC usa la fecha global (lo que corresponda al meridiano de Greenwich.
getHours()	Obtiene la hora de la fecha (número que va de 0 a 23).
getUTCHours()	Obtiene la hora en formato universal.
getMiliseconds()	Obtiene los milisegundos en la fecha actual.
getUTCMi1iseconds()	Obtiene los milisegundos en la fecha actual pasada al formato universal.
getMinutes()	Obtiene los minutos.
getUTCMinutes()	Obtiene los minutos en formato universal.
getMonth()	Obtiene el número de mes; de 0, enero, a 11, diciembre.
getUTCMonth()	Obtiene el número de mes en formato universal.
getSeconds()	Obtiene la parte de los segundos de la fecha.
getUTCSeconds()	Segundos en formato universal.
getTime()	Obtiene el valor en milisegundos de la fecha. Número de segundos transcurridos desde el 1 de enero de 1970, respecto a esa fecha.
getTimezoneOffset()	Obtiene el valor en milisegundos usando el formato universal.
setDate(día)	Modifica el día del mes de la fecha. Usa el indicado en el parámetro.
setUTCDate(día)	Versión en formato universal del método anterior.
setFullYear(año)	Modifica el año de la fecha.
setUTCFullYear(año)	Versión en formato universal del método anterior.
setHours(hora)	Modifica la hora.
setUTCHours(hora)	Versión en formato universal del método anterior.
setMi 11iseconds(ms)	Modifica los milisegundos.
setUTCMi11iseconds(ms)	Versión en formato universal del método anterior.
setMiñutes(miñutos)	Modifica los minutos.
setUTCMiñutes(miñutos)	Versión en formato universal del método anterior.
setMonth(mes)	Modifica el número de mes.
setUTCMonth(mes)	Versión en formato universal del método anterior.
setSeconds(segundos)	Modifica los segundos.
setUTCSeconds(seg)	Versión universal.

MÉTODO	SIGNIFICADO
setTime(mi1 isegundos)	Modifica la fecha haciendo que valga la fecha correspondiente a aplicar el número de milisegundos indicados, contados a partir del 1 de enero de 1970.
toDateString()	Muestra la fecha en un formato más humano de lectura. Ejemplo: `consolé.log(fecha.toDateString());` Muestra: `¡Mon Aug 27 2019`
toGMTString()	Igual que la anterior, pero antes de mostrarla convierte la fecha a la correspondiente según el meridiano de Greenwich.
toISOString()	Muestra la fecha en formato 1SO el cual cumple esta plantilla: **yyyy-mm-ddThh:mm:ss.sssZ** Ejemplo: `var d=new Date();` `consolé.log(d.toISOString());` Obtiene, por ejemplo (si son las 1:18:40,268 segundos del día 27 de mayo de 2019 en el meridiano de Greenwich): `¡2019-05-27T01:18:40.268Z`
toLocaleString([codigoLocal[,opciones]])	Sin parámetros muestra la fecha y hora en formato de texto usando la configuración local. `consolé. log( fecha . tol_ocaleString());` Obtiene, por ejemplo: `¡27/5/2019 17:02:55` Pero es posible indicar el código de país: `consolé. log( fecha . tol_ocaleString( "en" ));` Ahora muestra: `¡2019/5/27 5:02:55 PM` Las opciones permite indicar formato numérico, zona horaria, formato de hora, etc. Ejemplo: `consolé.log(fecha.toLocaleString(` `"es",{"timeZone":"UTC"})` `);`

MÉTODO	SIGNIFICADO
toLocaleDateString()	Muestra la fecha (sin la hora) en formato de texto usando la configuración local.
toTimeString()	Muestra la hora (sin la fecha) en formato de texto usando la configuración local.
toString()	Muestra la fecha en formato de texto usando la configuración habitual de JavaScript.
toUTCString()	Versión en formato universal de la anterior.
toJSON()	Muestra la fecha en formato JSON. Obtiene lo mismo que la anterior.

6.4.2.2 MÉTODOS ESTÁTICOS DE DATE

Estos métodos usan directamente el objeto global ***Date.*** Se usan en esta forma: Date.nombreMétodo.

La lista de métodos es:

MÉTODO	SIGNIFICADO
now()	Fecha actual en milisegundos desde el día 1 de enero de 1970.
parse(objetoFecha)	Obtiene una representación en forma de texto de la fecha.
UTC(año,mes,día,horas, minutos, segundos,ms)	Consigue, de la fecha indicada, la forma equivalente en milisegundos. Esta forma mostrará los milisegundos transcurridos desde el 1 de enero de 1970.

6.4.3 EXPRESIONES REGULARES

6.4.3.1 ¿QUÉ SON LAS EXPRESIONES REGULARES?

La mayoría de lenguajes de programación disponen de, al menos, alguna capacidad para trabajar con expresiones regulares. La cuestión es ¿para qué sirven estas expresiones?

Se utilizan, sobre todo, para establecer un patrón que permita establecer condiciones avanzadas en los textos, de modo que podamos validar los textos que encajen con ese patrón. Las labores típicas en las que ayudan las expresiones son: validación de errores, búsqueda de textos con reglas complejas y modificación avanzada de textos.

Por ejemplo, en España un número de identificación personal que se suele exigir a menudo es el N1F ***(Número de Identificación Fiscal).*** Este número está formado por 8 números y una letra cuando coincide con el DN1 ***(Documento Nacional de Identidad)*** de las personas, pero puede haber una letra en la primera cifra en otras circunstancias (si la persona tiene un ***Número de Identificación de Extranjero*** u otras circunstancias) y esa letra solo puede ser **K L, X, *Y*** o **Z.**

Validar todas esas posibilidades, aun sin contar con que la letra final debe cumplir una condición matemática, es muy complejo. Pero una sola expresión regular puede establecer el patrón que cumple el N1F y así facilitar mucho su validación.

6.43.2 EXPRESIONES REGULARES EN JAVASCRIPT

Las expresiones regulares de JavaScript en realidad son objetos de tipo **RegExp.** Debido a la importancia que tienen estas expresiones, JavaScript ha desarrollado una sintaxis que facilita mucho su uso.

Las expresiones regulares se pueden crear indicando las mismas entre dos barras de dividir. Entre ambas barras se coloca la expresión regular. Tras las barras se pueden indicar modificadores, la sintaxis sería esta:

```
/expresionRegular/
```

Así, para crear un objeto que contenga una expresión regular podemos hacer lo siguiente:

```
let expNIF=/[KLXYZ0-9][0-9]{7}[A-Z]/;
```

En esa expresión no hemos utilizado ningún modificador, no es obligatorio hacerlo. Esa expresión es la que permite establecer el patrón general que cumplen los N1F.

Otra forma de crear la expresión, usando una notación más formal es:

```
let expNIF=new RegExp('[KLXYZ0-9][0-9]{7}[A-Z]');
```

Es más habitual utilizar la otra forma al ser más rápida e incluso entendible.

6.4.33 ELEMENTOS DE LAS EXPRESIONES REGULARES

El patrón de una expresión regular puede contener diversos símbolos que se interpretan de forma especial. Para entender las posibilidades de estos símbolos debemos empezar a hablar del método **test** de las expresiones regulares. Este método devuelve verdadero si el texto que evaluamos con la expresión regular cumple el patrón.

Vamos a ver un ejemplo para intentar aclarar la idea:

```
let expresion=/c/;
consolé. log(expresión.test('casa'))//Escribe true ;
consolé.log(expresion.test('Casa[1])); //Escribe false
consolé.log(expresion.test('barbacoa'))i //Escribe true
```

La expresión /c/ la cumple cualquier texto que tenga dentro una letra ***c.*** Por defecto, se distingue entre mayúsculas y minúsculas, por eso la palabra ***Casa*** no cumple la expresión.

Si queremos no distinguir entre mayúsculas y minúsculas, podemos añadir el modificador ***i*** al final de la expresión. La letra i minúscula tras el cierre de la expresión indica que no se debe distinguir entre mayúsculas y minúsculas:

```
let expresion=/c/i;
consolé.log(expresion.test('casa')); //Escribe true
```

```
consolé.log(expresión.test('Casa')); //Escribe true
consolé.log(expresión.test('barbacoa')); //Escribe true
```

Si en lugar de un carácter usamos varios, solo los textos que contengan esos caracteres en ese orden cumplirán el patrón.

```
let expresion=/as/;
consolé.log(expresión.test('casa')); //Escribe true
consolé.log(expresión.test('Casa’)); //Escribe true
consolé.log(expresión.test('asador')); //Escribe true
consolé.log(expresión.test('valiosa')); //Escribe false
```

En el código anterior, solo los textos que tengan dentro las letras ***ay s*** seguidas se consideran válidas. La palabra ***valiosa*** contiene una ***a*** y una s pero no seguidas, por eso la validación es falsa.

La potencia de las expresiones regulares se dispara con el uso de símbolos especiales. Explicamos a continuación los principales.

Que comience por

El símbolo circunflejo (^) obliga a que el siguiente símbolo de la expresión sea el primero que, obligatoriamente, tenga el texto que validemos:

```
let expresion=/^c/;
consolé.log(expresion.test('casa’)); //Escribe true
consolé.log(expresion.test('barbacoa')); //Escribe false
```

Que termine por

El símbolo de dólar ($) hace que el carácter anterior sea, obligatoriamente, el último del texto obliga a que el siguiente símbolo de la expresión sea el primero que, obligatoriamente, tenga el texto que validemos:

```
let expresion=/a$/;
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresion.test('barbacoa')); //Escribe true
consolé.log(expresion.test('río')); //Escribe false, no acaba por “a”
```

Comodín de cualquier carácter

El punto (.) es un símbolo especial de las expresiones regulares que representa un carácter cualquiera. Ejemplo:

```
let expresion=/a.e/;
consolé.log(expresión.test(’caserío')); //Escribe true
consolé.log(expresion.test('aereo')); //Escribe false, debería haber
                                      //un carácter entre “a” y “e”
consolé.log(expresión.test('alambique'));//Escribe false
```

En el código anterior, la expresión /«.e/ la cumplen los textos con una letra ***a*** seguida de otro carácter (el que sea, pero uno) y luego de la letra e.

Símbolos opcionales

Cuando se pone una serie de símbolos entre corchetes, se permite cualquier de ellos. Por ejemplo, la expresión ***l\arft}/*** la cumpliría cualquier texto con una letra ***a*** o una ***erre*** o una ***efe*** o una ***te.*** Gracias a ello podemos conseguir códigos como este:

```
let expresión^/[ao]$/;
                                    //Es válido cualquier texto que acabe por “a” u “o”
consolé.log(expresión.test ('casa')); //Escribe true
consolé.log(expresión.test('barbacoa')); //Escribe true
consolé.log(expresión.test('río')); //Escribe true
```

Podemos incluso indicar rangos. Por ejemplo, la expresión ***[a-z]*** significa cualquier carácter entre la letra ***a y*** la ***z*** minúscula.

```
let expresion=/A[a-z]/;
//Es válido cualquier texto que empiece con minúsculas
consolé.log(expresion.test('casa’)); //Escribe true
consolé.log(expresion.test('Casa’)); //Escribe false
consolé.log(expresión.test('río')); //Escribe true
consolé.log(expresión.test ('árbol')); //Escribe false
```

Puede sorprendernos que se de como falso el texto ***árbol,*** pero es que el carácter ***á*** en Unicode no está entre la letra ***a*** y la z, está mucho después. Por eso hay que usar estas expresiones con cautela. Por ejemplo podríamos cambiar el código por:

```
let expresion=/A[a-záéíóúü]/;
                                    //Es válido cualquier texto que empiece con minúsculas
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresión.test('Casa')); //Escribe false
consolé.log(expresión.test('río')); //Escribe true
consolé.log(expresion.test('árbol')); //Escribe true
```

Carácter no permitido

Si dentro del corchete indicamos un carácter circunflejo, estamos indicando que el contenido no se tiene que cumplir. Por ejemplo: [Aae] significa que no puede aparecer ni la letra ***a*** ni la e, cualquier otra seria válida. Ejemplo:

```
let expresion=/A[AAEIOU]/;
consolé.log(expresion.test('Empresa')); //Escribe false
consolé.log(expresion.test('Casa')); //Escribe true
consolé.log(expresion.test('Ala')); //Escribe false
```

La expresión **/A[AAEIOU]/** significa que **no** empiece por ninguna vocal en mayúsculas.

Símbolos de cardinalidad

Son símbolos que sirven para indicar la repetición de una expresión. Son, concretamente:

SÍMBOLO	SIGNIFICADO
x+	La expresión a la izquierda de este símbolo se puede repetir una o más veces
X'*'	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
X?	El carácter a la izquierda de este símbolo se puede repetir cero o una veces
x{n}	Significa que la expresión ***x*** aparecerá ***n*** veces, siendo n un número entero positivo.
x{n,}	Significa que la expresión ***x*** aparecerá ***n*** o más veces.
x{m,n)	Significa que la expresión ***x*** aparecerá de ***m*** a ***n*** veces.

Ejemplos:

```
let expresion=/^[AEIOU].+a/;
consolé.log(expresión.test("Asa")); //Escribe true
consolé.log(expresión.test("Estación")); //Escribe true
consolé.log(expresión.test("Ea"));//Escribe false
```

La última expresión es falsa porque debería haber un símbolo entre la E y la a. En cambio:

```
let expresion=/^[AEIOU].*a/;
consolé.log(expresión.test("Asa")); //Escribe true
consolé.log(expresión.test("Estación")); //Escribe true
consolé.log(expresión.test("Ea"));//Escribe true
```

Si quisiéramos validar un código postal español, debemos indicar que solo se permiten 5 caracteres numéricos, que tienen que ser eso, exactamente cinco y, por ello, se indican los delimitadores de principio y final.

```
let CPválido=/^[0-9]{5}$/;
```

Paréntesis

Los paréntesis permiten agrupar expresiones. Eso permite realizar expresiones aun más complejas. Ejemplo:

```
let expresion=/([a-z]{2}[0-9]){3}/;
```

Eso obliga a que el texto tenga dos letras de la ***a*** a la ***z*** en un texto, luego un número. El último número entre llaves hace referencia a todo el paréntesis, por lo que las dos letras y el número deberán aparecer tres veces seguidas. Uso posible:

```
let expresion=/([a-z]{2}[0-9]){3}/;
consolé.log(expresión.test("ad3rflhj4")); //Escribe true
consolé.log(expresión.test("a3flj4")); //Escribe false
consolé.log(expresión.test("ab3fgl")); //Escribe false
```

Opciones

La barra vertical "|" permite indicar que se da por válida la expresión de la izquierda o la de la derecha. Ejemplo:

```
let expresion=/^([A-Z][0-9][6])|([0-9][A-Z][6])$/;
consolé.log(expresión.test("A123456")); //Escribe true
consolé.log(expresion.test("1ABCDEF")); //Escribe true
```

Ejemplo de expresión que valida un código postal (formado por 5 números del 00000 al 52999). Los paréntesis ayudan a agrupar esta expresión:

```
let cpl="49345";
let cp2="53345";
let expresion=/^((5[012])|([0-4][0-9]))([0-9]{3})$/;
consolé.log(exp.test(cpl)); //Escribe true
consolé.log(exp.test(cp2));//Escribe false
```

Símbolos abreviados

Son símbolos que se usan con el carácter ***backslash*** y que permite escribir expresiones de forma aún más rápida. Además funcionan muy bien con Unicode:

SÍMBOLO	SIGNIFICADO
\d	Dígito, vale cualquier dígito numérico.
\D	Cualquier carácter salvo los dígitos numéricos
\s	Espacio en blanco.
\s	Cualquier carácter salvo el espacio en blanco
\w	Vale cualquier carácter alfanumérico ***(word).*** Es lo mismo que [a-zA-Z0-9]
\W	Cualquier carácter que no sea alfanumérico. Equivalente a [^A-Za-z0-9]
\0	Carácter nulo.
\n	Carácter de nueva línea.
\t	Carácter fabulador.
****	El propio símbolo \
\"	Comillas dobles.
V	Comillas simples.
\c	Permite representar el carácter c cuando este sea un carácter que de otra manera no sea representable (como [,], /, \,...). Por ejemplo \\ es la forma de representar la propia barra invertida.
\ooo	Permite indicar un carácter Unicode mediante su código octal.
\xflf	Permite indicar un carácter ASC11 mediante su código hexadecimal.
\uflfff	Permite indicar un carácter Unicode mediante su código hexadecimal.

6.4.3.4 PROBLEMAS CON CARACTERES UNICODE

Los símbolos de la tabla anterior permiten abreviar mucho las expresiones y eso permite por ejemplo que cuando pidamos el nombre de usuario de una persona pensemos en esta posible expresión:

```
let usuario=/^[\w]+$/;
```

Al aplicarla nos encontramos con una sorpresa:

```
consolé.log(usuario.test("Jorge")); //Escribe true
consolé.log(usuario.test("kadminl234")); //Escribe true
consolé.log(usuario.test("Orduño")); //Escribe false
```

No da como válido a ***Orduño*** por usar el carácter ***ñ.*** Es decir, \w funciona con caracteres alfanuméricos del alfabeto inglés. La solución no es fácil, pasa por indicar los caracteres directamente, o bien indicar un rango mediante los códigos hexadecimales Unicode[1].

Por ejemplo, los caracteres en árabe ocupan el rango que va desde 0600-06FF en hexadecimal en la tabla Unicode. Con lo cual, para validar si un string contiene un carácter árabe:

```
let reg=/[\u0600-\u06FF]/;
consolé.log(reg.test()); //Escribe true"
consolé.log(reg.test("Hola")); //Escribe false
consolé.log(reg.test("Hoola")); //Escribe true
```

Pero volviendo al problema inicial, es compleja una expresión que admita códigos alfanuméricos de cualquier lengua. Por eso en el estándar ES2018 se añadió un nuevo símbolo: \p. Este símbolo admite indicar entre llaves una propiedad Unicode. Las propiedades Unicode son extensas y pueden indicar, por ejemplo, pertenencia a un rango (como ***Hebrew*** para los símbolos hebreos) o propiedades del texto (como ***Letter*** para indicar una letra)[2].

Utilizar este símbolo y otras características avanzadas para manejar adecuadamente los caracteres Unicode, requieren del uso del modificador **u** al final de la expresión:

```
let usuario=/^[\p{Letter}\p{Number}]+$/u;
consolé.log(usuario.test("Jorge"));
consolé.log(usuario.test("kadminl234"));
consolé.log(usuario.test("Orduño"));
```

Ahora aparecerá la palabra **true** tres veces.

El problema es que este símbolo (\p) no está disponible en todos los navegadores. En el momento de escribir este texto Chrome ya lo reconoce, pero no así Firefox y otros navegadores, ya que la norma es muy reciente. Indudablemente en el futuro será adoptado por todos los navegadores.

---

1 En la dirección https://en.wikipedia.org/wiki/Unicode_block disponemos de los rangos de las diferentes lenguas y tipos de símbolos Unicode.

2 En la dirección https://en.wikipedia.org/wiki/Unicode_character_property se describen las propiedades Unicode, no todas están soportadas en JavaScript.

6.43.5 MÉTODO **EXEC**

Las variables a las que se asigna una expresión regular son en realidad objetos que tienen métodos disponibles. El más importante es **test** que ya le hemos utilizado en los ejemplos anteriores, Pero hay algunos más.

Nos puede ser de utilidad, por su potencia, el método exec. Este método es similar a test en cuanto a que valida una expresión regular en un texto. Pero lo que hace es devolver un array con información sobre cuándo se encontró la expresión en el texto. En este array los elementos son:

- Elemento con índice 0. Es el primer texto encontrado que cumple la expresión.
- Elemento con propiedad **Índex.** Es un número que indica la primera posición en la que se encontró el texto.
- Elemento con propiedad **input.** Es el texto original.
- Se pueden usar subcadenas de búsqueda entre paréntesis y entonces los índices 1,2,.... nos devolverían los textos encontrados con las subcadenas.

Como vemos, el objeto devuelto es complejo, pero permite acciones muy impresionantes:

```
let texto="Este es un texto de prueba";
let expresion=/\w+/;
let res=expresion.exec(texto);
consolé.log(res);
```

Resultado:

```
[ 'Este',
  índex: 0,
  input: 'Este es un texto de prueba',
  groups: undefmed ]
```

El primer elemento (res [0]) muestra el primer texto que cumple la expresión, la cual busca una serie de letras. Por eso es la palabra ***Este*** la primera que lo cumple (después viene un espacio en blanco que ya no es una letra). El elemento siguiente: **res. índex** o también: **res ["índex"],** indica el índice del texto en el que empieza el texto que cumple esa expresión. Finalmente res. input escribe el texto tal cual.

Si variamos la expresión:

```
let texto="Este es un texto de prueba";
let expresion=/(\w+)(\s)/;
let res=expres ion.exec(texto);
consolé.log(res);
```

Ahora la expresión es capaz también de buscar subexpresiones gracias a los paréntesis. El resultado es:

```
[ 'Este ',
  'Este',
```

Es prácticamente el mismo resultado, pero hay dos elementos nuevos, el segundo (res [1]) muestra el texto que cumple la primera expresión entre paréntesis (\w+) el tercero (res [2]) muestra la primera vez que se cumple la segunda expresión entre paréntesis (\s).

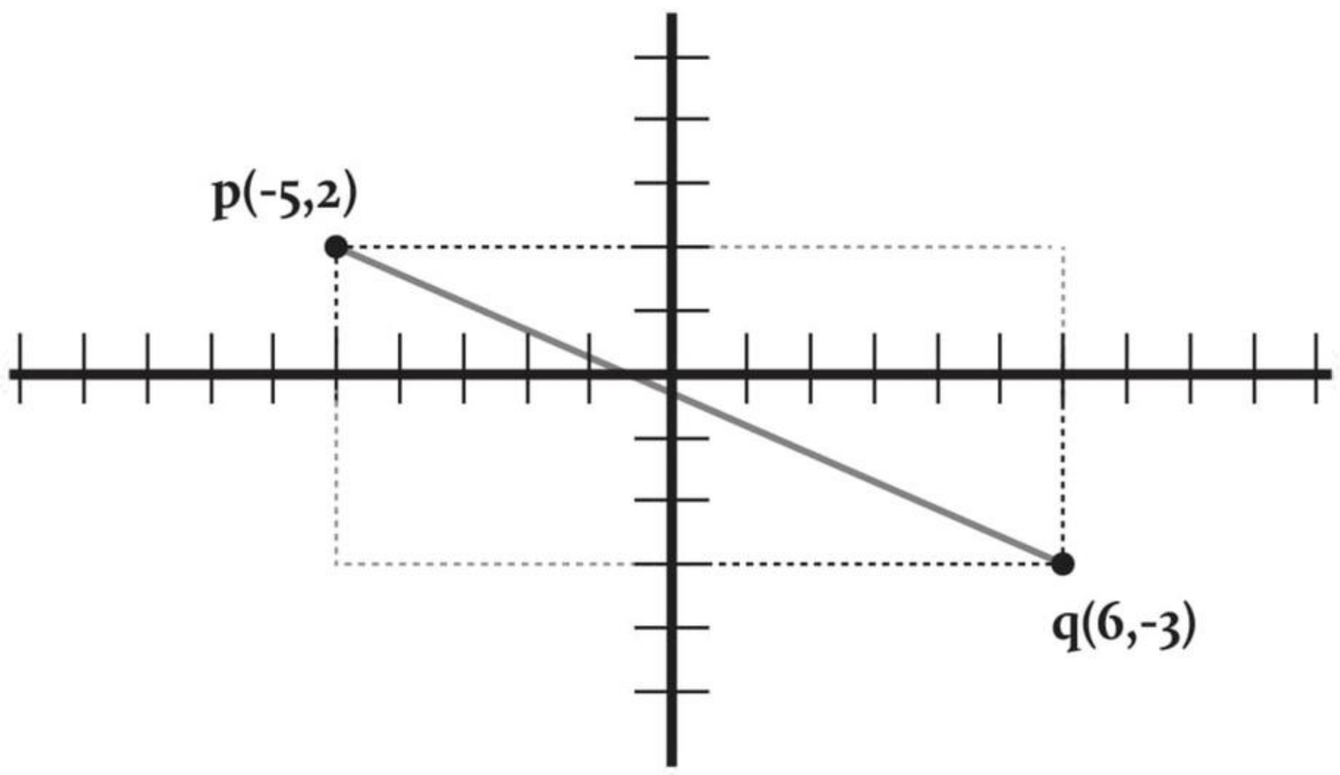
Con pericia, las posibilidades de este método son impresionantes.

6.5 PRÁCTICAS RESUELTAS

Práctica 6.1: Objetos de tipo Punto

• Queremos crear objetos que representen puntos, de ellos necesitamos:

- Que tengan dos propiedades: ***xey.*** Servirán para representar las coordenadas del punto.
- La construcción de los puntos usará una función constructora a la que se le pasan dos números. Si lo que recibe en cada coordenada no es un número, se coloca a cero.
- Un método llamado **cambiar** al que le pasamos dos números y nos permite modificar las coordenadas del número.
- Un método llamado **copia** que retorna una copia del objeto.
- Un objeto llamado **iguales** que recibe un segundo punto y nos dice si ambos puntos son iguales.
- Un método llamado suma que recibe un segundo **punto** y devuelve un tercer punto resultado de sumar las coordenadas de los dos anteriores.
- Un método llamado **obtenerDistancia** que también recibe un segundo punto y nos devuelve la distancia entre ambos puntos, para esta operación, tener en cuenta:



Para calcular la distancia aplicamos el Teórema de Pitágoras, la distancia sale de la fórmula:

^*[distHoriz*^ ~+*liistVerl*~ = -^/aZzs(6 —(-5))^2 + aZ>s(—3 - 2)^2 = Vi l^2 + 5^2 = -y/l46 = 12,08

Figura 6.2: Ejemplo de uso de clases y objetos con herencia en la POO

- Finalmente un método llamado **toString** que retorna un texto con las coordenadas del punto. Por ejemplo (-5,2)

SOLUCIÓN: PRÁCTICA 6.1

Mostramos la solución solo con código JavaScript. Realmente lo requerido es la función Punto, pero se ha añadido código para probar el correcto funcionamiento de esta función:

```
function Punto(x,y){
    this.x=x;
    this.y=y;

    this.cambiar=(x,y)=>{
        this.x=x;
        this.y=y;
    }

    this.copia=()=>(new Punto(this.x,this.y));

    this.suma=(p2)=>(new Punto(this.x+p2.x,this.y+p2.y));

    this.toString=() = >('(${this.x},${this.y})');

    this.obtenerDistancia=function(p2){
        return Math.sqrt(
                Math.pow(Math.abs(this.x-p2.x),2)          +
                Math.pow(Math.abs(this.y-p2.y),2)
            )
    };
}

//Prueba de los métodos y construcciones
var p=new Punto(l,2);
var q=new Punto(6,-3);
//modificamos coordendas de p
p.cambiar(-5,2);
//r será una copia de p
var r=p.copia()
r.x=9;
//comprobamos que r y p son puntos distintos
consolé.log("p: "+p.toString());
consolé.log("r: "+r.toString());

//s es un nuevo punto que toma la suma de coordenadas
//de p y r
var s=p.suma(r);
consolé.log("s: "+s.toString());
//Obtener distancia entre p y q
consolé.log("Distancia entre p y q: ”+p.obtenerDistancia(q));
```

Práctica 6.2: Añadir método de cálculo de media aritmética a los arrays

- Los objetos de tipo Array (en definitiva, los arrays) poseen numerosos métodos que resultan muy útiles.
- Vemos interesante añadir un método a todos los arrays para el cálculo de la media aritmética de sus elementos.
- Modificar el prototipo de los arrays para añadir dicho método.

SOLUCIÓN: PRÁCTICA 6.2

Bastará con modificar el prototipo del objeto Array. A partir de ese momento todos los arrays en el código podrán calcular la media con el nuevo método. El código muestra la creación del nuevo método y su aplicación en nuevos arrays.

El resultado es francamente fácil gracias a la ayuda délos métodos ya implementados en los arrays:

```
Array.prototype.media=function(){
        let suma=this.reduce((anterior,actual)=>(anterior+actual));
        let tamaño=this.length;
        return suma/tamaño;
}

let al=[l,2,3,4,5,6];
consolé.log(al.media()); //Escribe 3.5
let a2=[10,13,16,21,28,19,45,32,20];
consolé.Iog(a2. media ()); //Escribe 22.6666666666666668
```

Práctica 6.3: Objeto para fracciones y añadir métodos para el mcd y el mcm en la clase Math

- Queremos crear objetos que representen fracciones.
- Algunos métodos de estos objetos usan las operaciones matemáticas de cálculo de máximo común divisor y mínimo común múltiplo. Debemos crear estas funciones, pero mejor si son métodos estáticos del propio objeto **Math** (al igual que sus otros métodos vistos en esta misma unidad).
- El máximo común divisor de dos números naturales es el mayor número por el que podemos dividir de forma exacta a ambos números. El mínimo común múltiplo es el número múltiplo más pequeño de ambos números a la vez. Para ambos cálculos podemos usar estos detalles que nos van a facilitar la implementación de ambos métodos:

mcm(a,b)=a*b/mcd(a,b)

mcd(a,b)=b si el resto de a entre b es cero

mcd(a,b)=mcd(b,a%b) si el resto de a entre b no es cero

- Una fracción consta de un numerador y un denominador, ambos son números enteros. La idea es que cuando se usen estos objetos no se permita modificar directamente el numerador y el denominador, sino que podremos hacerlo mediante métodos.
- El método **getNumerador** obtendrá el valor del numerador, **getDenominador** obtendrá el denominador.
- Los métodos **setNumerador** y **setDenominador** permitirán cambiar tanto el numerador como el denominador. Ambos requieren del nuevo valor.
- El método **cambiarFracción** permitirá cambiar a la vez numerador y denominador.
- Implementar un método llamado **toString** que permita retornar la fracción en el formato ***numerador/denominador*** (ejemplo 5/6).
- Implementar un método llamado **simplificar** que permita simplificar la función. Eso se consigue calculando el **mcd** del numerador y el denominador. Y dividiendo el numerador y el denominador por ese mcd. Ejemplo:

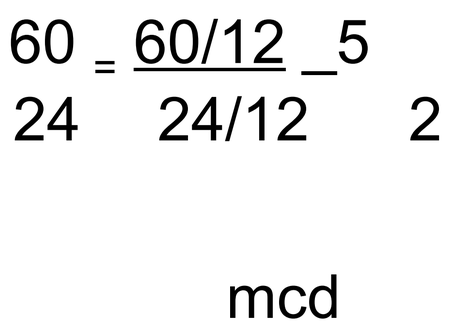


Figura 6.3: Simplificación de fracción

- Implementar un método para la **suma** de fracciones que requiere del cálculo del mínimo común múltiplo de los denominadores. Ejemplo:

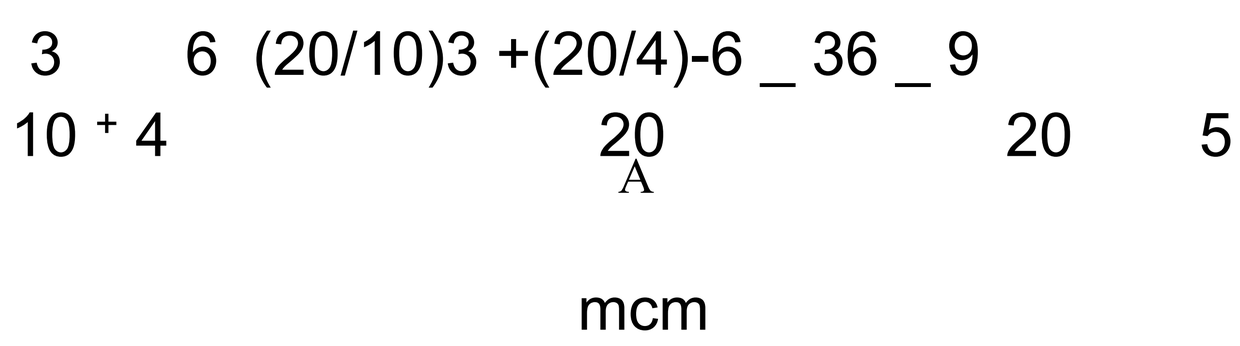


Figura 6.4: Suma de fracciones

El mcm de los denominadores es 20, se divide este número entre cada denominador y se multiplica por cada numerador. Se suman los numeradores multiplicados y finalmente se simplifica la fracción.

El resultado debe de ser una nueva fracción con la suma.

- Crear método para la **resta** de fracciones. Se hace igual que la suma pero restando los numeradores.
- Método para la **multiplicación** de fracciones. Se multiplican los numeradores y los denominadores entre sí.
- Método para la **división** de fracciones. Se multiplica de forma cruzada el numerador y el denominador.
- En todas las operaciones matemáticas, se debe de simplificar la fracción resultante.

SOLUCIÓN: PRÁCTICA 6.3

Esta es una práctica ya más compleja por la cantidad de operaciones. La forma de abordarla es ir solventando cada método y probando que funciona bien.

La solución la dejaremos en el mismo archivo, pero la mostraremos por partes. En primer lugar, se realizan los métodos para cálculo de máximo común divisor **(mcd)** y de mínimo común múltiplo **(mcm).** Ambos serán métodos accesibles desde el objeto **Math.** En este caso, no los añadimos al prototipo porque Math no se ha creado para crear objetos a partir de él (no permite ***instanciar).*** De hecho, no hay función constructora, luego no hay objetos de tipo ***Math.***

Usar métodos desde el propio tipo de objeto es utilizar lo que se conoce habitualmente como **métodos estáticos.** Pero, en realidad JavaScript no maneja métodos estáticos de la manera que los manejan lenguajes como Java. Es un artificio conceptual porque la realidad es que Math es un objeto y no una clase. Pero de forma conceptual se manejan este tipo de métodos de la misma forma.

Toda esta explicación teórica nos confundirá. Pero la primera idea es crear ambos métodos asignándolos a Math y probaremos que ambos funcionan:

```
Math.mcd=function(a,b){
    //debemos asegurarnos de que tenemos dos números
    //naturales, de otro modo devolvemos NaN
    //si hay decimales, los quitamos
    a=parselnt(Number(a));
    b=parselnt(Number(b));
    if(isNaN(a) II isNaN(b)) {
        return NaN;
    }
    //si algún número es cero, devolvemos cero
    else if(a==0 || b==0) return 0;
    //tomamos su valor positivo por si acaso
    a=Math.abs(a);
    b=Math.abs(b);

    //Algoritmo
    if(a%b!=0) return Math.mcd(b,a%b);
    else return b;
```

```
}

Math.mcm=function(a,b){
        //eliminamos errores de la misma forma que en el mcd
        arparseInt(Number(a));
        b=parselnt(Number(b));
        if(isNaN(a) II isNaN(b)) {
                return NaN;
        }
        else if(a==0 || b==0) return 0;
        a=Math.abs(a);
        b=Math.abs(b);

        //resultado basado en el mcd creado antes
        return (a*b)/Math.mcd(a,b);
}
```

Para hacer pruebas podemos escribir este código

```
consolé.log(Math.mcd(60,24)); //Escribe 12
consolé.log(Math.mcm(60,24)); //Escribe 120
```

Una vez comprobado el funcionamiento (convendría hacer más comprobaciones). Este podría ser el código de la función constructora de fracciones:

```
function Fracción(numerador,denominador){
        //puesto que el numerador y denominador son privados
        //se declararan como variables de la función constructora
        var _numerador=numerador;
        var .denominador=denominador;
        //si no se han recibido números, crearemos fracciones nulas
        if(isNaN(parseInt(Number(numerador)))) return nuil;
        if(isNaN(parseInt(Number(denominador)))) return nuil;

//Los métodos sí serán públicos
this.getNumerador=()=>_numerador;

this.getDenominador=()=>.denominador;

this.setNumerador=function(numerador){
        //solo cambiamos el numerador si el número es válido
        i f(isNaN(parseInt(Number(numerador)))==false)
                _numerador=parselnt(numerador);//quitamos decimales
};

this.setDenominador=function(denominador){
        //solo cambiamos el denominador si el número es válido
```

```
        if(isNaN(parseInt(Number(denominador)))==false)
            _denominador=parselnt(denominador);//quitamos decimales
};

this.cambiarFraccion=function(numerador,denominador){
        //aprovechamos los métodos anteriores
        this.setNumerador(numerador);
        this.setDenominador(denominador);
}

this.toString=()=>(_numerador+"/"+_denominador);

    this.simpli ficar=function(){
            //si el numerador o denominador son cero,                no simplificamos:
            i f(.numerador!=0     &<& .denominador!=0){
                    //miramos los signos
                    i f(Math.sign(.numerador)==Math.sign(.denominador)){
                            // signos iguales ambos números les pasamos a positivos
                            _numerador=Math.abs(_numerador);
                            .denominador=Math.abs(_denominador);
                    }
                    else{
//con signos distintos, dejamos el numerador en negativo si no lo está
                            i f(.numerador>=0) {
                                    _numerador=-_numerador;
                                    _denominador=-_denominador;
                            }
                    }
                    let mcd=Math.mcd(.numerador,.denominador);
                    _numerador/=mcd;
                    _denominador/=mcd;
            }
    }
    this.suma=function(f){
            let mcm=Math.mcm(_denominador,f.getDenominador());
            let denominadorRes=mcm;
            let numeradorRes=(mcm/_denominador)*_numerador +
                                    (mcm/f.getDenominador())*f.getNumerador();
            let res=new Fraccion(numeradorRes,denominadorRes);
            res.simpli ficar();
            return res;
    }
    this.resta=function(f){
            let mcm=Math.mcm(_denominador,f.getDenominador());
            let denominadorRes=mcm;
```

```
        let numeradorRes=(mcm/_denominador)*_numerador -
                                (mcm/f.getDenominador())*f.getNumerador();
        let res=new Fraccion(numeradorRes,denominadorRes);
        res.simplificar();
        return res;
    }
    this.multiplica=function(f){
        let res=new Fraccion(
                        _numerador*f.getNumerador(),
                        _denominador*f.getDenominador()
        );
        res.simplificar();
        return res;

    }

    this.divide=function(f){
        let res= new Fraccion(
                        (_numerador*f.getDenominador()),
                        (_denominador*f.getNumerador())
        );
        res.simplificar();
        return res;
    }

}
```

Pruebas:

```
var fl=new Fraccion(6,10);
var f2=new Fraccion(6,4);
var f3=new Fraccion(60,24);
fl.setNumerador(3);
consolé.log(fl.toString()); //Escribe 3/10
consolé.log(f2.toString()); //Escribe 6/4
consolé.log(f3.toString()); //Escribe 60/24
f3.simplificare);
consolé.log(f3.toString()); //Escribe 5/2
var fSuma=fl.suma(f2);
var fResta=fl.resta(f2);
var fMultiplica=fl.multiplica(f2);
var fDivide=fl.divide(f2);
consolé.log(fSuma.toString()); //Escribe 9/5
consolé.log(fResta.toString()); //Escribe -6/5
consolé.log(fMultiplica.toString()); //Escribe 9/20
consolé.log(fDivide.toString()); //Escribe 1/5
```

Práctica 6.4: Herencia

- Crea un tipo de objeto que sirva para representar figuras
- Sus propiedades son:
  - Marca, un texto
  - Modelo, un texto
  - Memoria RAM, un número que indica GB de capacidad
  - Capacidad Disco Duro, un número que indica GB de capacidad
  - Pulgadas de pantalla, un número que indica Pulgadas
- Métodos de los ordenadores
  - toString, sirve para obtener en forma de texto los detalles del ordenador
- Al crear un ordenador se pueden indicar todos los valores, pero por defecto (sin ser obligado indicarlos) se toman como valores 17 pulgadas, 512 GB de disco duro y 4 GB de RAM. La marca y el modelo sí son necesarias
- Crear otro tipo de objeto que represente ordenadores portátiles, los cuales heredan todo lo de los ordenadores pero añaden una propiedad llamada autonomía, que es un número que expresa horas. Se construye este objeto igual que los ordenadores, pero pudiendo añadir la autonomía (por defecto, 4 horas). Por defecto, en los portátiles las pulgadas son 12 y el disco duro 256 GB.

SOLUCIÓN: PRÁCTICA 6.4

La herencia es la capacidad de que un objeto (en otros lenguajes se hace con clases) herede las capacidades de otro objeto. En JavaScript no es posible como en otros lenguajes, pero se pueden embellecer los prototipos para que adopten las propiedades de otros objetos. Primero nos centraremos en el objeto **Ordenador:**

```
funetion Ordenador(marca,modelo,ram=4,disco=512,pulgadas=17){
        this.marca=marca;
        this.modelo=modelo;
        this.ram=ram;
        this.disco=disco;
        this.pulgadas=pulgadas;
        this.toString=function(){
        return "Marca: "+this.marca+"\n"+
                "Modelo: "+this.modelo+"\n"+
                "RAM: "+this.ram+"GB\n"+
                "Disco duro: "+this . disco+"GB\n"+
                "Pulgadas: "+this.pulgadas+" pulgadas\n";
      } }
```

Es una función constructora simple. Lo interesante está en el objeto **Portátil,** para poder heredar todo lo realizado en los ordenadores deberemos asignar un nuevo objeto de tipo Ordenador al prototipo (mediante la propiedadproto). De esta forma en un momento tendremos heredadas las propiedades y métodos y cada objeto de tipo Portátil dispondrá de ellas.

Por supuesto, las nuevas propiedades se rellenan de forma explícita y, además, la función **toString** se actualizará añadiendo una nueva línea.

En el siguiente código se expone la función constructora de los portátiles y código para comprobar la corrección de los tipos creados:

```
function Portati1(marca,modelo,ram=4,disco=256,pulgadas=13,autonomia=4){
        this.proto=new Ordenador(marca,modelo,ram,disco,pulgadas);
        this.autonomia=autonomia;
        this.toString=function(){
                return this.proto.toString() +
                        "Autonomía: "+this.autonomia+" horas\n";
        }
}


var ol=new 0rdenador("HP","EliteDisplay",8,256,23);
var o2=new 0rdenador("Del 1","Inspiron AI0");
var pl=new Portati1("Apple","Macbook Air",8,128,13,12);
var p2=new Portati1("Acer","Aspire");
consolé.log(ol.toString());
consolé.Iog(o2.toString());
consolé.log(pl.toString());
consolé.Iog(p2.toString());
```

Práctica 6.5: Diferencia entre fechas

- Haz una página que pida una fecha usando la notación, dia/mes/año.
- Si hay fallo en la fecha se pide de nuevo. No se comprobará la corrección de la fecha, pero sí que tenga dos dígitos para el día, dos para el mes y dos para el año.
- Se pedirá una segunda fecha y se validará también.
- Finalmente se crearán dos fechas de tipo Date y se mostrará la diferencia en días entre ellas.

SOLUCIÓN: PRÁCTICA 6.5

Los objetos de tipo Date son los idóneos para este tipo de práctica que además utiliza técnicas de validación. Bastará con tomar el tiempo justo antes de que aparezca el cuadro. El tiempo le tomaremos en formato día/mes/año.

Código JavaScript (deberá ser invocado por una página web):

```
//Obtiene un objeto Date de un texto tipo dd/mm/yyyy
function getFecha(fechaString){
        const VAL_FECHA_G=/(\d{2})\/(\d{2})\/(\d{4})/;
        let res=VAL_FECHA_G.exec(fechaString);
        i f(res==nul1)
                return nuil;
        else{
                //exec extraerá en un array cada apartado entre
                //paréntesis. Empezando por el elemento 1
                // Así en el 1 tendremos el día, en el 2 el mes
                // y en el 3 el año
                //Los objetos Date requieren pasar este orden al revés
                //Al mes hay que restarle 1
                return new Date(res[3],res[2]-1,res[1]);
        }
}

const VAL_FECHA=/\d{2}\/\d{2}\/\d{4}/;
var fechal;
var fecha2;
do{
        fechal=prompt("Escriba la primera fecha                (formato dd/mm/yyyy)");
}while(VAL_FECHA.test(fechal)==false);

do{
        fecha2=prompt("Escriba la segunda fecha                (formato dd/mm/yyyy)");
}while(VAL_FECHA.test(fecha2)==false);

//getFecha devuelve una fecha, de esa fecha usamos el
//método getTime para obtener los milisegundos desde 1970
//que representa esa fecha
let msFechal=getFecha(fechal).getTime();
let msFecha2=getFecha(fecha2).getTime();
//restamos los milisegundos y les pasamos a días
let diferencia=(msFechal-msFecha2)/(1000*60*60*24);
alert(xLa diferencia en días de esa fecha es ${diferencia}');
```

6.6 PRÁCTICAS RECOMENDADAS

Práctica 6.6: Validar URLs

- Crea una aplicación web que sirva para validar URLs
- Recordamos que una URL está formada por:
  - Protocolo, seguido de dos puntos y, según el protocolo, por hasta tres barras inclinadas (/), pero puede incluso no haber ninguna.
  - Puede haber un nombre de usuario (no es obligatorio) que constará solo de textos, números y/o guiones. Además, se pueden separar varias palabras con puntos, tipo **jorge.sanchez** por ejemplo).
  - Después del nombre de usuario (si le hay) puede haber dos puntos seguidos de la contraseña. En la contraseña vale cualquier serie de caracteres. Hay que hacer notar que si hay contraseña debe haber nombre de usuario seguido de dos puntos, pero puede haber nombre de usuario y no haber contraseña (los dos puntos no se pondrían tampoco).
  - Si hay nombre de usuario se coloca al final de la contraseña un símbolo de arroba. Si no hay contraseña se coloca detrás del nombre del usuario.
  - Después el nombre de la máquina, cuyos caracteres son los mismos que para el nombre de usuario. Al menos un nombre de máquina consta de dos palabras separadas por un punto, pero puede haber más palabras.
  - Opcionalmente, se pueden poner dos puntos seguidos de un número de puerto. El número de puerto es una serie de cifras numéricas.
  - También opcionalmente puede haber al final una ruta, la cual comenzaría por el símbolo de dividir (/) y luego palabras y símbolos de dividir. Esas palabras pueden tener puntos entre medias.
- Aunque puede haber cadena de búsqueda al final de todo lo anterior, no complicaremos más la validación. Pero al menos, sí se permitirá (sin ser obligatorio) que al final haya cualquier serie de caracteres si van precedidos por el símbolo literal del cierre de interrogación (?).

Práctica 6.7: Validar NIF

- Crea una aplicación web que sirva para validar NIFs indicando si es correcto o no.
- Un NIF consta de una posible primera letra que puede ser una X, una Y o una Z. Puede no haber letra y entonces habrá un número.
- Después habrá siete números más.
- Finalmente habrá una letra que cumple una fórmula.

- La fórmula de la letra consiste en dividir los números del N1F entre 23 y tomar el resto. Cuando la primera cifra es una X, se cambia por el número 0 para hacer este cálculo. La Y se cambia por el número 1 y la Z por el número 2. El resto obtenido se comprueba en esta serie:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

Práctica 6.8: Acceso prohibido

- Crea, una página web que muestre el texto "Bienvenido" pero si la hora es de 9:00 a 14:00 y es un día que no es ni sábado ni domingo.
- Si no se cumple esa premisa, entonces se mostrará el texto "Página cerrada"

Práctica 6.9: Buscaminas con objetos

- Crea una página que sirva para mostrar el mapa del juego de Buscaminas, según las directrices dadas en la Práctica 5.11 ***"Mapa buscaminas",*** en la página 198 .
- Soluciona el problema usando un objeto (o un tipo de objetos, mejor dicho) que admitan un constructor al que indicamos el número de filas y columnas y el número de minas a dibujar.
- En la construcción se conseguirá colocar los números de minas en las casillas sin mina.
- Además, habrá un método llamado **dibujar** que será el encargado de dibujar el tablero.
- Otro método llamado **recalcular** será capaz de recalcular el tablero de nuevo. Si a este método no se le pasan parámetros, recalcula usando los parámetros de filas, columnas y minas que se le indicaran anteriormente. Pero el método admite que se cambien dichos parámetros

Práctica 6.10: Modificar objetos Map

- Vimos que los objetos Map asocian claves a valores. Las claves no se pueden repetir, pero los valores sí.
- Crea, para este tipo de objetos, un método llamado **invertirMapa** que devuelva un nuevo objeto de mapa donde las claves son los valores del mapa original. Los valores del nuevo mapa serán arrays que contengan las claves donde ese valor se repetía.
- Por ejemplo, si teníamos un mapa con estas claves:
  - Clave 1: Valor: "Óptimo"
  - Clave 2: Valor: "Notable"
  - Clave 3: Valor: "Notable"

- Clave 4: Valor: "Excelente"
- Clave 5: Valor: "Mejorable"
- Clave 6: Valor: "Excelente"
- Clave 7: Valor: "Notable"

El método será capaz de obtener este otro mapa:

- Clave: "Óptimo", Valor: [1]
- Clave: "Notable", Valor:[2,3,7]
- Clave: "Mejorable", Valor:[5]
- Clave: "Excelente", Valor:[4,6]

Práctica 6.11: Baraja de cartas

- Crea un tipo de objetos que sirvan para representar **Cartas.** Estos objetos tendrán dos propiedades:
  - **palo.** Que será un número de 1 a 4 (donde 1 significa Oros, 2 Espadas, 3 Bastos y 4 Copas)
  - **valor.** Un número del 1 al 10. (8=sota, 9=caballo, 10=rey)
- Los objetos de este tipo se construyen indicando el palo y el valor. Si hay fallos en los datos, se devuelve un objeto nulo en la creación.
- Las cartas tendrán estos métodos:
  - **darValor.** Que recibe un número de palo y un valor para la carta para, con ellos, modificar la carta. Ante datos incorrectos no cambia nada en la carta.
  - **toString.** Método habitual (y estándar) para devolver en forma de texto entendible el valor de la carta. Por ejemplo: **As *de Oros.***
- Además, habrá otro tipo de objeto: **Baraja.** La idea es que represente una baraja de cartas españolas. Tendrá los siguientes detalles:
  - La baraja la forman 40 cartas. Para ello tendrá la propiedad **cartas** que será un array de 40 cartas.
  - Al construir la Baraja se rellenan las cartas en el siguiente orden: por palos y cada palo con las cartas del 1 al 10. No se podrá acceder directamente al array fuera de la función constructora.
  - El método **barajar** permite barajar las cartas. Es decir, desordenarlas de forma aleatoria.
  - El método toString permite obtener la baraja en forma de texto para saber cómo están ordenadas las cartas.

6.7 RESUMEN DE LA UNIDAD

- La Programación Orientada a Objetos es un modelo que se utiliza ampliamente en el desarrollo de todo tipo de aplicaciones debido a las ventajas que aporta.
- En este paradigma las aplicaciones se entienden como un conjunto de objetos que se comunican entre sí. Podemos concentrarnos en codificar todos los aspectos de cada objeto de manera independiente y así, concentrarnos en tareas más simples y programar de forma más efectiva.
- Aunque JavaScript no sigue el modelo orientado a clases de otros lenguajes como Java o C++, se considera un lenguaje orientado a objetos, aunque con ciertas peculiaridades y limitaciones en la implementación de este modelo. En todo caso, programar código en JavaScript implica manipular objetos en todo momento.
- El operador punto permite acceder a las propiedades y métodos de un objeto: **coche, color, coche, acelerar (10),** etc. También es posible usar corchetes indicando la propiedad o método entre comillas: **coche["color"] , coche["acelerar"](10).**
- La sintaxis para indicar un objeto vacío puede ser mediante **new Ob ject ()** o bien indicando apertura y cierre de llaves: { }. En todo caso, tras crear el objeto las propiedades y métodos se pueden crear sobre la marcha de esta forma:

```
objeto.nombrePropiedad=valor;
objeto.nombreMétodo=function...
```

- Es posible indicar propiedades y métodos en la declaración si dentro de las llaves usamos la sintaxis siguiente:

```
objeto={
  propiedadl:valorl,
  propiedad2:valor2,
  ■ ■■
  método1: function(...) ,   {...}
  ...
}
```

- La palabra reservada **this** se usa para llegar al objeto en sí que se manipula es fundamental para crear muchas de las funciones y métodos de trabajo con objetos.
- Hay una forma de crear objetos que utiliza una función especial llamada **constructor,** que permite crear objetos que comparten el mismo tipo de objeto. Después (a diferencia de otros lenguajes) cada lenguaje puede personalizar sus métodos y propiedades, por lo que esta forma de crear objetos no es como la creación por clases de otros lenguajes.
- El operador **instanceof** permite saber el tipo de un objeto concreto. Todos los objetos son objetos Object y los objetos creados a partir de un constructor harán que **instanceof** también devuelva verdadero con el nombre de su constructor (son de ese tipo).

- JavaScript utiliza un mecanismo llamado **prototipo** para conseguir mecanismos de herencia entre objetos. Cuando varios objetos se crean a partir del mismo constructor, todos ellos comparten un objeto común (sobre el que tienen un enlace) que es el prototipo de esos objetos. A ese prototipo le podemos asignar métodos y propiedades y, entonces, serán métodos y propiedades que todos los objetos compartirán. En realidad hay un prototipo común a todos los objetos que es el objeto **Object.**

- El formato JSON es un formato documental muy relacionado con JavaScript porque usa una notación similar a la de los objetos literales de JavaScript. No permite métodos y el nombre de las propiedades debe ir entrecomillado. Es un formato ideal para almacenar datos y que estos se manipulen desde JavaScript o desde otros lenguajes.

- En JavaScript existen numerosos objetos ya predefinidos. Algunos de los más útiles son:
  - El objeto **Math** que permite realizar operaciones matemáticas avanzadas como: cálculos trigonométricos, potencias, logaritmos, números aleatorios, etc.
  - El objeto **Date** para el manejo de fechas y horas que posee numerosos métodos para realizar las operaciones más habituales.
  - Las expresiones regulares son en realidad objetos **RegExp.** Admiten una sintaxis muy cómoda que permite indicar expresiones regulares entre signos de dividir. A través de las expresiones regulares podemos validar que un texto cumpla unas determinadas y complejas reglas sintácticas.
  - Las expresiones regulares son muy útiles y conviene aprender sus símbolos y posibilidades por el tremendo ahorro y eficiencia a la hora de escribir código.

6.8 TEST DE REPASO

1.- **Un objeto llamado personal tiene una propiedad llamada nombre. ¿Cuáles de estas formas son válidas?**

a) **personal.nombre**
b) **personal["nombre"]**
c) **personal['nombre']**
d) **personal[nombre]**
e) **personal{"nombre"}**
f) **personal{"nombre"}**
g) **personal{nombre}**
h) Ninguna es válida

2.- **Queremos crear un objeto llamado dado y en él un método llamado nueva-recolocacion. ¿Cuáles de estas sintaxis son válidas? (el cuerpo de la función le dejamos con puntos suspensivos).**

a)
```
let dado={
        nueva-recolocacion:function()
                                {.■■}
```
b)
```
let dado={
"nueva-recolocacion":
                    function(){...}
```
c)
```
let dado={};
dado.nueva-recolocacion=
                    function(){}
```
d)
```
let dado={};
    dado."nueva-recolocacion"=
                        function(){}
```
e)
```
let dado={};
dado["nueva-recolocacion"]=
                        function(){}
```

3.- **¿Qué propiedad de la POO es la que se relaciona con la capacidad de ocultar algunos métodos *y* propiedades de los objetos?**

a) Herencia
b) Polimorfismo
c) Abstracción
d) Encapsulamiento

4.- **¿Qué propiedad de los objetos permite acceder a su prototipo?**

a) **getPrototype**
b) **__proto___**
c) **prototype**
d) Ningún objeto puede acceder a su prototipo

5.- **¿Qué escribe este código por pantalla?**

```
function f(){
      consolé.log(this.nombre);
}
function Persona(nombre){
      this.nombre=nombre,
      this.escribirNombre=f
}
let personal=new
                    Persona("Amelia");
let persona2=new
                    Persona("Darío");
personal.escribirNombre();
persona2.escribirNombre();
```

a) **f**
   f
b) **function**
   function
c) **Amelia**
   Darío
d) Ocurre un error: **this** solo se puede usar en métodos de objetos.

6.- **¿Puede un objeto modificar las propiedades y métodos que hereda de su prototipo?**

a) Sí
b) No
c) Solo las propiedades
d) Solo los métodos

7.- **¿Qué bucle permite recorrer las propiedades de un objeto?**

a) **for . . in**
b) **for. .of**
c) **for**
d) Todos ellos

8.- **¿Se pueden añadir nuevos métodos a las clases predefinidas a través de su prototipo?**

a) No

b) Sí

c) Solo en el caso de Arrays, Mapas y Conjuntos

d) Solo para los Arrays y Strings

9.- **¿Cuáles de estos elementos no pueden estar en un documento JSON?**

a) Propiedades sin entrecomillar

b) Propiedades con comillas simples

c) Propiedades con comillas dobles

d) Métodos sin entrecomillar

e) Métodos con comillas simples

f) Métodos con comillas dobles

10.- **Queremos convertir un objeto llamado *objl* de JavaScript a su formato JSON correspondiente. ¿Qué instrucción lo consigue?**

a) objl.stringify()

b) objl.parse()

c) JSON.stringify(objl)

d) JSON.parse(objl)

11.- ¿Cómo podemos conseguir la raíz cuadrada de 2 en JavaScript?

a) Math.sqrt(2)

b) Math.SQRT2

c) Con las dos anteriores

d) No se puede obtener la raíz cuadrada en JavaScript

12.- **Queremos crear un objeto de fecha que represente el día 3 de mayo de 2017 ¿Qué expresión lo consigue?**

a) let f=new Date(3,5,2017);

b) let f=new Date(2017,5,3);

c) let f=new Date(5z3,2017);

d) let f=new Date("3/5/2017","es");

13.- **¿Qué método permite ver una fecha en formato local de España?**

a) toLocaleString("es");

b) toString("es");

c) Cualquiera de los dos anteriores

d) No se puede especificar código de idioma, se debe usar la configuración local del usuario

14.- **¿Qué significa el código \D en las expresiones regulares?**

a) Cualquier código alfanumérico

b) Cualquier código alfabético

c) Cualquier dígito numérico

d) Cualquier carácter no alfanumérico

e) Cualquier carácter no alfabético

f) Cualquier carácter no numérico

15.- **¿Qué textos validarían esta expresión regular?/[0-9] {3}$/**

a) Alarmal23

b) Alarmal23$

c) 123

d) 123$

e) 123$Alarma

f) Alarmal23$Alarma

16.- **¿Qué textos validarían esta expresión regular?/A [A\w] /**

a) w

b) www

c) labe

d) abe

e) Aw

f) Awww

g) Alabc

h) Aabc

i) -w

j) -www

k) -abe

l) -labe

UNIDAD 7

MANIPULACIÓN DEL MODELO DE OBJETOS DEL DOCUMENTO

OBJETIVOS

- Analizar el funcionamiento del modelo de objetos del documento y del navegador
- Describir la estructura y relación entre los distintos objetos del documento y el navegador
- Reconocer el funcionamiento de los principales objetos pertenecientes al objeto window
- Utilizar propiedades y métodos de los objetos del BOM para obtener información del usuario
- Analizar y modificar el texto y estructura de los elementos del documento
- Recorrer los elementos del documento obteniendo información relevante
- Analizar y modificar el formato CSS de los elementos del documento
- Crear y aplicar temporizadores en las aplicaciones web
- Gestionar cookies para almacenar información tanto temporal como persistente

CONTENIDOS

7.1 EL OBJETO WINDOW
- 7.1.1 BOM, MODELO DE OBJETOS DEL NAVEGADOR
- 7.1.2 OBJETO NAVIGATOR
- 7.1.3 OBJETO SCREEN
- 7.1.4 OBJETO LOCATION
- 7.1.5 OBJETO HISTORY
- 7.1.6 OTRAS PROPIEDADES Y MÉTODOS DE WINDOW

7.2 ¿QUÉ ES EL DOM?
- 7.2.1 EL MODELO DE OBJETOS DEL DOCUMENTO
- 7.2.2 EL OBJETO DOCUMENT
- 7.2.3 TIPOS DE NODOS

7.3 SELECCIÓN DE ELEMENTOS DEL DOM
- 7.3.1 SELECCIÓN POR EL IDENTIFICADOR
- 7.3.2 POR ETIQUETA
- 7.3.3 OBJETOS NODELIST
- 7.3.4 POR CLASE
- 7.3.5 POR SELECTOR CSS

7.4 OBTENER Y MODIFICAR DOM
- 7.4.1 MANIPULACIÓN DE ATRIBUTOS
- 7.4.2 MANIPULACIÓN DEL CONTENIDO DE LOS ELEMENTOS
- 7.4.3 MODIFICAR CSS
- 7.4.4 OBTENER ATRIBUTOS DATA
- 7.4.5 NAVEGAR POR EL DOM
- 7.4.6 AÑADIR ELEMENTOS
- 7.4.7 REEMPLAZAR ELEMENTOS
- 7.4.8 ELIMINAR ELEMENTOS
- 7.4.9 COLECCIONES VIVAS
- 7.4.10 OTRAS PROPIEDADES Y MÉTODOS DE LOS ELEMENTOS

7.5 TEMPORIZADORES
- 7.5.1 ¿QUÉ SON LOS TEMPORIZADORES?
- 7.5.2 SETTIMEOUT
- 7.5.3 SETINTERVAL

7.6 COOKIES
- 7.6.1 ¿QUÉ SON LAS COOKIES?
- 7.6.2 LECTURA Y GRABACIÓN DE COOKIES
- 7.6.3 FECHAS DE EXPIRACIÓN
- 7.6.4 RUTA Y DOMINIO DE LAS COOKIES
- 7.6.5 BORRAR COOKIES

7.7 PRÁCTICAS RESUELTAS

7.8 PRÁCTICAS RECOMENDADAS

7.9 RESUMEN DE LA UNIDAD

7.10 TEST DE REPASO

7.1 EL OBJETO WINDOW

7.1.1 **BOM,** MODELO DE OBJETOS DEL NAVEGADOR

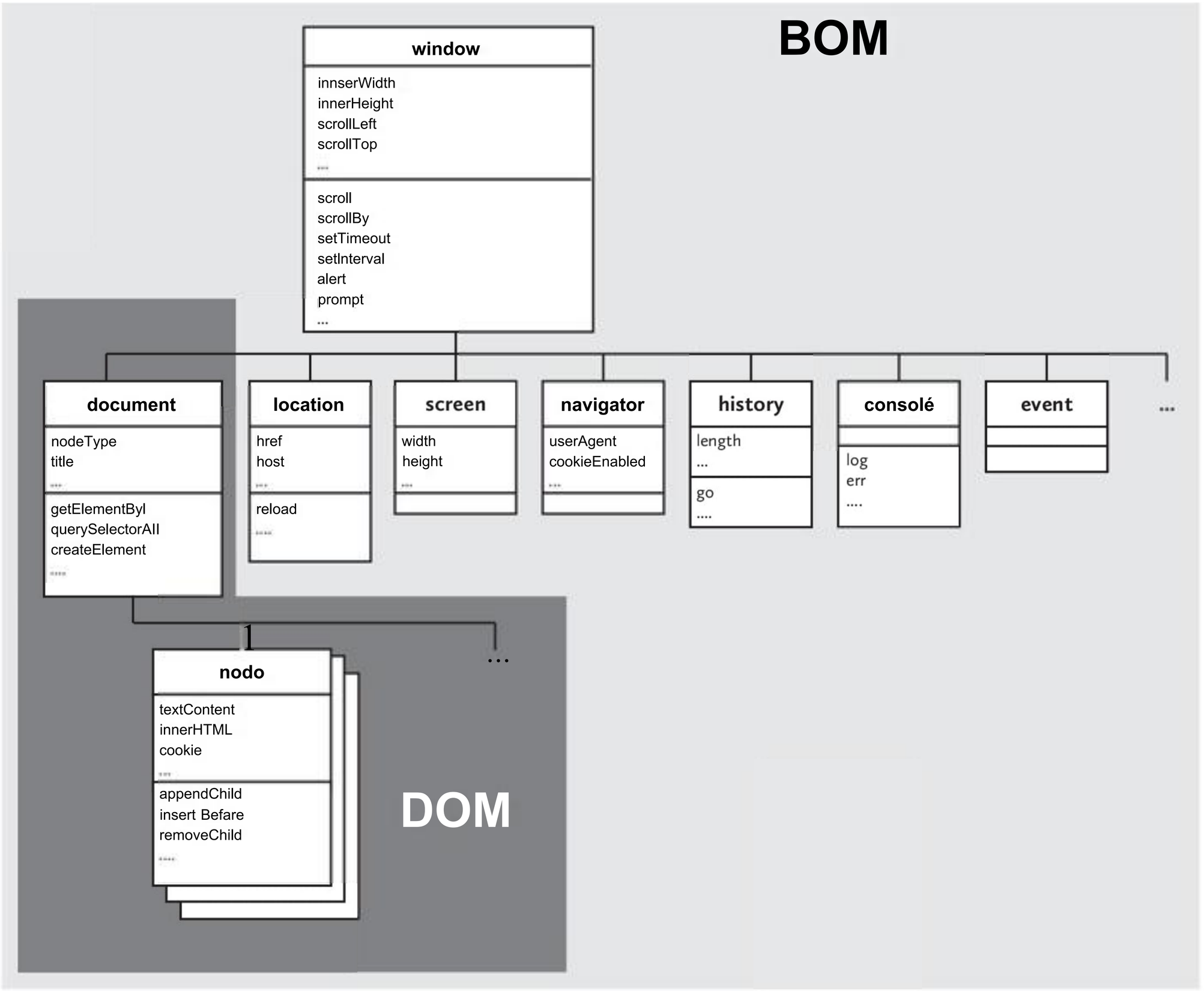


Figura 7.1: Estructura del modelo de objetos del navegador (BOM), resaltando el encaje del modelo de objetos del documento en la estructura

Hemos trabajado ya con varios métodos del objeto **window,** que resultan tan importantes que incluso se puede obviar su nombre al usar sus métodos y propiedades. Es el caso de **alert** que es un método que permite mostrar al usuario información en un cuadro de diálogo. Casi ningún desarrollador lo invoca con el nombre completo **window .alert,** usan **alert** a secas.

Esta es una prueba de lo importante que es el objeto window en la creación de aplicaciones web. El objeto window es la propia ventana y es el elemento fundamental del manejo de las aplicaciones web desde JavaScript. Es también la raíz de toda la organización de objetos a los que JavaScript puede acceder para manipular todos los aspectos de una aplicación web.

Una característica fundamental del JavaScript, de hoy en día, es que se puede ejecutar en todo tipo de entornos, no solo en los navegadores. Pero, por ejemplo, **node.js** no dispone del objeto window porque es un entorno basado en la consola del navegador. Cuando usamos este objeto es señal de que estamos escribiendo una aplicación web en el lado del cliente.

Dentro de unas pocas páginas hablaremos del **DOM,** el modelo de objetos del documento que es la parte más importante del trabajo de JavaScript en la creación de aplicaciones web.

Pues bien, realmente el DOM es parte de lo que se conoce como **BOM** abreviatura de ***Browser Objects Model*** **(Modelo de Objetos del Navegador)** y que aglutina toda la estructura organizativa de los objetos del navegador. A partir de ahora, el navegador no es una simple pantalla, sino un conjunto de objetos que podemos consultar y manipular. En él tenemos objetos como el navegador, la barra de búsqueda, la consola, el documento, etc.

Aunque el BOM no forma parte del estándar oficial del lenguaje JavaScript, lo cierto es que los distintos navegadores han igualado la forma de trabajar con estos objetos. Por lo tanto, podemos hablar de un uso estándar de los objetos del navegador.

En este apartado, vamos a conocer alguno de los objetos importantes del navegador y veremos una gran lista de métodos y propiedades de todos ellos. No se trata de aprenderse esta lista, pero sí de intuir qué ayuda pueden proporcionar estos objetos a nuestras aplicaciones. En este aprendizaje será muy importante realizar las prácticas del final de esta unidad.

7.1.2 OBJETO NAVIGATOR

Es un objeto que representa al navegador del usuario que está utilizando la aplicación web. Posee numerosas propiedades de lectura que nos permiten obtener información interesante. Quizá la más utilizada es **userAgent** que permite obtener la cadena de información del navegador mediante la que podremos saber el navegador del usuario, su versión, motor, etc. Abriendo el navegador y escribiendo este comando en la consola:

```
consolé.log(navigator.userAgent);
```

Obtendremos información que varía en cada navegador. En Google Chrome veremos:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.100 Safari/537.36
```

En Internet Explorer 11:

```
Mozilla/5.0 (Windows NT 10.0; W0W64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; Tablet PC 2.0; rv:11.0) like Gecko
```

En Firefox:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:68.0) Gecko/20100101 Firefox/68.0
```

Otras propiedades y métodos del objeto **navigator** son:

PROPIEDAD O MÉTODO	USO
clipboard	Devuelve un objeto para gestionar desde el código el portapapeles.
cookieEnabled	Valor booleano que indica si las cookies están activadas en el navegador.
geolocation	Obtiene un objeto que permite usar la API de geolocalización para acceder a la posición GPS del usuario. Véase 11.5 *"GeoLocation",* en la página 436.
javaEnabled()	Devuelve un valor booleano que indica si el plugin de Java está activado en el navegador
language	Devuelve un string con el código de lenguaje del navegador del usuario. Por ejemplo ***"es-ES".***
mimeTypes	Obtiene un array con los tipos MIME aceptados por el navegador.
onLine	Valor booleano que indica si el navegador está trabajando on line o no.
plugins	Devuelve un array con información sobre los plugins instalados en el navegador.
serviceWorker	Obtiene un objeto de tipo **ServiceWorkerContainer** capaz de trabajar con objetos de tipo **ServiceWorker.**
storage	Devuelve un objeto de tipo **StorageManager** capaz de manejar la API de almacenamiento de datos persistentes.

Algunas propiedades abren la puerta a manejar elementos avanzados en la creación de aplicaciones web, por lo que el objeto **navigator** no tiene que ser olvidado si se desea profundizar en esos aspectos.

7.1.3 OBJETO SCREEN

Se trata de otra propiedad miembro de window. En este caso, esta propiedad nos permite acceder a un objeto que representa a la pantalla. Permite obtener propiedades relacionadas con la pantalla del dispositivo en el que está navegando el usuario. Propiedades interesantes de este objeto son:

PROPIEDAD O MÉTODO	USO
availTop	Primera coordenada superior de la pantalla que se puede utilizar en nuestras aplicaciones.
availLeft	Primera coordenada izquierda de la pantalla que se puede utilizar en nuestras aplicaciones.
availHeight	Número que indica el tamaño en píxeles de la altura de pantalla del usuario.

PROPIEDAD O MÉTODO	USO
availWidth	Número que indica el tamaño en píxeles de la anchura de la pantalla del usuario.
height	Altura completa de la pantalla del usuario en píxeles.
width	Anchura completa de la pantalla del usuario en píxeles.
colorDepth	Profundidad en bits de los colores de la pantalla. El número de colores disponibles en la pantalla será 2 elevado a la cantidad devuelta por esta propiedad, por ejemplo: 2^{24}
orientation	Devuelve un objeto de tipo ScreenOrientation que sirva para saber la orientación de la pantalla. Ejemplo de uso: `consolé.log(screen.orientation.type);` En una pantalla normal de escritorio retornaría: `landscape-primary`

7.1.4 OBJETO LOCATION

Se trata de otro objeto interesante que posee el objeto window. La utilidad de location reside en el hecho de que representa la dirección URL de la propia aplicación. Esta misma propiedad es accesible mediante **document.location,** que es una referencia al mismo objeto.

La propiedad más importante del objeto es href que contiene la URL completa de la página. Ejemplo:

```
consolé.log(location.href);
```

Podría devolver algo como: **https://jorgesanchez.net/**

Más interesante aún es el hecho de que podemos ir a otra página simplemente cambiando esta propiedad:

```
location.href="https://github.com";
```

Si ejecutamos este comando desde la consola, veremos cómo nos trasladamos a la página de GitHub, independientemente de la página que estuviéramos viendo. Hay una forma más cómoda incluso de obtener el mismo resultado, que es modificar directamente la propiedad location:

```
location ="https://github.com”;
```

Este objeto tiene otra serie de propiedades (que son modificables) y que nos permiten obtener cada apartado de la URL de la página:

PROPIEDAD O MÉTODO	USO
protocol	Obtiene el nombre del protocolo que se usa para acceder a la aplicación. Ese nombre incluye los dos puntos. Ejemplo: https:

PROPIEDAD O MÉTODO	USO
host	Toma de la URL lo que se corresponde con la dirección del host. Incluye el puerto si no es un puerto estándar. **Ejemplo: jorgesanchez. net: 3241**
hostname	Idéntica al anterior, pero sin incluir el puerto.
pathname	Obtiene la ruta de directorios a partir del host en la URL. Por **ejemplo: /manuales/html/introduccion-html. html**
search	Obtiene la cadena de búsqueda de la URL. Por ejemplo: **?lang=es&registrado=si**
hash	Obtiene el marcador, si es que lo lleva, de la URL. Por ejemplo en la dirección **http://jorgesanchez.net/web#lmsgi** obtendría: **#lmsgi**
username	Obtiene el nombre de usuario que haya en la URL, si es que lo hay.
password	Si es el caso, devuelve la contraseña que haya en la URL.
origin	Es la única propiedad no modificable, es de solo lectura. Permite obtener de la URL la dirección canónica. La cual está formada solo por el protocolo, el puerto y el nombre de host.
reload([servidor])	Recarga la página. El parámetro opcional ***servidor*** es un booleano que, si vale **true,** obliga a recargar la página desde el servidor, si vale false se permite cargar la página usando la caché del navegador.

7.1.5 OBJETO HISTORY

Se trata del objeto que representa el historial de páginas visitadas por el usuario. El método más interesante de este objeto es **go,** el cual recibe un número que nos permite navegar por el historial de páginas. Así: go(-l) iría a la página anterior en el historial, go(l) iría a la página siguiente, go(0) es otra forma de recargar la página actual.

La otra propiedad interesante de este objeto es **length,** que devuelve el tamaño actual del historial.

Ejemplo:

```
history.go(-(history.length-1))
```

Este código nos permite volver a la primera página desde el historial.

7.1.6 OTRAS PROPIEDADES Y MÉTODOS DE WINDOW

El objeto window tiene otras propiedades y métodos que pueden ayudarnos a realizar algunas acciones desde JavaScript. Ya conocemos las propiedades **history, screen** y **navigator** cuyas posibilidades hemos comentado en los apartados anteriores. Hay otros objetos interesantes.

También conocemos métodos de window, concretamente: **alert, confirm** y **prompt** que se encargan de los mensajes al usuario.

A continuación, se detallan algunas de esas propiedades

PROPIEDAD O MÉTODO	USO
innerWidth	Anchura interior de la ventana. Es la anchura real disponible en el área de contenido de la ventana teniendo en cuenta su tamaño actual.
innerHeight	Altura interior de la ventana. Es la altura real disponible en el área de contenido de la ventana teniendo en cuenta su tamaño actual.
outerWidth	Anchura exterior de la ventana.
innerHeight	Altura exterior de la ventana.
screenX	Distancia de la ventana al borde izquierdo de la pantalla.
screenLeft	Idéntica a la anterior.
screenY	Distancia de la ventana al borde superior de la pantalla.
screenTop	Idéntica a la anterior.
scrollX	Índica el desplazamiento horizontal que el usuario ha realizado usando las barras de desplazamiento horizontales o arrastrando un dispositivo táctil en horizontal.
scroílY	Indica el desplazamiento vertical que el usuario ha realizado usando las barras de desplazamiento verticales o arrastrando un dispositivo táctil en vertical.
status	Referencia a la **barra de estado** de la ventana. Permite ver y modificar su contenido. La barra de estado era una franja horizontal situada en la parte inferior de la ventana en la que se daba información de estado. Hoy en día no se utiliza esta barra, pero la propiedad sigue estando disponible.
consolé	Consola del navegador. Accede a todos los métodos de la consola, incluido **console.log.**
event	Objeto del evento actualmente en ejecución. Los eventos protagonizan la unidad siguiente de este libro.

PROPIEDAD O MÉTODO	USO
fullScreen	Booleano que indica si la aplicación se muestra a pantalla completa. No funciona en **Google Chrome** en el momento de escribir este texto.
localStorage	Referencia al objeto de tipo **Storage** que permite almacenamiento de datos en modo local sin tiempo de expiración. Véase 11.4 *"Web storage",* en la página 434.
getSelectionf)	Obtiene un objeto de tipo **Selection** que nos sirve para poder obtener información sobre el texto seleccionado. Si queremos, simplemente, obtener el texto seleccionado actualmente por consola, sería: `consolé.log(getSelection().toString())`
scroll(x,y) o bien **scroll(objetoScroll)**	Sirve para desplazar la página a una posición concreta. Permite indicar el desplazamiento mediante dos coordenadas (x horizontal *ey* en vertical). Ejemplo: `scrol1(0,100) H` Desplaza la página haciendo que la barra vertical se desplace a la posición marcada por 100 píxeles. En horizontal se desplaza totalmente a la izquierda. Se puede pasar, en lugar de las dos coordenadas, un objeto de tipo **ScrollToOptions** (algunos navegadores como Edge o Explorer no aceptan este parámetro todavía). Los objetos **ScrollToOptions** tienen estas tres propiedades: ■ **top.** Desplazamiento vertical. ■ **left:** Desplazamiento horizontal. ■ **behavior:** Comportamiento. Por ahora admite como valores: **auto** (se desplaza sin animación) y **smooth** (desplazamiento con animación suave). Ejemplo: `scrol1({ top:1000, left:0, behavior:"smooth" });` El documento se desplaza automáticamente a la posición 1000 píxeles de forma progresiva con una animación.

PROPIEDAD O MÉTODO	USO
scrollBy(x,y) o bien **scrollBy(objetoScroll)**	Los parámetros funcionan igual que con la propiedad anterior, pero ahora el desplazamiento es relativo respecto a la posición actual. Ejemplo: `scrollBy({ top:innerHeight, left:0, behavior:"smooth" |});` Este código hace que la página se desplace suavemente una pantalla completa hacia abajo.
scrollTo(x,y) **o** bien **scrollTo(objetoScrol)**	Los parámetros funcionan igual que con la propiedad **scroll.** La diferencia es que los valores indicados son coordenadas y no píxeles de desplazamiento (normalmente ambas cosas coinciden).
stop()	Detiene la carga de la página.
find(texto)	Busca el texto en el documento actual y devuelve true o false si lo encuentra. Además, la mayoría de navegadores resaltan la palabra dentro de la página a la vez que realizan un scroll hacia ella.
getComputedStyle(elemento)	La detallaremos más adelante dada su importancia (véase 7.4.3.2 *"Obtener los estilos CSS que se aplican a un elemento",* en la página 268). Permite obtener la información de las propiedades CSS en uso por parte de un elemento.
open(URL,nombreVentana, ajustes)	Abre una ventana y muestra el contenido de la URL indicada. Opcionalmente, se puede indicar un nombre para la ventana y los ajustes de la misma (tamaño, barras de desplazamiento, etc.).

7.2 ¿QUÉESELDOM?

7.2.1 EL MODELO DE OBJETOS DEL DOCUMENTO

JavaScript se ideó para que fuera un lenguaje con capacidad de manipular el contenido y apariencia de una página web de forma dinámica. Para hacer esto posible, los elementos de la página web se tienen que poder manipular como una serie de objetos relacionados. Así apareció el término **DOM, *Document Objects Model*** (Modelo de Objetos del Documento) que define una página web como una estructura organizada de objetos que forman un árbol de recorrido.

Si, por ejemplo, tenemos este código HTML:

```
<!DOCTYPE html>
<html lang=”es">
```

```
<head>
        <meta charset="UTF-8">
        <title>Documento HTML</title>
</head>
<body>
    <section id="sl">
            <hl>Título</hl>
            <p>Ejemplo de documento <strong>DOM</strong>Vísión en árbol</p>
    </section>
    <section id="s2">
            <p>Aquí sólo hay texto</p>
    </section>
</body>
</html>
```

Podemos entender que forman este tipo de estructura:

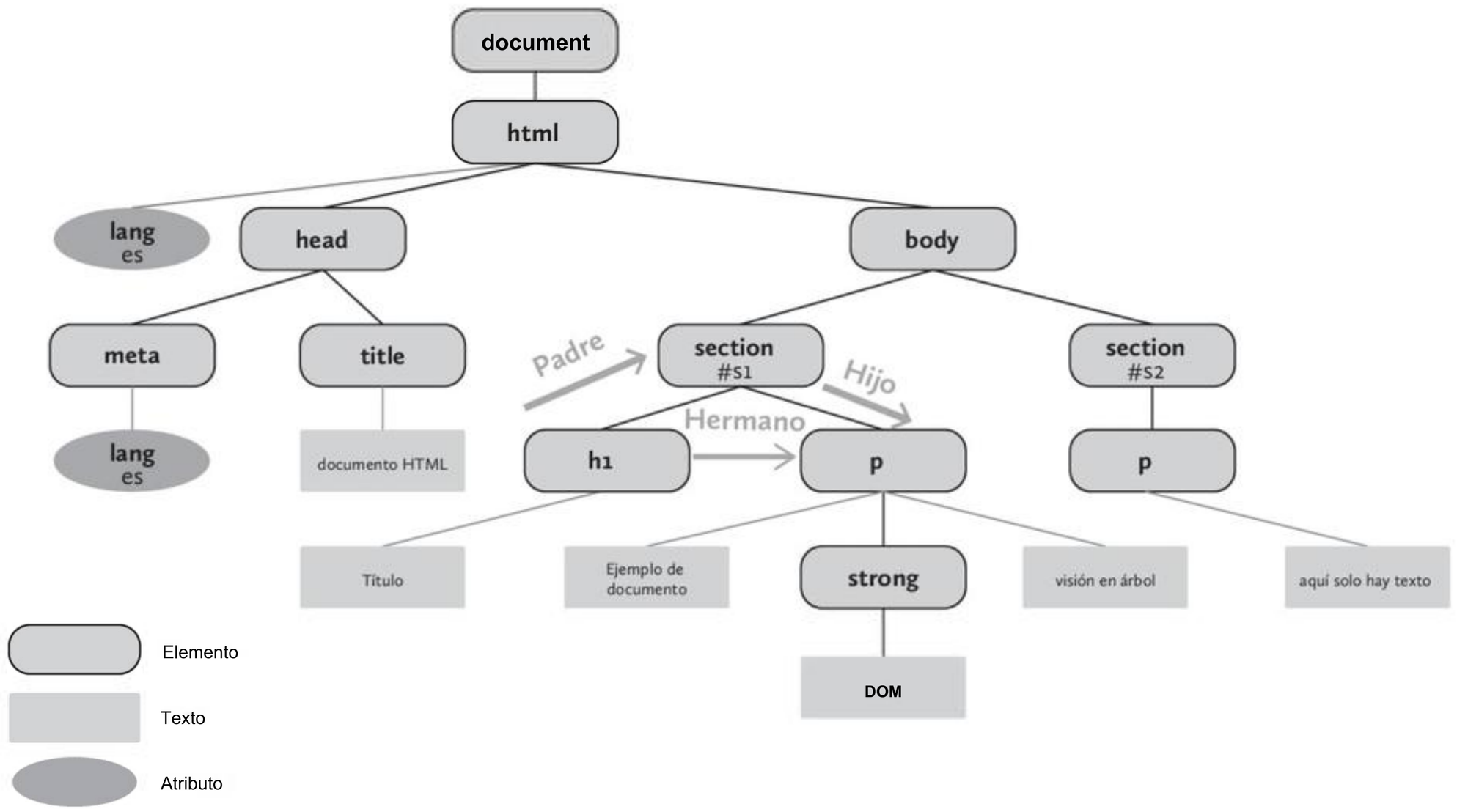


Figura 7.2: Ejemplo de estructura de modelo de objetos del documento (DOM)

Realmente la estructura del DOM no se creó solo para JavaScript, la idea inicial es que fuera manipulable también mediante otros lenguajes. Claramente, JavaScript es el lenguaje que más se utiliza para utilizar el DOM, pero lo cierto es que es una estructura de objetos independiente de todo lenguaje. Eso hace que haya ciertas particularidades en el manejo de los objetos del documento que son extrañas desde el punto de vista de JavaScript.

Otro detalle importante es que esta estructura manipulativa, no solo se ideó para documentos HTML, sino que funciona de la misma manera para manipular documentos XML.

En todo caso, la idea es la de la imagen anterior: una estructura de objetos en forma de árbol donde los nodos representan objetos del documento y cada nodo posee un nodo **padre** a través del cual nos acercamos a la raíz del árbol.

7.2.2 EL OBJETO DOCUMENT

La forma de poder acceder a los objetos del documento en javaScript es a través de un objeto llamado precisamente **document.** Este objeto en realidad es una propiedad del objeto **window.** Podemos indicar simplemente la palabra **document** para acceder a este objeto, aunque su nombre completo es **window.document.**

Los diferentes métodos y propiedades del objeto document nos van a permitir acceder a los elementos del documento para su manipulación.

7.2.3 TIPOS DE NODOS

Más adelante veremos muchos métodos para acceder a los elementos de un documento, pero el más habitual es el método **getElementByld** del objeto **document.** Así si usamos el código anterior podemos acceder a la primera sección a través del código:

```
let seccion=document.getElementById("#sl");
```

Este código hace que la variable ***sección*** sea una referencia al elemento identificado como ***si.*** Es decir, lista es un nodo del documento. Todos los nodos poseen una propiedad llamada **nodeType.** Si la mostramos en este caso:

```
consolé.log(sección.nodeType);
```

Se muestra el valor 1. Este valor significa que el node es un elemento. Si ejecutamos este otro código:

```
consolé.log(document.nodeType);
```

Escribirá el valor 9.

La lista de posibles valores de **nodeType** es la siguiente:

VALOR	TIPO DE NODO	CONSTANTE RELACIONADA
1	Elemento	**Node. ELEMENT.NODE**
2	Atributo	**Node.ATTRIBUTE_NODE**
3	Texto	**Node.TEXT_NODE**
4	Apartado CDATA	**Node.CDATA_SECTION_NODE**
5	Referencia a entidad	**Node.ENTITY_REFERENCE_NODE**
6	Entidad	**Node.ENTITY_NODE**
7	Instrucción de procesado	**Node.PROCESSING_INSTRUCTION_NODE**
8	Comentario	**Node.COMENT_NODE**

VALOR	TIPO DE NODO	CONSTANTE RELACIONADA
9	Documento completo	**Node.DOCUMENT_NODE**
10	Nodo de tipo de documento	**Node.DOCUMENT_TYPE_NODE**
11	Nodo de fragmento de código	**Node.DOCUMENT_FRAGMENT_NODE**
12	Nodo de anotación	**Node.NOTATION_NODE**

Algunas entradas están muy relacionadas con documentos **XML** por lo que no nos serán útiles para manipular documentos HTML. Normalmente, nos bastará con las tres primeras entradas de esta tabla. Las constantes definidas en el objeto Node, nos permiten no tener que aprender los números y manejar en su lugar las constantes que son más entendibles.

Hay que hacer notar que la propiedad **nodeType** es de lectura. Como también lo es la propiedad **nodeName.** Esta última propiedad nos permite obtener el nombre del nodo

```
consolé.log(sl.nodeName);
```

El código anterior escribe **SECT1ON** (en mayúsculas), porque la variable lista hace referencia a un elemento **section** de HTML. Sin embargo:

```
consolé.log(document.nodeName);
```

Escribe **#document.** Los valores de nodeName cambian en función del tipo de nodo que tengamos:

La lista de posibles valores de **nodeName** es la siguiente:

TIPO DE NODO	VALOR DEVUELTO POR NODENAME
Elemento	Nombre de la etiqueta del elemento
Atributo	Nombre del atributo
Texto	#text
Comentario	#comment
Documento	#document

Finalmente, la propiedad **nodeValue** es capaz de devolver el valor del nodo, aunque en el caso del documento completo nodeValue devuelve **nuil.**

Estas tres propiedades **(nodeType, nodeName** y **nodeValue)** facilitan obtener la información más importante sobre un nodo a la hora de manipularlo.

7.3 SELECCIÓN DE ELEMENTOS DEL DOM

EL objeto **document** proporciona numerosos métodos para seleccionar elementos. Las últimas revisiones del DOM han añadido nuevos métodos que facilitan mucho la labor del desarrollador.

7.3.1 SELECCIÓN POR EL IDENTIFICADOR

Se trata de seleccionar un nodo del documento por el valor del atributo **id.** El método que lo permite es **getElementByld** que recibe un string con el valor buscado. Retorna como resultado el elemento en sí o bien el valor **nuil** si ese identificador no se encuentra en el elemento.

```
let sl=document.getElementBy!d("si");
```

7.3.2 POR ETIQUETA

La selección por etiqueta permite seleccionar todos los elementos que tengan ese nombre de etiqueta. El método que lo consigue es **getElementsByTagName.** En este caso se devuelve una lista de elementos de tipo **NodeList.** Es decir, no se selecciona un elemento (como ocurría con **getElementByld),** sino todos los elementos que usen esa etiqueta.

Usando la misma página web podemos escribir este código JavaScript en la consola del navegador:

```
let elementosP=document.getElementsByTagName("p");
```

El cual permite que **elementosP** sea una referencia a todos los elementos de tipo **p.** Por eso este código:

```
consolé.log(elementosP.length);
```

Escribe el número 2 por consola, ya que hay 2 elementos de tipo p. Mientras que este otro código:

```
consolé.log(elementosP[0]);
```

Mostraría este resultado, que se corresponde con el contenido del primer elemento de tipo p.

```
<p>Ejemplo de documento <strong>DOM</strong>Visión en árbol</p>
```

El método **getElementsByTagTagName** lo tienen todos los elementos. Si en lugar de utilizar **document,** usamos un elemento concreto, entonces se buscan los elementos con esa etiqueta dentro del que digamos. Ejemplo:

```
<main id="principal">
            <p>Uno</p>
            <p>Dos</p>
            <p>Tres</p>
</main>
<p>Cuatro</p>
<p>Cinco</p>
<script>
                        let principal=document.getElementBy!d("principal");
    let interiores=principal.getElementsByTagName("p");
    consolé.log(interiores.length)
</script>
```

El resultado de el código anterior es el número 3. La capa principal tiene tres párrafos dentro, los otros dos están fuera. La expresión document. getElementsByTagName("p") hubiera devuelto 5.

7.3.3 OBJETOS NODELIST

El selector por etiqueta nos ha permitido conocer a los objetos NodeList que representan una colección de nodos del DOM, es decir, una serie de elementos del documento. Es fundamental este objeto porque son muchas las propiedades y métodos de document que devuelven este tipo de objetos.

Aunque los objetos NodeList no son realmente arrays, se manejan en gran medida como si lo fueran. Se pueden usar todos los bucles de recorrido vistos con los arrays, se puede acceder a cada elemento indicando su índice entre corchetes y dispone de propiedades comunes a los arrays como **length** o **forEach.** Sin embargo, otros métodos de los arrays como **slice, sort** o **join,** no están disponibles.

No obstante, si deseamos disponer de esos métodos y de otros, tendremos que convertir este tipo de objetos en un array, lo cual es posible mediante el operador de propagación:

```
let arrayElementosP=[...elementoP];
```

O, también:

```
let arrayElementosP=Array.from(elementosP);
```

Pero, lo cierto es que no suele ser necesaria esta conversión.

7.3.4 POR CLASE

Se trata de seleccionar todos los elementos que tienen asignada una clase determinada de CSS. Por ejemplo:

```
<ul>
      <li class="verdura">Cebolla</li>
      <1 i c1ass="verdura">Ajo</1 i >
      <li class="carne">Pollo</li>
      <li class="verdura">Lechuga</li>
      <1 i class="carne">Ternera</li>
      <li class="hidrato">Pan</I i>
      <1 i class="hidrato">Arroz</li>
      <1 i class="carne">Cerdo</li>
      <1 i class="verdura">Tomate</li>
      <1 i class="hidrato">Lentejas</li>
</ul>
```

En esta lista de alimentos cada elemento se relaciona con una clase concreta, eso permite aplicar un CSS distinto a cada clase de alimento. Si deseamos operar desde JavaScript con los

elementos de una clase concreta, disponemos del método: **getElementsByClassName ,** el cual nos devuelve un objeto de tipo **NodeList** con todos los elementos que sean de la clase indicada,

```
let verduras=document.getElementsByClassName("verdura");
```

7.3.5 POR SELECTOR CSS

Indudablemente el selector más poderoso es **querySelectorAll.** Permite indicar, como texto, cualquier selector válido CSS. Si la sintaxis del selector no es correcta, entonces devolverá un error. Lo que devuelve, nuevamente, es una lista de elementos con aquellos que cumplan el selector.

Por ejemplo, para seleccionar los elementos de clase verdura, podemos:

```
let verduras=document.querySelectorAll(".verdura");
```

Se usa el punto, porque ahora hay que hacerlo en el formato CSS. Pero las posibilidades son muy espectaculares gracias a la potencia de los selectores de CSS:

```
//Selecciona los elementos li que están dentro de ul
document.querySelectorAl1("ul 1 i");
//Selecciona el primer elemento li de cada lista
document.querySelectorAl1("1 i:first-of-type");
//Selecciona el primer elemento li de cada lista si es de clase verdura
document.querySelectorAl1("1 i.verdura:first-of-type");
```

Hay un método llamado **querySelector** a secas (sin la palabra ***all)*** que funciona igual, pero no devuelve una lista, sino el primer elemento en el documento que cumpla el criterio. No suele ser muy habitual su uso.

Hayquerecordarqueaunqueuncódigocomodocument. querySelectorAl 1 ("#capal7"); está claro que solo devuelve un elemento, este método siempre devuelve una colección de elementos. Es decir, acceder al elemento en cuestión implica utilizar el índice cero del resultado.

7.4 OBTENER Y MODIFICAR DOM

7.4.1 MANIPULACIÓN DE ATRIBUTOS

7.4.1.1 OBTENER EL VALOR DE UN ATRIBUTO

Los elementos del DOM disponen del método **getAttribute** para obtener el valor de un atributo concreto. Por ejemplo:

```
let 1is=document.querySelectorAl1("1 i");
for(let li of lis){
        consolé.log(1 i.getAttribute("class"));
}
```

Este código recorre cada elemento de tipo li y nos muestra por consola la clase CSS de cada elemento.

Si un elemento no tiene el atributo indicado, **getAttribute** devuelve **nuil.**

7.4.1.2 MODIFICAR EL VALOR DE UN ATRIBUTO

El método **setAttribute** es el que permite esta operación. El primer parámetro será el nombre del elemento y el segundo el nuevo valor.

```
elemento.setAttribute("class","verdura");
```

7.4.1.3 ELIMINAR UN ATRIBUTO

El método **removeAttribute** seguido de un nombre de atributo, nos permite eliminar el atributo que tenga ese nombre del elemento.

7.4.1.4 AÑADIR Y QUITAR ATRIBUTO DE FORMA RÁPIDA

El método **toggleAttribute** permite añadir al elemento el atributo que se indique si no está añadido ya, o quitar el atributo con ese nombre si ya está añadido. Si la primera vez que usamos el método ***toggleAtribute,*** añade el atributo, la siguiente lo eliminará, la siguiente lo volverá a añadir y así sucesivamente.

7.4.1.5 SABER SI UN ELEMENTO TIENE ATRIBUTOS

El método **hasAttribute** devuelve verdadero si el elemento tiene el atributo cuyo nombre se indica y falso si no los tiene.

7.4.1.6 OBTENER TODOS LOS ATRIBUTOS DE UN ELEMENTO

La propiedad **atributes** de un elemento permite obtener todos los atributos de un elemento. Lo que devuelve es un objeto de tipo **NamedNodeList** que no es una estructura de array por lo que no tenemos acceso a la mayoría de propiedades del array. Pero sí podemos acceder por número de índice y usar la propiedad length.

Cada elemento de la lista que devuelve esta propiedad posee una propiedad llamada **ñame** que contiene el nombre del atributo y **valué** que contiene su valor. Por ejemplo, si tenemos este código HTML:

```
<form>
        <label for="edad">Escriba su edad></label>
        <input type="number" id="edad" name="edad" min="18" max="65">
</form>
```

Para navegar por los atributos del cuadro numérico y mostrarlos por consola podríamos escribir este código:

```
let 1istaAtributos=document.getElementById("edad").attributes;
for(let atributo of 1istaAtributos){
        consolé.log('Atributo: ${atributo.ñame} '+
```

```
                    "Valor: ${atributo.valué}');
```

El resultado es:

```
Atributo:      type      Valor:      number
Atributo: id Valor: edad
Atributo:      ríame     Valor:      edad
Atributo: min Valor: 18
Atributo: max Valor: 65
```

7.4.2 MANIPULACIÓN DEL CONTENIDO DE LOS ELEMENTOS

7.4.2.1 PROPIEDAD TEXTCONTENT

La propiedad **textContent** de un elemento permite obtener y modificar el texto que contiene ese elemento. Hay que tener en cuenta que un elemento puede tener dentro más elementos y que esta propiedad obtendrá el texto de todos ellos. Por ejemplo, si observamos este código:

```
<main>
   <hl>Sistema Solarc/hl>
   <h2>Sol</h2>
   <p id="pSol">
      El Sol, una estrella de tipo espectral <strong>G2</strong> que
      contiene más del 99,85 % de la masa del sistema. Con un diámetro
      de 1 400 000 km, se compone de un 75 % de hidrógeno, un 20 % de
      helio y 5 % de oxígeno, carbono, hierro y otros elementos.
   </p>
   <h2>Planetas</h2>
   <p id="pPlanetas">
      Los planetas, divididos en planetas interiores (también llamados
      terrestres o telúricos) y planetas exteriores o gigantes. Entre
      estos últimos <strong>Júpiter</strong> y <strong>Saturno</strong>
      se denominan gigantes gaseosos, mientras que Urano y Neptuno
      suelen nombrarse gigantes helados. Todos los planetas gigantes
      tienen a su alrededor anillos.
   </p>
</main>
```

Al ejecutar esta instrucción:

```
consolé.log(document.getElementById("pSol          ").textContent);
```

Escribirá:

```
El Sol, una estrella de tipo espectral G2 que contiene más del 99,85 % de
la masa del sistema. Con un diámetro de 1 400 000 km, se compone de un
```

```
75 % de hidrógeno, un 20 % de helio y 5 % de oxígeno, carbono, hierro y
otros elementos.
```

Que es el texto del párrafo indicado. Sin embargo, dentro de ese elemento hay otro de tipo **strong** y su texto aparece también en el resultado a la vez que la propia etiqueta strong desaparece del resultado. Es decir, solo recoge el texto interior (todo el texto) y no las etiquetas.

A través de esta propiedad podemos cambiar el contenido:

```
document.getElementById("pSol").textContent=
                                        "Soy el <strong>Sol</strong>";
```

Si observamos el resultado en la página (ese código le podemos escribir en la consola del navegador), veremos que aunque hemos indicado que la palabra Sol se encuentre dentro de etiquetas **strong,** estas no funcionan, se toman de forma literal. La razón es que **textContent** tiene capacidad para modificar el texto de un elemento, pero no los elementos interiores.

7.4.2.2 PROPIEDAD INNERHTML

Se trata de un propiedad similar a la anterior, pero esta sí lee y manipula las etiquetas HTML. Por ejemplo:

```
consolé.log(document.getElementById("pSol").innerHTML);
```

Este código muestra:

```
El Sol, una estrella de tipo espectral <strong>G2</strong> que contiene
más del 99,85 % de la masa del sistema. Con un diámetro de 1 400 000 km,
se compone de un 75 % de hidrógeno, un 20 % de helio y 5 % de oxígeno,
carbono, hierro y otros elementos.
```

Parece el mismo resultado que el obtuvimos con **textContent.** Pero la palabra G2 está contenida en una etiqueta **strong** que con textContent no podíamos ver. La diferencia es que innerHTML sí lee las etiquetas.

Y no solo las lee, puede modificar el contenido usando etiquetas HTML:

```
document.getElementById("pSol").innerHTML^
                                        "Soy el <strong>Sol</strong>";
```

Si vemos el resultado, ahora sí veremos que entiende las etiquetas y realmente vemos en negrita la palabra **Sol.**

7.4.3 MODIFICAR CSS

Es posible modificar el CSS de los elementos a través de los atributos **style** o **class** mediante el método **setAttribute** visto anteriormente. Pero al ser un elemento tan importante, JavaScript proporciona métodos especiales para manipular las clases CSS de un elemento.

7.4.3.1 PROPIEDAD STYLE

Los elementos poseen una propiedad llamada style que permite acceder a las propiedades CSS de un elemento. Lo que realmente hace es modificar el atributo style del elemento, modifican los valores de la misma. La idea es que **style** es un objeto que tiene como propiedades todas las propiedades CSS en formato **Camel Case** (mayúsculas estilo camello). Veámoslo con ejemplos,

```
let párrafo=document.getElementsByTagName("p")[0];
```

El código anterior selecciona el primer párrafo (de tipo p) del documento, sea cual sea.

```
párrafo.style.color="red";
```

Esta línea colorea de color rojo el texto de ese párrafo.

Podemos de esa forma modificar todo el CSS que queramos, por ejemplo:

```
párrafo.style.border="lpx solid black";
```

Y así con todas las propiedades. El problema viene si hubiéramos querido modificar solo el borde inferior. La propiedad CSS para hacer esa modificación se llama **border-bottom,** esun nombre problemático para la sintaxis anterior, porque hay un guión en él. En JavaScript los identificadores de propiedades, funciones, métodos, etc, no pueden tener guiones (se confunde con la resta. Lo mismo ocurre la propiedad que modifica el color de fondo: **background-color.**

Hay que tener en cuenta que el formato de modificación CSS explicado anteriormente, es antiguo y entonces no se disponía de la posibilidad de acceder a una propiedad mediante corchetes. Por eso la propiedad style contiene una propiedad por cada propiedad CSS, pero usando el llamado formato **Camel Case.**

En el formato Camel Case, **background-color** se convierte en **backgroundColor, border-bottom** en **borderBottom, text-align** en **textAlign** y así con todas las propiedades. Por lo que modificar el color de fondo se hace así:

```
párrafo.style.backgroundColor="ttCCC";
```

No obstante, la mayoría de navegadores actuales acepta también el formato idéntico a CSS a través de corchetes:

```
párrafo.style["background-color"]="#FCC"; //Colorea de rojo claro
```

No obstante, hay varios problemas a tener en cuenta a la hora de usar style para modificar el CSS:

- Manipular la propiedad de los elementos style da prioridad a cualquier otro CSS (ya que se modifica la propiedad **style** de HTML), pero no siempre es lo deseable.
- Es más fácil de mantener el código si se usan clases para aplicar CSS. Especialmente, cuando queremos modificar varias propiedades CSS a la vez.
- CSS avanza muy rápido por lo que muchas propiedades CSS ya disponibles, en algunos navegadores no lo están a través de la propiedad style de JavaScript. Aunque la mayoría de navegadores se ponen al día rápido.

- No podemos consultar a través de esta propiedad las propiedades CSS que tiene un elemento al cual se le ha dado formato a través de hojas de estilo externas.

7.43.2 OBTENER LOS ESTILOS CSS QUE SE APLICAN A UN ELEMENTO

Precisamente para paliar el último problema comentado en el apartado anterior, disponemos de un método en el objeto window llamado **getComputedStyle.** Este método devuelve un objeto (idéntico al de la propiedad style vista en el apartado anterior) en el que se pueden consultar las propiedades CSS que se están aplicando a un elemento concreto. Este método solo permite ver las propiedades del elemento, pero no modificarlas.

Como hemos comentado, se trata de un método del objeto window y no de los elementos en sí. Por ello, el método requiere que se le pase el elemento a evaluar:

```
let cssPárrafo=window.getCompLitedStyle(párrafo);
consolé.log(cssPárrafo.fontFamily);
```

Este código mostrará por pantalla el valor de la propiedad CSS **font-family** que tiene el párrafo. Se mostrará el valor, independientemente de si esa propiedad se asignó usando la propiedad style o mediante una hoja de estilos externa o mediante JavaScript.

7.4.33 MANIPULAR LAS CLASES CSS

Los elementos disponen de una propiedad llamada **className.** Mediante esa propiedad podemos asignar una clase CSS a un elemento de la página. Por ejemplo:

```
párrafo.className="remarcado";
```

También podemos simplemente obtener la clase del elemento. Por ejemplo:

```
consolé.log(párrafo.className); //Escribe remarcado
```

Los problemas surgen cuando más de una clase asignada al elemento. Sí podemos ver las clases asignadas. Por ejemplo:

```
<p id="nota20" class="remarcado anotación">
A tener en cuenta que el 6 de abril se cierra por reforma</p>
```

Si ahora ejecutamos este código en la consola:

```
consolé.log(document.getElementById("nota20").className);
```

Obtendremos este resultado:

```
remarcado anotación
```

Vemos las dos clases asignadas al párrafo. El problema es que acciones como añadir una clase más a un elemento o quitar una de las clases, se pueden hacer con de esta forma, pero es muy incómodo manejar la propiedad class como un string.

Por esa razón JavaScript incorporó la propiedad **classList** a los elementos del DOM. Se trata de un objeto que representa las clases aplicadas a un elemento. Nuevamente, tenemos un objeto que parece un array y nuevamente, se parece en cuanto a que podemos acceder a sus elementos por el índice, a que disponemos de la propiedad length para saber, en este caso, cuántas clases tiene asignado un objeto y en que podemos usar los bucles de recorrido habituales para recorrer las diferentes clases de un elemento. Pero se diferencia en que la mayoría de métodos de lops arrays no están disponibles.

Así, obtener el nombre las clases de un elemento mediante **classList** se haría, por ejemplo, de esta forma:

```
for(let clase of párrafo.classList){
      consolé.log(clase);
}
```

Escribiría, en este caso:

```
remarcado
anotación
```

La propiedad **classList** dispone de estas interesantes propiedades:

- **add(nombreClasel [, nombreClase2]):** Permite añadir una clase al elemento. Basta con indicar el nombre de la clase entre comillas. Admite indicar varias clases si se separan con comas.

- **remo ve (nombreCl asel [, nombreClase2¡):** Método contrario al anterior. Retira del elemento la clase o clases que se indiquen.

- **toggle(nombreClase [, forzar]):** Si se usa un solo parámetro, será el nombre de una clase. El método hace que si el elemento no tiene asignada esa clase, se asigna. Si el elemento ya tenía esa clase, la quita. Es un método muy interesante y que se usa muy a menudo.

  El segundo parámetro ***(forzar)*** es un valor booleano que si vale **true** entonces añade la clase, independientemente de si el elemento ya la tenía o no. Si vale false, entonces quita la clase del elemento. Es decir, este método es capaz de hacer lo que hacían los otros dos.

- **contains(nombreClase):** Devuelve **true** si el elemento tiene asignada la clase cuyo nombre se indica en el parámetro.

- **replace(nombreViejo, nombreNuevo):** Permite cambiar una clase por otra en el elemento.

En la práctica, dar formato CSS usando clases, en lugar de manipular propiedades CSS una a una, es más cómodo. Y **classList** aporta todavía más comodidad gracias a las propiedades que ofrece.

7.4.4 OBTENER ATRIBUTOS DATA

En los documentos HTML existe la posibilidad de crear atributos **data.** Se trata de un tipo de atributos creados por los propios desarrolladores a voluntad. Se usan para almacenar información que puede ser manipulada desde JavaScript.

Los atributos data empiezan por ese mismo término (data) seguidos de un guión y luego el nombre en sí que queramos data al atributo. Por ejemplo:

```
<p id="pl" data-tipo="Inicio de libro" data-1ibro="Don Quijote"
    data-autor-principal="Miguel de Cervantes">
En un lugar de La Mancha...
</p>
```

En este ejemplo se han diseñado dos atributos de forma personal. No es muy lógico grabar tanta información, pero nos interesa como ejemplo formativo. Ahora la cuestión es cómo se manipulan estos atributos desde JavaScript.

La propiedad **dataset** es la encargada de conseguirlo. El manejo es muy similar a lo que vimos respecto a la propiedad **style.**

dataset es un objeto que contiene como propiedades cada una de los atributos data del elemento actual. Para acceder concretamente a uno se usa una sintaxis de tipo **Camel Case** (de la que ya hablamos en el apartado de la propiedad style). Concretamente en este ejemplo, podemos ver los valores de los tres atributos de esta forma:

```
consolé.log(pl.dataset.libro);
consolé.log(pl.dataset.autorPrincipal);
consolé.log(pl.dataset.tipo);
```

Muestra:

```
Don Quijote
Miguel De Cervantes
Inicio de libro
```

Es posible también modificar un atributo data, o crearlo si no existe:

```
pl.dataset.publicacion="Siglo XVII";
```

7.4.5 NAVEGAR POR EL DOM

Los métodos que se explican en este apartado nos permiten obtener información de la estructura del DOM del documento. Son imprescindibles para manipular dicha estructura y par.

7.4.5.1 OBTENER LOS HIJOS DE UN ELEMENTO

Veamos este código:

```
<section id="sl">
<hl id="titulo">¿Quién soy?</hl>
```

```
<p id="parrafol">Hola, soy <strong>Jorge Sánchez</strong>, soy profesor
de informática< /P>
<p id="parrafo2">Y también soy un cinéfilo</p>
<p id="parrafo3">Y un seriéfilo</p>
</section>
```

Todos los elementos disponen de una propiedad llamada **childNodes,** la cual devuelve una lista de nodos. Como ejemplo, utilizaremos el párrafo identificado como párrafol, para obtener su lista de nodos hijo a través de la consola de JavaScript:

```
let parrafoí=document.getElementById("parrafol");
for(let hijo of parrafol.childNodes){
      consolé.log('Texto: "${hijo.nodeValue}"\n' +
                  `Tipo de nodo: ${hijo.nodeType}, ${hijo.nodeName}');
}
```

El resultado es:

```
Texto: "Hola, soy "
  Tipo de nodo: 3, #text
Texto: "nuil"
  Tipo de nodo: 1, STRONG
Texto: ", soy profesor de informática                "
  Tipo de nodo: 3, #text
```

Un detalle que suele confundir es que el texto que está dentro de un elemento es también un nodo del DOM. Si mostramos los tipos, al texto se le otorga el número 3 y a los elementos el número 1.

Hay una propiedad llamada **children** que funciona igual que childNodes, pero solo obtiene los nodos hijos que sean elementos:

```
for( let hijo of parrafol.children){
      consolé.log('Texto: "${hijo.nodeValue}"\n'+
                  `Tipo de nodo: ${hijo.nodeType}, ${hijo.nodeName}');
}
```

Ahora el resultado es:

```
Texto: "nuil"
  Tipo de nodo: 1, STRONG
```

Solo hay un nodo hijo del primer párrafo que sea un elemento.

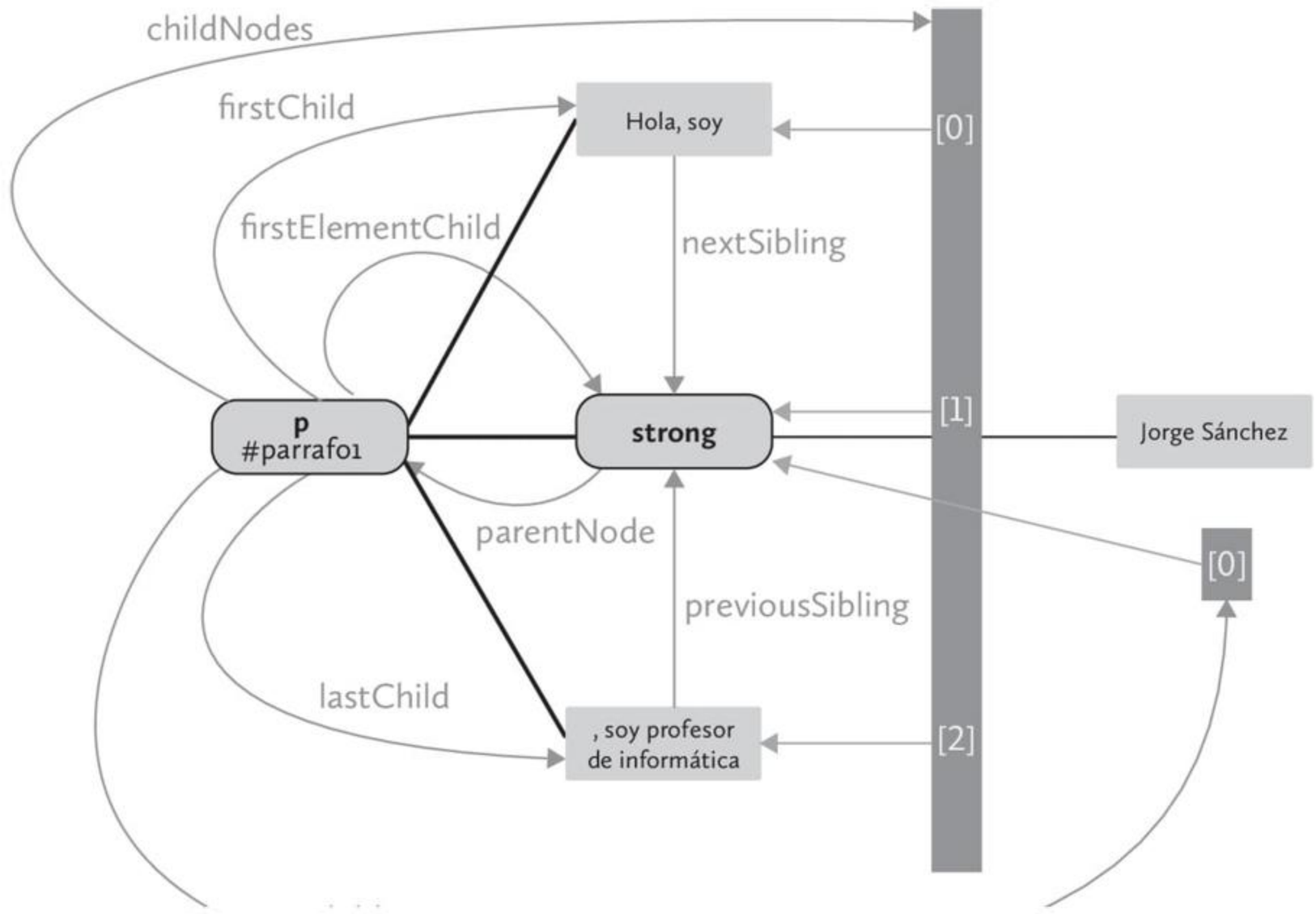


Figura 7.3: Retazo de objetos del DOM y uso de los métodos de recorrido

7.4.5.2 PROPIEDADES QUE FACILITAN LA NAVEGACIÓN POR EL DOM

La Figura 7.3 mostrada anteriormente nos da una idea de cómo podemos recorrer el DOM. Aparte del método que obtiene todos los hijos, tenemos las siguientes propiedades que permiten referenciar nodos relacionados con el elemento actual:

PROPIEDAD	USO
firstChild	Selecciona el primer nodo hijo del elemento actual.
lastChild	Selecciona el último nodo hijo del elemento actual.
nextSibling	Selecciona el siguiente hermano respecto del nodo actual. Son hermanos los nodos que están al mismo nivel en el árbol DOM. Dicho de otra forma, son hermanos los nodos que tienen el mismo padre.
previousSibling	Obtiene el hermano anterior al nodo actual.
parentNode	Obtiene el padre del nodo actual.
firstElementChild	Obtiene el primer hijo del nodo actual que sea un elemento.
lastElementChild	Obtiene el último hijo del nodo actual que sea un elemento.
nextElementSibling	Obtiene el siguiente hermano del anterior que sea de tipo elemento
previouslementSibling	Obtiene el elemento hermano anterior al nodo actual
childElementCount	Número de elementos hijos del nodo actual

Como se puede apreciar en la tabla anterior, todas las propiedades tienen dos versiones: una sin la palabra ***Element*** y otra con ella. Suelen ser más interesantes las que tienen la palabra Element, ya que solo realizan recorridos a través de nodos que sean elementos de la página **(p, strong, hl, div,** etc.)

7.4.6 AÑADIR ELEMENTOS

7.4.6.1 CREAR ELEMENTOS

El método **createElement** del objeto document es el que permite crear elementos. Requiere que indiquemos el nombre de la etiqueta de dicho elemento:

```
let capal=document.createElement("div");
```

La variable ***capal*** es la referencia al nuevo elemento. Este elemento no se mostrará todavía en el documento, porque realmente no forma parte de los nodos visibles al no haber indicado su posición en el árbol de nodos que cuelgan de **document.**

Sí tiene disponibles las propiedades habituales de los elementos.

```
capal.innerHTML="<p>Esta es mi nueva capa</p>";
capal.setAttribute("id","capal");
```

7.4.6.2 CREAR NODOS DE TEXTO

El método **createTextNode** crea un nodo de tipo texto. Para ello basta con indicar el texto que deseamos colocar en el nodo:

```
let nodoTl=document.createTextNode("Este texto no se muestra todavía");
```

Al igual que ocurría con createElement, los nodos recién creados no se muestran hasta que no los coloquemos dentro de document.

7.4.63 AÑADIR UN NODO HIJO

El método **appendChild** permite colocar un nodo como hijo de otro elemento. Es un método que poseen todos los nodos del DOM y que tiene como único parámetro el nodo que deseamos colocar como hijo.

Ejemplo:

```
<main id="principal">
        <p>Soy el primer párrafo</p>
        <p>Soy el segundo párrafo</p>
        <p>Soy el tercer párrafo</p>
</main>
<script>
        let capa=document.getE1ementById("principal");
        let nuevoP=document.createElement("p");
        nuevoP.textContent="Soy el nuevo párrafo";
```

```
    capa.appendChild(nuevoP);
</script>
```

En el código anterior la capa HTML de tipo **main** identificada como ***principal*** tiene tres párrafos. El código JavaScript crea un nuevo párrafo y le coloca el texto ***"Soy el nuevo párrafo".*** Cuando usamos el método **appendChild** para añadirlo como hijo de la capa principal, aparecerá al final de los párrafos anteriores. Resultado:

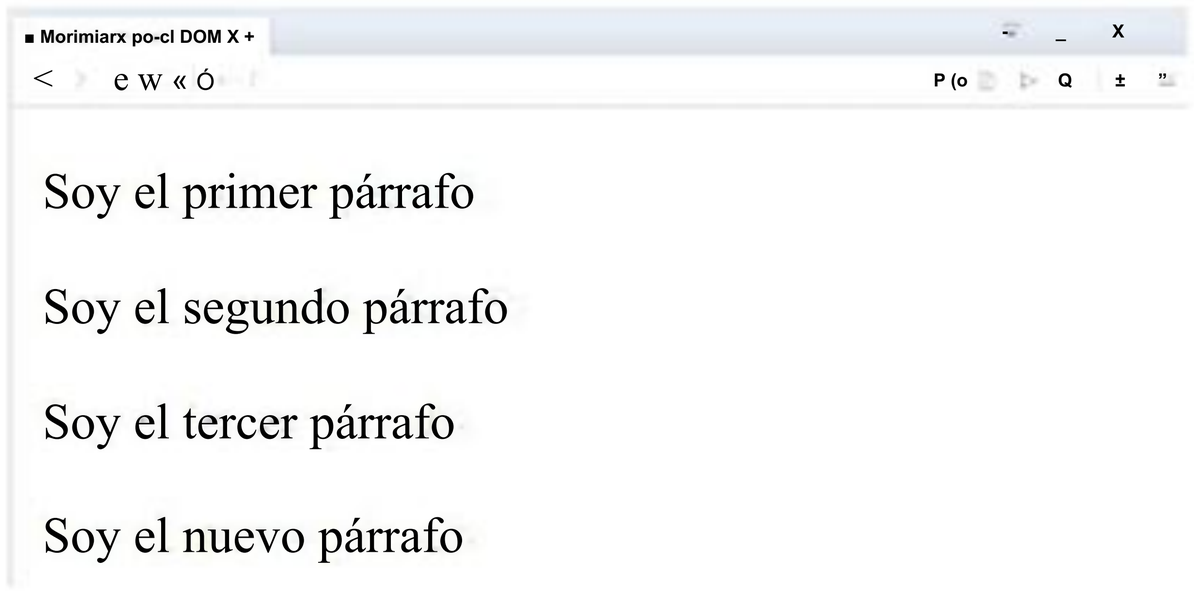


Figura 7.4: Nodo añadido al final de un elemento a través de **appendChild**

7.4.6.4 INSERCIÓN DE NODOS EN POSICIONES CONCRETAS

El método anterior tiene como inconveniente, que el elemento que deseamos añadir debe ir siempre colocado al final. Por ello disponemos de otro método llamado **insertBefore** que permite indicar la posición en la que queremos colocar el elemento. Para ello el nodo que será padre del que queremos añadir, debe de tener nodos hijo. De otro modo deberíamos usar **appendChild.**

El método **insertBefore** tiene dos parámetros: el primero es el nodo que deseamos añadir y el segundo es el nodo hijo delante del cual quedará colocado el que deseamos añadir. Evidentemente no podemos, con este método, añadir un nodo al final, pero para eso disponemos de appendChild.

Usando el mismo código HTML, veamos cómo colocar un nuevo párrafo como segundo párrafo:

```
let capa=document.getElementById("principal");
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
//El elemento pPosterior es el segundo párrafo de la capa principal
let pPosterior=document.querySelectorAll(
                                        "«principal p:nth-of-type(2)")[0];
capa.insertBefore(nuevoP,pPosterior);
```

El resultado es el que se puede observar en la Figura 7.5.

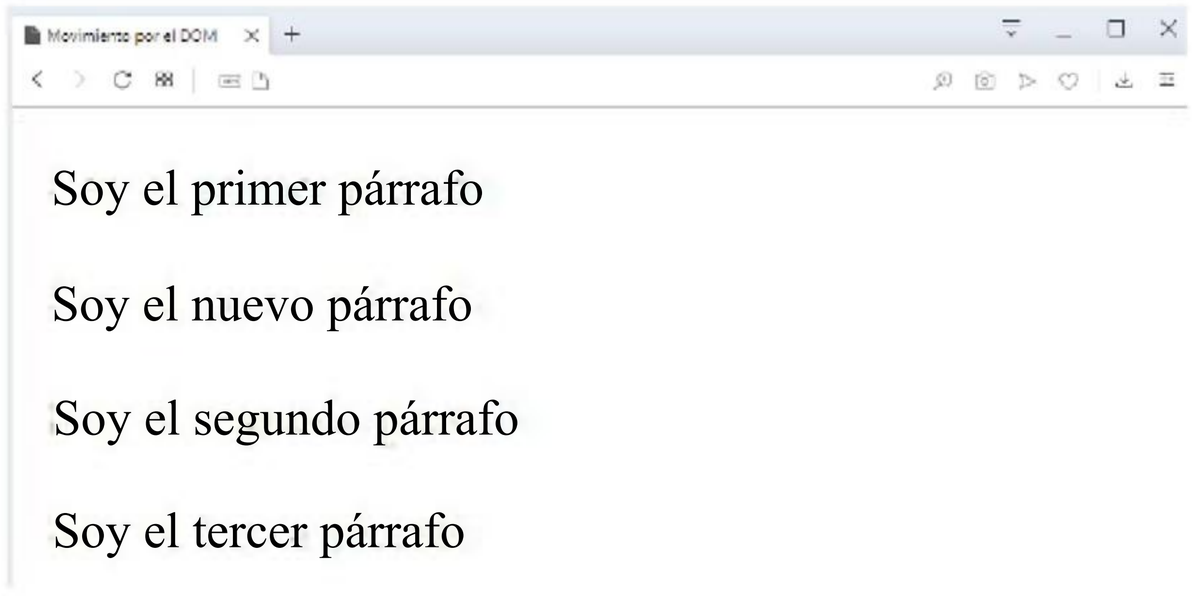


Figura 7.5: Nodo añadido como segundo hijo a través de **insertBefore**

7.4.7 REEMPLAZAR ELEMENTOS

El método **replaceChild** es más agresivo. Permite cambiar un nodo por otro. Necesita como parámetros: primero el nuevo nodo que se desea colocar y después el nodo que se desea quitar. El nuevo nodo reemplaza el antiguo.

El método devuelve una referencia al nodo retirado, que realmente no se quita del todo, sino que simplemente se retira del árbol del DOM. Podemos usar la referencia al nodo quitado para colocarlo dentro de otro elemento.

Ejemplo:

```
let capa=document.getElementById("principal");
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
//El elemento pPosterior es el segundo párrafo de la capa principal
let pPosterior=document.querySelectorAl1(
                                        "«principal p:nth-of-type(2)")[0];
capa.replaceChi Id(nuevoP,pPosterior);
```

Ha desaparecido el segundo párrafo y en su posición aparece el nuevo nodo.



Figura 7.6: El segundo hijo se cambia por el nuevo elemento mediante **replaceChild**

7.4.8 ELIMINAR ELEMENTOS

El método **removeChild** es un método de los nodos que recibe como único parámetro el nodo a eliminar. Ese nodo no se elimina realmente de la memoria solo se retira del árbol de elementos, pero permanece en memoria. De hecho, el método devuelve una referencia al objeto retirado.

En este ejemplo se quitaría el segundo párrafo y se colocaría al final con ayuda de **appendChild:**

```
let capa=document.getElementById("principal");
let pSegunda=document.querySelectorAl1(
                                        "«principal p:nth-of-type(2)")[0];
//se elimina el segundo párrafo
capa.removeChild(pSegunda);
capa.appendChild(pSegunda);
```

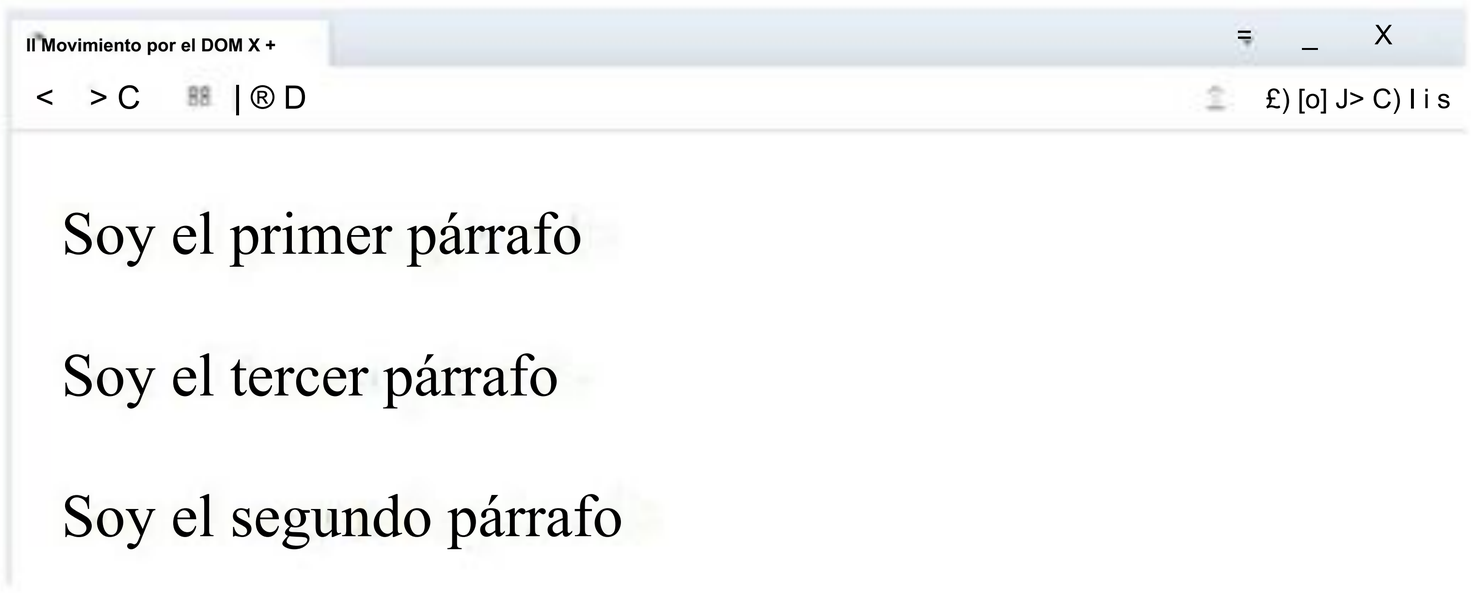


Figura 7.7: El segundo párrafo pasa a ser el último gracias a **removeChild**

7.4.9 COLECCIONES VIVAS

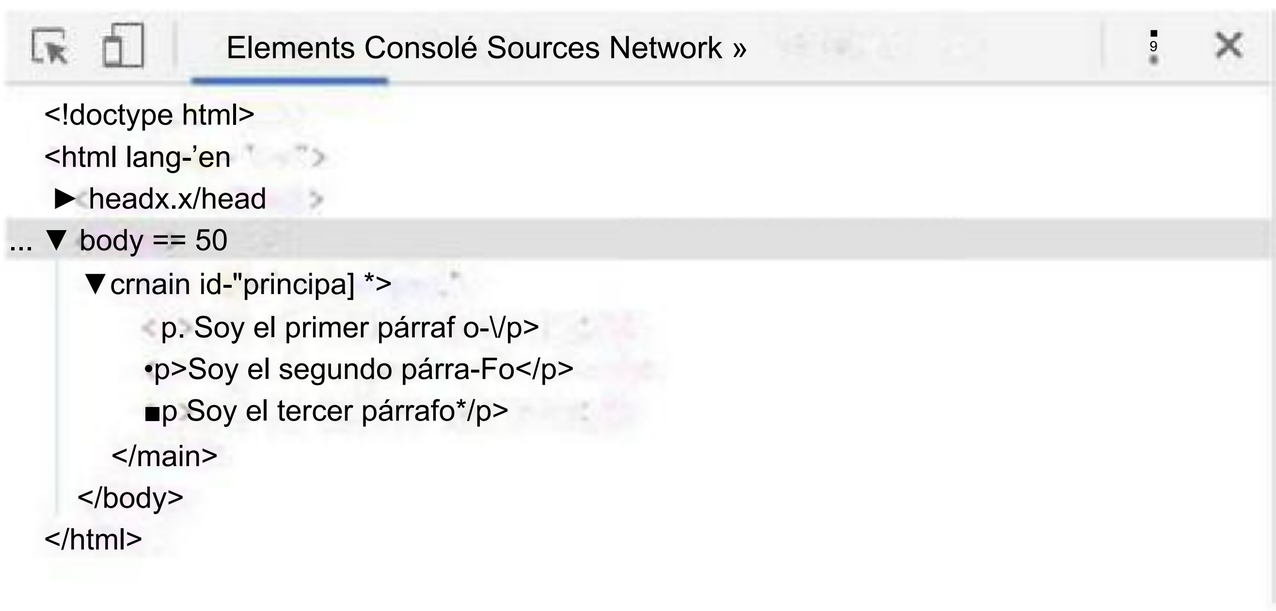


Figura 7.8: El panel de depuración de Google Chrome mostrando los elementos originales sin JavaScript

Las acciones vistas en los apartados anteriores permiten modificar los elementos que inicialmente aparecían en la página. Los elementos creados, modificados o eliminados desde JavaScript se dice que han sido acciones dinámicas o acciones vivas o en directo.

No hay ningún problema en hacer esas modificaciones, es verdad que el código HTML del documento mostrará elementos que ya no están en la página. Pero todos los navegadores permiten ver la realidad de los elementos tal cual se muestran en cada instante a través del panel de depuración.

Pero sí hay problemas con algunos métodos porque algunos retornan colecciones de datos que no se actualizan cuando la página cambia. La mayoría sí lo hace como podemos ver en este ejemplo:

```
let capa=document.getElementById("principal");
let hijos=capa.children;
consolé.log(hijos.length);
```

Supongamos, si seguimos teniendo el mismo código HTML original de los ejemplos anteriores, que por consola aparece el número 3, indicando que el elemento con identificador **principal** tiene tres hijos. Este otro código añade un nuevo hijo:

```
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
capa.appendChiId(nuevoP);
consolé.log(hijos.length);
```

Escribe directamente 4. Eso significa que la propiedad **children** retorna una **colección viva** de nodos, porque se actualiza al instante con los cambios que javaScript realiza al DOM.

Lo mismo si borramos elementos:

```
capa.removeChiId(capa.firstElementChiId);
consolé.log(hijos.length);
```

Vuelve a aparecer el número 3 por consola y vemos que las capas que quedan son estas:



Figura 7.9: El panel de depuración de Google Chrome muestra los elementos ***vivos*** tal cual les va modificando el código JavaScript

La colección **children** es una lista viva que se actualiza con los cambios en el DOM. Son también métodos que devuelven colecciones vivas:

- **getElementsByTagName**
- **getElementsByClassName**

- **atributes**
- **cíassList**
- **children**

Es decir, la mayoría de métodos.

De los que no son vivos, el más preocupante es **querySelectorAll,** que es muy utilizado por su potencia para seleccionar elementos, pero que la colección de elementos que devuelve no está viva, es **estática.** Si queremos la lista al día deberemos ejecutarlo de nuevo. Para entender mejor el problema veamos este código:

```
<main id="principa1">
            <p class="especial">Uno</p>
            <p>Dos</p>
            <p class="especial">Tres</p>
</main>
<p>Cuatro</p>
<p>Cinco< /p>
<script>
   let principal=document.getElementById("principal");
   let 1istal=document.getElementsByClassName("especial");
   let 1ista2=document.querySelectorAl1(".especial");
   consolé.log(1istal.length);//Escribe 2
   consolé.log(1ista2.length);//Escribe 2
   //Eliminamos el párrafo de clase especial con el texto "Uno"
   principal.removeChild(pri ncipal.children[0]);
   consolé.log(1istal.length); //Escribe 1, está al día
   consolé.log(1ista2.length); //Escribe 2, no actualiza los cambios
<script>
```

7.4.10 OTRAS PROPIEDADES Y MÉTODOS DE LOS ELEMENTOS

Los objetos de tipo **Element** poseen numerosas propiedades, en los apartados anteriores hemos visto las fundamentales, pero lo cierto es que hay bastantes más. Algunas son repetitivas sobre lo ya visto y otras se consideran obsoletas, enunciamos algunas de las que nos pueden ser útiles, y que no hemos visto en los apartados anteriores.

PROPIEDAD O MÉTODO	USO
clientHeight	Altura en píxeles del elemento.
clientWidth	Anchura en píxeles del elemento.
clientLeft	Anchura del borde izquierdo del elemento en píxeles.
clientTop	Altura del borde izquierdo del elemento en píxeles.

PROPIEDAD O MÉTODO	USO
outerHTML	A diferencia de **innerHTML,** esta propiedad devuelve el código HTML completo del elemento, incluidas sus propias etiquetas
scrollHeight	Tamaño completo en píxeles del scroll vertical del elemento. Índica cuánto se puede desplazar en vertical el contenido del elemento haciendo scroll. Por ejemplo, si tenemos un elemento que mide 500 píxeles en vertical (clientHeight=500) y tenemos que puede haber 1500 píxeles de scrollHeight, eso significa que el contenido interior mide tres veces lo **que** mide el elemento.
scrollWidth	Tamaño completo en píxeles del scroll horizontal del elemento.
scrollTop	Posición actual del scroll vertical en el elemento. Si el **scrollHeight** fuera de 1500 y el **scrollTop** es de 750, estamos a mitad de camino en el desplazamiento vertical. Esta propiedad admite modificar su valor para hacer de forma manual un desplazamiento.
scrollLeft	Posición actual del scroll horizontal en el elemento. Esta propiedad admite modificar su valor.
scroll(x,y) o bien **scroll(objetoScroIl)**	Desplaza la página. Permite indicar el desplazamiento mediante dos coordenadas ***(x*** horizontal ***ey*** en vertical), o bien por un objeto de tipo **ScrollToOptions** (7.1.6 ***"Otras propiedades y Métodos de Window",*** en la página 255).
scrollBy(x,y) o bien **scrollBy(ob j etoScroll)**	Los parámetros funcionan igual que con la propiedad anterior, pero ahora el desplazamiento es relativo respecto a la posición actual.

7.5 TEMPORIZADORES

7.5.1 ¿QUÉ SON LOS TEMPORIZADORES?

El objeto window posee varios métodos relacionados con temporizadores. El manejo del tiempo es un elemento fundamental de la programación. Es fundamental en la programación de juegos, en la creación de animaciones, en publicidad y en otras muchas áreas.

Todas estas capacidades tienen que ver con dos métodos que permiten ejecutar un determinado código (normalmente por medio de una función callback) cuando pase cierto tiempo.

El hecho de que se ejecute un código cuando ocurra un tiempo concreto nos aproxima a la gestión de eventos que es la base del tema siguiente. No obstante, la buena noticia es que el manejo de temporizadores en JavaScript es francamente fácil.

7.5.2 SETTIMEOUT

El método **setTimeout** de **window** tiene dos parámetros: el primero es la función que se ejecutará cuando se cumpla el tiempo, el segundo es un valor numérico que indica el tiempo a cumplir en milisegundos. Ejemplo:

```
setTimeout(()=>alert("Hola"),5000);
```

El primer parámetro del código anterior es la función flecha ()=>alert("Hola") esa función simplemente lanza el mensaje "Hola" en un cuadro de diálogo. El segundo parámetro indica que se esperará 5 segundos antes de ejecutar el código de la función.

El método setTimeout devuelve un número que identifica al temporizador en sí. Es decir, a cada temporizador lanzado con setTimeout se le asigna un número o identificador. Ese número se puede usar para cancelar el temporizador mediante **clearTimeout.**

Ejemplo:

```
var templ=setTimeout(()=>alert("Hola"),5000);
clearTimeOut(templ);
```

Tal cual está escrito el código anterior, jamás aparecerá el mensaje **Hola** porque se cancela el temporizador inmediatamente (no habrán pasado los 5 segundos).

7.5.3 SETINTERVAL

El método **setlnterval** es muy similar al anterior. Tiene los mismos parámetros y devuelve también un identificador de temporizador que puede ser almacenado en una variable para poder cancelar el temporizador con un método llamado **clearlnterval.**

La diferencia es que, en este caso, el código asignado al temporizador se invoca cada vez que pase el tiempo indicado. Es decir, setTimeout invoca al código una sola vez y setlnterval lo invoca constantemente:

```
var temp2=setlnterval(()=>alert("Hola"),5000);
```

El código es casi idéntico al que vimos en el apartado anterior. Pero ahora el cuadro de mensaje diciendo ***Hola*** aparecerá cada 5 segundos y no solo una vez como antes. Jamás dejará de aparecer a no ser que lo cancelemos:

```
var cont=0;
var temp2=setlnterval(function(){
    alert("Hola");
    cont++;
    if(cont>=10) clearlnterval(temp2);
},5000);
```

Este código muestra la palabra ***Hola*** en un mensaje cada 5 segundos. Lo hará 10 veces y luego ya no lo hará, ya que cuando el contador valga diez, se habrá eliminado el temporizador mediante clearlnterval.

7.6 COOKIES

7.6.1 ¿QUÉ SON LAS COOKIES?

Las cookies no son más que una serie de archivos de texto que las aplicaciones web graban en el ordenador del usuario que contienen datos que la aplicación puede volver a recuperar. Dado que **http** es un protocolo sin estado, cada petición http es independiente de la anterior. Las cookies se envían automáticamente en la cabecera de cada petición permitiendo a las aplicaciones recordar aspectos de peticiones anteriores gracias a los datos que han grabado en las cookies.

Cada cookie puede grabar como mucho 4KB de datos y se permiten 20 cookies por dominio. Suelen ser suficiente estas restricciones con casi cualquier tipo de aplicación.

Todos los usuarios europeos de Internet han oído hablar de las cookies, ya que en la Unión Europea hay una directiva relativa a su uso que obliga a las aplicaciones web a pedir permiso al usuario antes de poder grabar cookies. Ese aviso tiene que incluir una explicación (o un enlace a la explicación) que aclare la utilización y necesidad que la aplicación requiere de las cookies.

En realidad, esta directiva distingue entre las cookies técnicamente necesarias y las no necesarias. Las primeras son imprescindibles para el correcto funcionamiento de la aplicación. Un ejemplo de este tipo de cookies son las que graban los datos de inicio de sesión del usuario que es clave para que ese usuario acceda a su carrito de la compra o a su espacio personal. Las cookies técnicamente necesarias, se permite que se graben directamente y que luego se pida permiso al usuario. Si el usuario deniega el permiso habría que borrarlas y, siendo imprescindibles, la aplicación dejará de trabajar con normalidad, pudiendo incluso redirigir al usuario a una página en blanco.

Las cookies que, según la UE, se consideran técnicamente no necesarias son las cookies de seguimiento y de análisis de la acción del usuario. Estas no se pueden utilizar antes de pedir permiso al usuario.

La controversia y el uso malévolo por parte de algunos servicios de las cookies, las ha creado mala fama, por lo que muchas aplicaciones para almacenar los datos no imprescindibles utilizan otras opciones como, por ejemplo, la **API localStorage** que tiene más posibilidades para almacenar datos.

En todo caso, las cookies siguen siendo la opción habitual para almacenar datos relacionados con la identificación del usuario ya que, como hemos comentado, tienen la ventaja de enviarse automáticamente como datos anexos en cada petición http, tanto del cliente como del servidor.

7.6.2 LECTURA Y GRABACIÓN DE COOKIES

Es el objeto document, una vez más, es el que posee la propiedad cookie que es la que controla las cookies. A esta propiedad se asigna un texto que es el que graba en sí la cookie. La sintaxis básica es la siguiente:

```
document.cookie="nombreCookie=valor";
```

Por ejemplo:

```
document.cookie="usuario=Jorge";
```

Hemos grabado una nueva cookie con el nombre ***usuario*** y el valor ***Jorge.*** Cada nueva cookie puede utilizar el mismo formato, tendría otro nombre y el valor que le queramos dar.

Si queremos modificar el valor de una cookie ya creada basta con indicar el mismo nombre e indicar el nuevo valor:

```
document.cookie="usuario=Ana";
```

Para leer el contenido basta con ver el contenido de esta propiedad. Ejemplo:

```
consolé.log(document.cookie);
```

El resultado podría ser:

```
nombre=Jorge; lenguaje=es; fechaUltimaSesion=ll/8/2019
```

El problema en la lectura es cómo leer cada cookie por separado. En ese sentido nos puede ayudar la función **split** (véase ***"split",*** en la página 125) de los string la cual puede dividir las cookies usando el separador entre cada una (que es el punto y coma seguido de un espacio en blanco). Cada elemento del array resultante lo volveremos a dividir mediante el mismo método split para separar el nombre y el valor.

```
let arrayCookie=document.cookie. splitC'; ");
for(let cookie of arrayCookie){
    let [nombre,valor]=cookie.split("=");
    consolé.log('La cookie "${nombre}" tiene el valor "${valor}"');
}
```

7.6.3 FECHAS DE EXPIRACIÓN

Por defecto, las cookies se eliminan cuando el usuario finaliza su sesión. El fin de sesión ocurre cuando se cierra el navegador.

Pero hay cookies que se requiere que sean **persistentes;** es decir, que sus valores aguanten entre sesión y sesión. Para ello hay que indicar una fecha de expiración futura. La fecha de expiración es el momento máximo en el que esa cookie sobrevive. Los navegadores eliminan las cookies que han caducado cada vez que el usuario inicia sesión en el dominio que grabó la cookie.

La fecha de expiración se debe indicar al modificar el valor de una cookie colocando un punto y coma tras el valor e indicando la palabra **expires** seguida de la fecha de expiración en formato UTC. La forma de grabar la fecha de expiración en JavaScript es obtener la fecha actual, después añadimos el tiempo de duración de la cookie. Añadir tiempo a las fechas mediante objetos de tipo **Date** nos obliga a hacerlo en milisegundos. Finalmente, la fecha ya incrementada se pasa a formato UTC.

Veamos un ejemplo:

```
//Crea un objeto de tipo fecha con la fecha actual
let hoy=new Date();
//crea una variable tomando los milisegundos de hoy
//y sumando los milisegundos de la semana que viene
let caducidadMs=hoy.getTime() + 1000 * 60 * 60 * 24 * 7;
//Convierte los milisegundos a un objeto Date que representa la fecha
//de dentro de una semana
let caducidad=new Date(caducidadMs);
//Se coloca la fecha de expiración en formato UTC, la cookie "usuario"
//caducará dentro de una semana
document. cookie = 'usuario=Jorge;expires=${caducidad. tollTCString()} ';
```

7.6.4 RUTA Y DOMINIO DE LAS COOKIES

Por defecto una cookie solo puede ser leída por aplicaciones del dominio que creó la cookie y que estén en el mismo directorio en el que se creó la cookie. Eso significa, por ejemplo que si una cookie se graba en un directorio como ***www.jorgesanchez.net/manuales*****,** los documentos de la raíz, fuera de la carpeta de ***manuales,*** no pueden leer esas cookies.

La propiedad **path** de las cookies permite indicar la ruta raíz desde la que cualquier aplicación puede leer esa cookie:

```
document.cookie="usuario=Jorge;path=/";
```

Con la propiedad **path,** hemos indicado que la raíz desde la que cualquier aplicación puede leer la cookie es "/", que es el nombre del directorio raíz del dominio. Es decir, en este caso, cualquier documento del dominio puede leer esa cookie, ya que se incluyen los subdirectorios de la ruta.

Con el dominio ocurre lo mismo. Si hemos grabado la cookie en un subdominio y queremos que otro subdominio lea esa cookie, por defecto no podrá. Por ello, deberemos indicar un dominio más general. Por ejemplo:

```
document.cookie="teleo=si;path=/;domain=jorgesanchez.net";
```

Al indicar como dominio: ***jorgesanchez.net*****,** aunque hayamos grabado la cookie en un documento del subdominio ***www.jorgesanchez.net*****,** un documento de ***practicas.jorgesanchez.net*** también las podrá leer.

7.6.5 BORRAR COOKIES

La forma de que una aplicación elimine una de sus cookies es indicar como fecha de expiración una fecha del pasado. Ejemplo:

```
document.cookie="usuario=Jorge;expires=Sat, 01 Jan 2000 00:00:01 GMT";
```

Al colocar como fecha el 1 de Enero del año 2000, la cookie quedará caduca y el navegador la borrará.

7.7 PRÁCTICAS RESUELTAS

Práctica 7.1: Añadir un párrafo

- En editores como **Visual Studio Code,** que tienen el módulo **Emmet** incorporado, se pueden crear párrafos con el texto clásico de pruebas ***"Lorem ipsum...".*** Si estando en el body de una nueva página web escribimos ol > 1 **i*9> lorem** y luego pulsamos el tabulador, se crearán 9 párrafos numerados con el texto en cuestión.
- Una vez creados los párrafos la página consistirá en pedir al usuario un texto y un número (del 1 al 10) para la posición de ese texto. El texto se mostrará en negrita
- Se mostrará el párrafo (sin eliminar ninguno de los anteriores) en la posición indicada. Si se indica el 3, se dejarán sin cambiar el primer y segundo párrafos iniciales, el nuevo se colocará como tercer y el que antes era el tercero, ahora será el cuarto y así sucesivamente.
- Si el número no está en ese rango (o no es un número) se avisa del error y se pide de nuevo.
- Si el usuario cancela, la aplicación no hace nada.

SOLUCIÓN: PRÁCTICA 7.1

Basta con generar el código inicial de la página. La validación de los datos es algo ya trabajado en unidades anteriores y después se trata de usar los métodos de creación y colocación de nodos vistos en esta unidad.

El código de la aplicación completa es este:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <meta name="viewport" contenta"width=device-width,
initial-scale=l.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Añadir       párrafo</title>
</head>
<body>
        <ol >
                <li>Lorem    ipsum, dolor sit amet consectetur adipisicing elit.
Itaque,      rerum rem     dolorem amet, iste iure perferendis veritatis praesen-
tium qui      reprehenderit molestias commodi expedita quasi quod. Beatae
voluptate est harum aliquid?</li>
                <li>Est,                          aliquam! Témpora, eos rerum perferendis architecto
inventore cumque impedit ratione dolores dicta non repellat tempore, vero
quisquam numquam quasi magnam similique. Ad molestiae máxime voluptas
provident praesentium illo sunt.</li>
```

```
            <li>Mollitia eos nulla atque quae tenetur facere corrupti ipsum
quas nemo, illum fugit? Nisi placeat nobis quis quo pariatur ñeque eos,
sit ab distinctio dicta sunt corporis aliquam témpora eius.</li>
            <li>Nobis dolor mínima laborum quod quibusdam iure esse ex?
Provident voluptas labore sunt ullam unde rem veniam ñeque veritatis
officiis accusamus magni, ipsum nulla, tenetur asperiores a odit tempori-
bus amet.</li>
            <li>Ea aspernatur assumenda, repudiandae minus, ratione nesciunt
vitae fuga delectus incidunt laboriosam, nisi amet mollitia qui aliquam!
Culpa ratione, molestiae vel, eligendi recusandae laudantium similique
maiores quos fuga accusamus ex?</li>
            <li>At autem possimus magnam numquam provident adipisci aliquam
eaque quidem témpora ex nesciunt molestiae dolorem corporis, placeat iste
hic, facilis totam rem officia ratione, cupiditate quisquam? Laborum ali-
quid dolore dolores.</li>
            <li>Quis quasi provident itaque. Totam dolorem ipsam necessitati-
bus architecto unde qui doloremque modi molestiae incidunt a ad reprehen-
derit, deleniti ullam at culpa blanditiis odio ipsa commodi perspiciatis
enim iusto quae.</li>
            <li>Facere laboriosam dicta repellat cumque consequatur hic
pariatur doloribus exercitationem distinctio, sequi a cum culpa quam
laborum dolorem cupiditate dignissimos magni quo dolorum et unde vero
reprehenderit mínima. Ullam, iure?</li>
            <li>Quaerat voluptatum odit ab fugit reiciendis eius molestiae
natus. Consequuntur, perferendis! Quas exercitationem dolorem magnam hic!
Corrupti ea ratione unde error, mínima cum enim quaerat eaque debitis
quas. Culpa, magnam.</li>
        </ol>
<script>
    var texto=prompt("Escribe el texto que desees");
    var posición;
i f(texto!=nul1){
    do{
        posición^
            prompt("Escriba la posición de ese texto (número del 1 al 10)");
        i f(posición!=nul1){
            posición=parseInt(Number(posición));
            if(isNaN(posición) I| posición<l
                I|posición>10){
                    alert("Número no válido")
            }
        }
    }while(posición!=null &&
                    (isNaN(posición) ||
                     posición<l || posición>10)
                );
```

```
//colocación del texto en su posición
var 1ista=document.getElementsByTagName("ol")[0];
var nuevoElemento=document.createElement("1 i");
nuevoElemento.innerHTML=x <strong>${texto}</strongK;
i f(posición==10){
      //si el número es el 10 se añade al final
       1ista.appendChi Id(nuevoElemento);
  }
  else{
      //si no se añade delante de la posición indicada
      //Se resta uno porque se numera desde el 0
       let 1iSiguiente=lista.children[posición-1];
            1ista.insertBefore(nuevoElemento,1iSiguiente)
        }
   }
</script>
</body>
</html>
```

Práctica 7.2: Salto a Google en 5 segundos

- Muestra una página que diga **Nos vamos** y que tras cinco segundos nos redirija a la página del buscador Google.

SOLUCIÓN: PRÁCTICA 7.2

En este caso se trata de usar intervalos y de utilizar el objeto **location.** El código es muy sencillo:

```
<!DOCTYPE html>
<html lang="es">
<head>
      cmeta charset="UTF-8">
      <title>Marcha a Google</title>
</head>
<body>
      <p>Nos vamos</p>
</body>
<script>
      setTimeout(()=>{location="https://google.es"},5000);
</script>
</html>
```

Práctica 7.3: Contador de visitas con aviso de cookies

- Crear una aplicación que nos muestre el número de veces que hemos entrado.
- Lo haremos mediante una cookie que caducará después de un año.
- Pero primero avisaremos con un cuadro de confirmación, que grabamos cookies en la página. Si el usuario no acepta cookies las borramos y no mostraremos las visitas. Ese mensaje aparecerá al principio.
- Ese cuadro sale una vez por sesión

SOLUCIÓN: PRÁCTICA 7.3

En esta página lo primero es el cuadro de confirmación de uso de cookies. Si el usuario no acepta las cookies, la página saldrá en blanco. Cuando el usuario elija sí, grabaremos una cookie que haga que no preguntemos de nuevo en la misma sesión. Esa cookie se graba sin fecha de caducidad.

Si el usuario ha elegido que sí quiere cookies, entonces indicaremos las visitas gracias a una cookie en la que grabamos las visitas, y cuyo valor incrementamos en cada visita. A esa cookie sí el colocamos caducidad por un año.

Código:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Contador de visitas</title>
</head>
<body>

<script>
let hayCookies=false;
//primero leemos si en esta sesión ya hemos
//aceptado el cuadro
let exp=/(cookie=)((si I no))/;
let res=exp.exec(document.cookie);
i f(res){
        i f(res[2]=="si") hayCookies=true;
}
i f(!hayCookies){
        hayCookies=confirm(
                "Esta página usa una cookie para almacenar" +
                "el número de visitas que hace cada usuario." +
                "Si está conforme con ello, pulse Sí\n" +
                "De otro modo la página no realizará su labor "+
                "y eliminará las cookies existentes");
```

```
    }
    if(hayCookies){
        //grabamos la cookie que indica que hemos aceptado el cuadro
        document.cookie="cookie=si";
        //Expresión Regular para sacar el número de visitas
        exp=/(visitas=)(\d+)/;
        //El tercer elemento de exec son las visitas: \d+
        res=exp.exec(document.cookie);
        //Las visitas valen 1 si no hay cookie
        //y uno más que antes si sí la hay
        let visitas=(res?++res[2]:1);
        //cálculo de la caducidad
        let hoyMs=(new Date()).getTime();
        let caducidad=new Date(hoyMs+1000*60*60*24*365);
        //cálculo de las visitas
        document.cookie='visitas=${visitas};' +
                        'expires=${caducidad . tolITCStr ing()} ';

        //Mostrar visitas
        let p=document.createElement("p");
        p.innerHTML='Esta es tu visita número '+
                ' <strong>${visitas}</strong>';
        document.body.appendChild(p);

    }
    else{
        document.cookie="cookie=no";
        document.cookie="visitas=0;expires=Sat, 01 Jan 2000 00:00:01 GMT";
    }
</script>
</body>
</html>
```

Práctica 7.4: Buscar una palabra dentro del documento

- Crea una página que tenga mucho texto como contenido.
- Crea una aplicación que busque una palabra escrita por el usuario en un cuadro dentro del documento.
- La aplicación hará scroll hacia la palabra y esta aparecerá remarcada.

SOLUCIÓN: PRÁCTICA 7.4

Parece una práctica muy compleja pero, sin embargo, es muy sencilla, gracias a que el método find de la clase window lo hace todo. Se aconseja no utilizar este método en una aplicación final,

dado que no es parte de JavaScript estándar, pero es interesante como ejemplo de las cosas interesantes que están escondidas en muchos de los objetos del DOM.

Como ejemplo de página con bastante texto hemos copiado el texto de los capítulos 1 y 11 del Quijote de la dirección de la biblioteca virtual del Instituto Cervantes:

http://www.cervantesvirtual.com/obra-visor/el-ingenioso-hidalgo-don-qiiijote-de-la-mancha-6/html/

Suponemos ya creada la página habiendo copiado numerosos párrafos. En ella se cargaría el siguiente código JavaScript:

```
let texto=prompt("Escriba la palabra a buscar");
i f(texto){
   window.find(texto);
}
```

Práctica 7.5: Marcar palabras dentro del documento

- Esta práctica es una versión más avanzada de la anterior.
- Se trata de pedir una palabra y remarcar todas las apariciones de la palabra en el texto del documento marcándolas con una etiqueta **mark.**
- Esta puede ser una idea de resultado:

aventurero todo cuanto pensaba, veía o imaginaba le parecía ser hecho y pasar al modo de lo que había leído, luego que vio la venta se le representó que era un castillo con sus cuatro torres y chapiteles de luciente plata, sin faltarle su puente levadiza y honda cava, con todos aquellos adherentes que semejantes castillos se pintan. Fuese llegando a la venta, que a él le parecía castillo a poco trecho della detuvo las riendas a Rocinante, esperando que algún enano se pusiese entre las almenas a dar señal con alguna trompeta de que llegaba caballero al castillo. Pero como vio que se tardaban y que Rocinante se daba priesa por llegar a la caballeriza, se llegó a la puerta de la venta, y vio a las dos distraídas mozas que allí estaban, que a él le parecieron dos hermosas doncellas o dos graciosas damas que delante de la puerta del castillo se estaban solazando. En esto sucedió acaso que un porquero que andaba recogiendo de unos rastrojos una manada de puercos (que, sin perdón, así se llaman) tocó un cuerno, a cuya señal ellos se recogen, y al instante se le representó a don Quijote Lo que deseaba, que era que algún enano hacía señal de su venida, y así, con extraño contento llegó a la venta y a las damas, las cuales, como vieron venir un hombre de aquella suene armado, y con lanza y adarga, llenas de miedo se iban a entrar en 1 venta: pero don Quijote, coligiendo por su huida su miedo, alzándose la visera de papelón y descubriendo su seco y polvoroso rostro, con gentil talante y voz reposada, les dijo:

-Non fúyan las vuestras mercedes, ni teman desaguisado alguno; ca a la orden de caballería que profeso non toca ni atañe facerle a ninguno, cuanto más a tan altas doncellas como vuestras presencias demuestran.

Mirábanle las mozas, y andaban con los ojos buscándole el rostro, que la mala visera le encubría; mas como se oyeron llamar doncellas, cosa tan fuera de su profesión, no pudieron tener la ns fue de manera que don Quijote vino a correrse y a decirles:

-Bien parece la mesura en las iermosas, y es mucha sandez, además, la risa que de leve causa procede; pero no vos lo digo porque os acuitedes ni mostredes mal talante; que el mío non es de i que de sen iros.

El lenguaje, no entendido de las señoras, y el mal talle de nuestro caballero acrecentaba en ellas la risa, y en él el enojo, y pasara muy adelante si a aquel punto no saliera el ventero, hombre q por ser muy gordo, era muy pacifico, el cual, viendo aquella figura contrahecha, armada de armas tan desiguales como eran la brida, lanza, adarga y coselete, no estuvo en nada en acompañar las doncellas en las muestras de su contento. Mas, en efeto, temiendo la máquina de tantos pertrechos, determinó de hablarle comedidamente, y asi le dijo:

-Si vuestra merced, señor caballero, busca posada, amén del lecho (porque en esta venta no hay ninguno), todo lo demás se hallará en ella en mucha abundancia.

Viendo don Quijote la humildad del alcaide de la fortaleza, que tal le pareció a él el ventero y la venta, respondió:

-Para mi, señor castellano, cualquiera cosa basta, porque mis arreos son las armas, mi descanso el pelear, etc.

Pensó el huésped que el haberle llamado castellano había sido por haberle parecido de los sanos de Castilla, aunque él era andaluz, y de los de la playa de Sanlúcar, no menos ladrón que Caco menos maleante que estudiante o paje, y así le respondió:

-Según eso, las camas de vuestra merced serán duras peñas, y su dormir, siempre velar, y siendo asi, bien se puede apear, con seguridad de hallar en esta choza ocasión y ocasiones para no dormir en todo un año, cuanto más en una noche.

Figura 7.10: Palabras resaltadas como resultado de una búsqueda de palabras

SOLUCIÓN: PRÁCTICA 7.5

En este caso el código es más complejo. No vamos a depender del método **find,** sino que buscaremos una forma más manual de solucionar el problema. La idea es buscar la palabra dentro del contenido de cada elemento de la página. Una función nos ayudará a recorrer cada elemento y

otra a, una vez indicado un elemento, a buscar en él el texto y reemplazarlo, pero colocado dentro de una etiqueta **mark.**

Se utiliza la función **replace** que admite expresiones regulares y así se podrá reemplazar todas las apariciones de la palabra en el mismo elemento.

Código JavaScript (valdría para cualquier contenido):

```
/ **
  *   Busca el texto en un elemento y
  *   @param {string} palabra El texto a buscar
  *   @param {Element} elemento El elemento en el que se quiere buscar el
texto
  *   ©returns {boolean} true si la palabra se encontró
  */
function marcarPalabra(palabra,elemento){
        let texto=e1emento.textContent;
        //comprobamos si la palabra está en el elemento
        let pos=texto.indexOf(palabra);
        if(pos!=-l){
                //usamos una expresión regular para modificar todas
                //las apariciones de la palabra con el modificador g
                //El método replace de los trings admte expresiones
                //regulares
                let expReg=new RegExp(palabra,"g");
                //se reemplazan todas las apariciones de la palabra
                //contenido con el texto marcado
                elemento.innerHTML=elemento.innerHTML.replace(
                        expReg,"<mark>"+palabra+"</mark>");
                return true;
        }
        else
                return false;
}

/ **
  *   Función recursiva que usa la función marcarPalabra
  *   para buscar en todos los hijos del elemento que se pase
  *   la palabra que se busque.
  *   Cuando un hijo tiene más hijos, se invoca de nuevo a
  *   esta misma función
  *   @param {Element} elemento
  *   @param {string} palabra
  */
function buscarTextoEnElemento(elemento,palabra){
        i f(elemento.children){
                //Recorrido de todos los hijos del elemento
```

```
            for(let hijo of elemento.children){
                if(hijo.children.length>0){
                    //Llamada recursiva si el elemento
                    //tiene hijos
                    buscarTextoEnElemento(hijo);
                }
                else{
                    marcarPalabra(palabra,hijo);
                }
            }
        }
}

//Código principal
let texto=prompt("Escriba la palabra a marcar");
i f(texto){
        buscarTextoEnElemento(document.body,texto);
}
```

7.8 PRÁCTICAS RECOMENDADAS

Práctica 7.6: Scroll suave

- Genera una página con mucho contenido. Por ejemplo, con p*50> lorem600 seguido del tabulador, o usando el contenido del Quijote sugerido en páginas anteriores.
- Consigue que, después de 5 segundos, la página, ella sola se vaya al final haciendo un desplazamiento suave.

Práctica 7.7: Cuenta palabras

- Genera una página con mucho contenido. Por ejemplo, con p*50> lorem600 seguido del tabulador, o usando el contenido del Quijote sugerido en páginas anteriores.
- Consigue que aparezca en un párrafo al principio del documento, el número de palabras que tiene el documento.

Práctica 7.8: Invertir párrafos

- Genera una página con muchos párrafos de tipo p. Por ejemplo, con p*50> lorem600 seguido del tabulador, o usando otro contenido, pero asegurándote de que tienes muchos párrafos de tipo p.
- Consigue por JavaScript que los párrafos aparezcan en la página en orden contrario al que aparecerían normalmente.

Práctica 7.9: Misma posición a la vuelta

- Genera una página con mucho contenido (usa el HTML de cualquiera de las anteriores).
- Consigue que cada vez que el usuario vuelva a la página (incluso cerrando el navegador), tenga la página con scroll en la misma posición que la última vez.
- Aunque necesitas cookies para hacer esta práctica, puedes obviar el aviso de uso.

Práctica 7.10: Párrafos poco a poco

- Nuevamente, usa una página con muchos párrafos.
- Consigue que el contenido de la página aparezca poco a poco, haciendo que aparezca un párrafo cada segundo.

Práctica 7.11: Ordenar lista

- Muestra una página que muestre una lista de palabras (haz unas 10 o 15).
- Pregunta al usuario (después de 3 segundos en la página) si quiere que la lista salga ordenada alfabéticamente.
- Si acepta la sugerencia, consigue que la lista aparezca ordenada.

7.9 RESUMEN DE LA UNIDAD

- El lenguaje javaScript se creó para dar dinamismo a las aplicaciones web. La forma de realizar esta labor se basa en una serie de objetos que parten de un objeto global llamado **window** que posee todos los demás objetos formando una estructura en forma de árbol conocida como **BOM *(Browser Object Model).***

- Entre los principales componentes de **window** están:

  - El objeto **navigator** que posee propiedades que nos permiten obtener información relacionada con el navegador del usuario.

  - El objeto **screen** que posee propiedades y métodos que nos permiten obtener información y realizar acciones sobre la pantalla del usuario como: obtener su tamaño y orientación.

  - El objeto **location** que nos permite obtener la URL de la página incluso desglosada en sus diferentes partes. También permite variar esa URL para redirigir la aplicación a otro destino.

  - El objeto **history** representa el historial de navegación, permite obtener información sobre el mismo e incluso utilizarlo desde JavaScript.

- El propio objeto window proporciona métodos muy potentes con los que obtener información y manipular aspectos como el tamaño real y el disponible de la ventana, la posición de las barras de desplazamiento generales, los estilos aplicados, la selección de texto o el acceso a objetos como los del evento actual, ***localStorage*** o la geolocalización.

- El objeto más importante de window es **document** que representa el propio documento HTML. El DOM (Modelo de Objetos del Documento) es otra estructura en forma de árbol donde la raíz es document y cada uno de sus elementos son nodos formados por el contenido de la página web.

- La estructura del DOM permite ver el documento HTML como una serie de objetos capaces de comunicarse y de ser modificados. Los nodos de la estructura los forman los elementos **(hl, section, p, strong,** etc.), el texto que contiene cada elemento, los atributos y otros tipos de nodos (comentarios, CDATA, etc.)

- Sobre cada nodo podemos obtener su contenido, nombre y tipo a través de propiedades comunes a todos ellos.

- En el caso de los elementos poseemos métodos pertenecientes a **document** que nos permiten seleccionarlos por su id, por la clase CSS, por el nombre de la etiqueta o incluso por cualquier selector del lenguaje CSS.

- Muchos métodos de selección devuelven listas de objetos que se parecen a los arrays en cuanto a que se recorre cada elemento con el mismo tipo de bucle que los arrays, pero que no poseen todos los métodos de los objetos Array. Aunque sí se pueden convertir fácilmente en arrays.

- Estas listas de nodos casi siempre son **vivas,** es decir, reflejan al instante los cambios en el DOM que produzcan las instrucciones JavaScript. Pero en algunos casos son estáticas y no reflejarán dichos cambios.

- Todos los elementos poseen métodos para obtener el contenido de sus atributos y para modificar su contenido. También disponen de métodos para eliminar atributos o para averiguar si tienen o no atributos.

- El contenido de los elementos se puede modificar con ayuda de las propiedades **textContent** e **innerHTML.** También disponemos de una propiedad llamada **dataset** con la que accedemos a los atributos de tipo data de los elementos.

- Los elementos también disponen de métodos para modificar las propiedades CSS de los elementos, así como para asignar y retirar clases CSS. El método de **window getComputedStyle** permite conocer las propiedades CSS en uso de un elemento independientemente de cómo se asignaron al mismo.

- Los métodos de recorrido por el árbol DOM permiten obtener los nodos hijos de un elemento, o solo el primero o solo el último de los hijos, también el nodo padre de un elemento, el hermano siguiente y el anterior y, además, hay versiones de la mayoría de esos métodos para recorrer solo nodos de elementos.

- Los métodos para crear **(createElement, createTextNode),** añadir **(appendChildren, insertBefore),** quitar **(removeChild)** y reemplazar elementos **(replaceChild)** permiten variar completamente la estructura de nodos del DOM.

- Además, sobre los elementos disponemos de propiedades, podemos obtener el tamaño y otros aspectos de los elementos y la posición del scroll, que además podremos modificar.

- El objeto window aporta propiedades que permiten crear temporizadores, funciones cuyo código se ejecuta cuando pase un determinado intervalo de tiempo. Se puede lanzar una sola vez **(setTimeout)** o continuamente cada vez que discurra ese intervalo **(setInterval).** En ambos casos se puede cancelar la ejecución cuando deseemos mediante los métodos **clearTimeout** o **clearInterval.**

- Por su parte, el objeto document aporta una propiedad llamada cookies que nos permite crear, modificar y eliminar cookies. Las directivas de la UE nos obligan a avisar si nuestra página usa cookies y la finalidad de las mismas en nuestras aplicaciones. El usuario puede no aceptar esas condiciones y, entonces, no podremos usarlas.

- Las cookies pueden asociarse solo a la sesión actual o bien asignarlas un tiempo de expiración futuro en cuyo caso son persistentes hasta esa fecha. La ruta y el dominio de aplicación son otras de las propiedades de las cookies que nos permiten determinar su rango de aplicación.

7.10 TEST DE REPASO

1.- **¿Qué instrucción nos permite volver a la primera página del historial?**

a) history.go(O)

b) history.go(-l)

c) history.go(-(history.length-l))

2.- history.go()¿Qué **propiedad del objeto navigator es la más fiable para saber la versión del navegador del usuario?**

a) browserVersion

b) versión

c) appVersion

d) appCode

e) userAgent

3.- **Queremos lograr seleccionar mediante** document.querySelectorAl1 **todas las celdas (td) que estén en la primera fila de una tabla ¿Qué selector habría que usar?**

a) td:first-of-type

b) tr:first-of-type td

c) td(l)

d) tr(l) td

e) table:first-row td

f) table tr td

4.- **¿Qué instrucción nos permite redirigir la página a otra URL?**

a) window.location="http://google.es"

b) location="http://google.es"

c) location.href="http://google.es"

d) window.location.href="http://google.es”

e) Cualquiera de las anteriores

5.- **¿Qué ocurre si asignamos a una cookie una fecha de expiración del pasado?**

a) El navegador elimina la cookie

b) La cookie solo tiene vigencia en la sesión actual

c) No ocurre nada, no se tiene en cuenta esa fecha

d) Ocurre un error que se mostrará por consola

6.- **¿Qué objeto forma la raíz de la estructura conocida como BOM?**

a) document

b) window

c) navigator

d) screen

7.- **¿Qué valor devuelve la propiedad node-Name en los nodos de tipo texto?**

a) El texto del nodo.

b) La palabra **text.**

c) La palabra **#text.**

d) El número 3.

8.- **Supongamos que una página web tiene solo este contenido en el body**

```
<p>Hoy es <mark>fiesta</mark></p>
```

Y que esa misma página tiene este código JavaScript

```
let el=document.
      getElementsByTagName("mark");
consolé.log(el.textContent);
```

9.- **Esperamos que salga la palabra *fiesta*, pero, en su lugar aparece *undefined*. ¿Por qué?**

a) Porque mark no es un nombre de elemento válido

b) Porque tendríamos que haber usado nodeValue y no textContent

c) Porque tendríamos que haber usado innerHTML y no textContent

d) Porque la variable el es una colección de elementos

10.- **¿Qué diferencia hay entre setTimeout y setInterval?**

a) setTimeout ejecuta una vez el código y setInterval indefinidamente

b) setInterval ejecuta una vez el código y setTimeout indefinidamente

c) Ninguna, son sinónimos

d) setInterval usa milisegundos y setTimeout fechas de tipo Date

1L- Observa este código:

```
<div id="pl''>
    <p>Necesitamos un
        <strong>nuevo</strong>
     juego de llaves</p>
</div>
```

12.- **¿Qué devuelve el método innerHTML de el elemento identificado como "pl"?**

a)
Necesitamos un
nuevo
juego de llaves

b) Necesitamos un
nuevo
juego de llaves

c) Necesitamos un
juego de llaves

d) Necesitamos un
juego de llaves

e) Necesitamos un nuevo
juego de llaves

13.- **Queremos saber qué color de letra tiene el elemento llamado *el* ¿Qué código lo consigue?**

a) el.style.color

b) el.getComputedStyle().color

c) window.getComputedSty!e(el).color

d) Todas son válidas

14.- **Un elemento llamado *el* tiene un atributo *lang* con valor es.¿ Qué haría este código en él?**

```
el.toggleAttribute("lang")
```

a) Quita el atributo **lang**

b) Deja el atributo **lang** vacío

c) Deja el atributo **lang** con su valor por defecto

d) En este caso no hace nada

5 Observa este código:

```
        <p id="pl">Necesitamos un
            <strong>nuevo</strong>
juego de llaves</p>
```

16.- **¿Qué devolvería este código JavaScript?**

```
let pl=document.
          getElementById("pl");
let hijos=pl.children;
for(let h of hijos)
    consolé.log(h.textContent);
```

a) Necesitamos un
nuevo
juego de llaves

b) Necesitamos un
nuevo</sstrong>
juego de llaves

c) undefined
nuevo
undefined

d) undefined
nuevo
undefined

e) nuevo

17.- **¿Qué hace el método nextSibling?**

a) Obtiene el siguiente nodo "hermano" del actual

b) Obtiene el siguiente nodo en el orden del código sin importar si es padre o hijo o hermano

c) Obtiene el siguiente hijo del nodo actual

d) Obtiene el siguiente elemento sin importar si es padre o hijo o hermano

18.- **Necesitamos modificar el color de fondo de un elemento llamado *el* ¿qué código es el válido?**

a) el.style["background-color"]="red"

b) el.style.backgroundColor="red"

c) el.style["backgroundColor"]="red"

d) Todas son válidas

UNIDAD 8

GESTIÓN DE EVENTOS

OBJETIVOS

— Reconocer las ventajas y aplicaciones del control de eventos en JavaScript

— Identificar los principales métodos de captura de eventos

— Asimilar el funcionamiento de los eventos y su propagación por los elementos del DOM

— Crear aplicaciones capaces de lanzar código asociado a eventos

— Reconocer las aplicaciones del objeto de evento y sus principales métodos y propiedades

— Diferenciar los principales tipos de eventos identificando sus acciones principales

— Aplicar el uso eventos a la interacción con el usuario utilizando formularios

— Resolver problemas de aplicaciones web completas utilizando control de eventos

CONTENIDOS

8.1 INTRODUCCIÓN A LOS EVENTOS

8.2 CAPTURA DE EVENTOS
- 8.2.1 MÉTODOS CLÁSICOS Y OBSOLETOS
- 8.2.2 MÉTODO ACONSEJABLE
- 8.2.3 PROPAGACIÓN DE EVENTOS
- 8.2.4 ANULAR EVENTOS
- 8.2.5 CAPTURA DE EVENTOS EN ELEMENTOS DINÁMICOS

8.3 OBJETO DE EVENTO
- 8.3.1 UTILIDAD Y USO DEL OBJETO DE EVENTO
- 8.3.2 OBTENER COORDENADAS DEL EVENTO
- 8.3.3 OBTENER LA TECLA PULSADA
- 8.3.4 ANULAR COMPORTAMIENTOS PREDETERMINADOS
- 8.3.5 CANCELAR PROPAGACIÓN
- 8.3.6 LANZAR EVENTOS

8.4 LISTA DE TIPOS DE EVENTOS
- 8.4.1 EVENTOS DE RATÓN
- 8.4.2 EVENTOS DE TECLADO
- 8.4.3 EVENTOS DE MOVIMIENTO EN LA VENTANA
- 8.4.4 EVENTOS SOBRE CARGA Y DESCARGA DE ELEMENTOS
- 8.4.5 OTROS EVENTOS

8.5 FORMULARIOS
- 8.5.1 EL FORMULARIO COMO OBJETO DEL DOM
- 8.5.2 EVENTOS DE FORMULARIO
- 8.5.3 PROPIEDADES DE LOS CONTROLES
- 8.5.4 MÉTODOS DE LOS CONTROLES

8.6 PRÁCTICAS SOLUCIONADAS

8.7 PRÁCTICAS RECOMENDADAS

8.8 RESUMEN DE LA UNIDAD

8.9 TEST DE REPASO

8.1 INTRODUCCIÓN A LOS EVENTOS

Gracias a esta unidad, tendremos ya todas las armas que permiten crear aplicaciones en el lado del cliente. Con esta unidad hemos llegado al colofón de la programación de aplicaciones web del lado del cliente. Los elementos fundamentales que aporta el lenguaje JavaScript para construir aplicaciones web en el lado del cliente estarían ya completos.

Con esta unidad podremos hacer casi cualquier tipo de aplicación web front-end (salvo las que requieran interacción con servidores). Los conceptos son sencillos, si esta unidad es larga se debe más a la cantidad de eventos ***y*** propiedades que aporta JavaScript y a la cantidad de prácticas que se proponen, y que se recomienda realizar encarecidamente, ya que el manejo de los eventos requiere mucha práctica para poder asimilar sus capacidades.

Los eventos son el mecanismo fundamental para comunicar la aplicación con el usuario. En esta interacción, la aplicación variará su funcionamiento en razón de las acciones realizadas por el usuario o usuaria.

Un evento no es más que un suceso, algo que ha ocurrido como resultado de un acto del usuario o por otras razones. Para que se considere realmente un evento, la aplicación tiene que ser capaz de detectarlo y, lo más importante, tiene que ser capaz de ejecutar un código asociado a dicho evento.

Los eventos se asocian a un elemento concreto del DOM, de modo que, por ejemplo, podemos hacer una cosa cuando el usuario haga clic sobre un párrafo y otra cuando el clic sea sobre otro párrafo.

La clave de los eventos en JavaScript es su capacidad asincrona. Podemos estar esperando que el usuario haga clic en un botón, pero no se puede parar el resto del código por esa espera. La espera se produce en segundo plano, es un proceso especial llamado **listener** (escuchante) el que está esperando a recibir la notificación del clic del usuario. La gestión de estos listeners la hace el navegador.

Por lo tanto, hay un proceso que automatiza la captura, de modo que cuando se produce un evento (por ejemplo, un clic sobre un elemento de la página), se ejecutará el código de una función callback. Lo único que hay que hacer es preparar la escucha del evento ***y esta se*** produce en segundo plano.

8.2 CAPTURA DE EVENTOS

8.2.1 MÉTODOS CLÁSICOS Y OBSOLETOS

8.2.1.1 ATRIBUTOS HTML ASOCIADOS A EVENTOS

Hace años, en las primeras versiones de JavaScript, la captura de un evento se realizaba desde el propio código HTML. Todavía se puede hacer porque se ha mantenido esta forma de captura por cuestiones de compatibilidad, pero no es nada aconsejable, ya que complica enormemente el mantenimiento del código al estar mezclado HTMl y JavaScript de forma confusa.

Este primer método utiliza atributos en las etiquetas de los elementos de la página que comienzan con la palabra ***on*** seguida del nombre del evento. Por ejemplo: ***onclick.*** Como valor del atributo se indica el nombre de la función que contiene el código a ejecutar. Ejemplo:

```
<p id="parrafol" onclick="alert('Me has hecho click')">
Soy 'clickable'</p>
<p id="parrafo2">Yo no</p>
```

EL primer párrafo usa el atributo onclick que permite lanzar código JavaScript cuando el usuario haga clic sobre el párrafo. El segundo párrafo no reacciona a la pulsación, porque no se ha indicado evento alguno.

8.2.1.2 PROPIEDADES DE LOS ELEMENTOS ASOCIADOS A EVENTOS

Es una mejora del método anterior, que permite que el código JavaScript se independice de HTML, lo que mejora su mantenimiento. Se trata de usar propiedades que tienen el mismo nombre que los atributos HTML: la palabra ***on*** seguida del nombre del evento **(onclick, onmouseover,** etc). Ha sido el método más utilizado durante muchos años, pero actualmente no se recomienda su uso.

Un ejemplo de funcionamiento sería el siguiente:

```
<p id="parrafol">Soy 'clickable'< /p>
<p id="parrafo2">Yo no< /p>
...
<script>
   let parrafol=document.getElementById("parrafol");
   parrafol.onclick=()=>{alert('Me has hecho click')};
</script>
```

Se ha asignado como función callback una función flecha que muestra el mismo mensaje de tipo ***alert*** que en el ejemplo anterior.

8.2.2 MÉTODO ACONSEJABLE

El método que se aconseja actualmente es utilizar el método **addEventListener** que está disponible en todos los elementos. A este método hay que indicarle el nombre del evento y la función callback a ejecutar. Es el método que se debe utilizar según la norma actual. Todos los navegadores actuales lo reconocen (las versiones de Internet Explorer anteriores a la 9, no lo hacen) y tiene una ventaja técnica sobre los dos métodos anteriores: se puede asignar más de una función al mismo evento.

El código equivalente de este método, equivalente a los usados con los métodos anteriores es:

```
<p id="parrafol">Soy elickable'</p>
<p id="parrafo2">Yo no</p>
■ ■■
<script>
```

```
  let parrafol=document.getElementById("parrafol");
  parrafol.addEventListener("click",()=>{alert('Me has hecho click')});
</script>
```

Para asignar varias acciones, basta con invocar otra vez a la función **addEventListener** con el mismo tipo de evento y usando otra función callback. Cuando se produzca el evento, el código de todas las funciones que se han asociado al evento, se ejecutará. En los métodos explicados anteriormente, solo se puede asignar una función a cada evento.

8.2.3 PROPAGACIÓN DE EVENTOS

Una cuestión fundamental en la captura de eventos es cómo se propagan los eventos sobre los contenedores de los elementos. Es decir, supongamos que tenemos una capa de tipo **div,** en ella un párrafo y dentro del párrafo un botón. Hacemos clic sobre el botón. ¿El párrafo podría capturar el click? ¿Y la capa?

Veamos lo que ocurre con esta página completa:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-      8" >
        <title>Propagación de eventos</title>
        <style>
                div{
                        position:fixed;
                        left:0;
                        top:0;
                        height:50%;
                        width:100%;
                        background-color:gray;
                }
                p{
                        height:50%;
                        margin:0;
                        background-color:1ightgray;
                }
        </style>
</head>
<body>
        <div>
                <P>
                        <button> ¡Hazme elic!</button>
                </p>
        </div>
<script>
let capa=document.querySelector("div");
```

```
let p=document.querySelector("p");
let boton=document.querySelector("button");
capa.addEventListener("click",()=>{consolé.log("click en capa")});
p.addEventListener("elick",()=>{consolé.log("elick en p")});
boton.addEventListener("elick",()=>{consolé.log("elick en boton")});
</script>
</body>
</html>
```

La situación comentada es la que presenta el código anterior. Hemos capturado los eventos de tipo ***click*** en los tres elementos **(dív, *p* y *button).*** Si ejecutando el código, hacemos clic en el botón, por consola veremos este texto:

```
click en boton
click en p
click en capa
```

El evento se propaga desde el botón (elemento más interior) hasta el elemento de tipo **div** (elemento más exterior, elemento más cercano a la raíz del DOM).

El proceso de captura de eventos se describe en esta imagen:

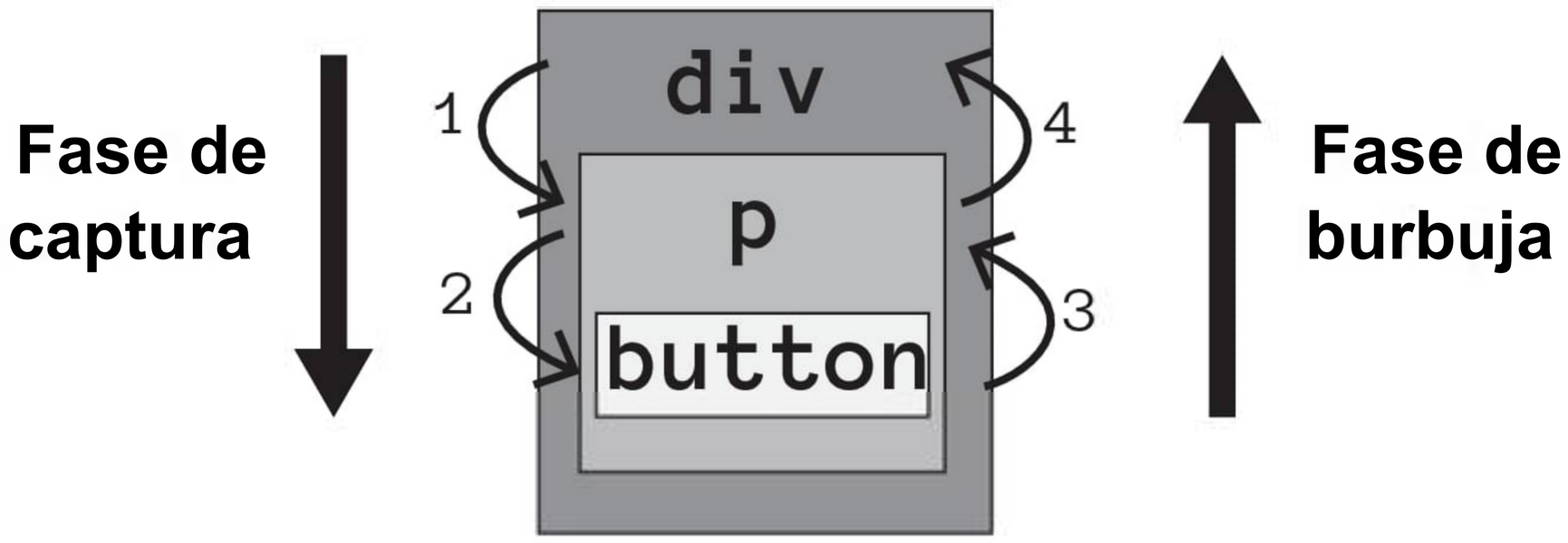


Figura8.1: Propagación de los eventos

Por defecto, la función asociada al evento se lanza en la fase de burbuja, por eso en el código anterior podemos observar primero el mensaje del botón, luego el del párrafo y luego el de la capa.

Podemos modificar este comportamiento, si indicamos que el lanzamiento sea en la fase de captura. Se consigue este efecto (no disponible en versiones de Internet Explorer anteriores a la 9) gracias a que, en realidad, el método **addEventListener** tiene un tercer parámetro relacionado con el tipo de captura. Se trata de un valor booleano que tiene estas posibilidades:

- **true.** El código de captura se lanza en la fase de captura.
- **false.** Es el valor por defecto. El código se lanza en la fase de burbuja.

Por lo tanto, si cambiamos las tres últimas líneas del código JavaScript anterior:

```
let capa=document.querySelector("div");
let p=document.querySelector("p”);
let boton=document.querySelector("button");
capa.addEventListener("click",
                                        ()=>{consolé.log("elick en   capa")},true);
p.addEventListener("click",()=>{consolé.log("elick                   en   p")},true);
boton.addEventListener("click",
                                        ()=>{consolé.log("elick en   boton")},true);
```

Si ejecutamos el código, por consola tendremos el siguiente resultado:

```
click en capa
click en p
click en boton
```

8.2.4 ANULAR EVENTOS

Imaginemos que deseamos que cuando hagamos un clic a un elemento concreto se nos muestre un mensaje, pero queremos que ese mensaje aparezca una sola vez. Por defecto, el evento se captura indefinidamente, pero existe un método contrario a **addEventListener** que tiene los mismos parámetros. Se trata de **removeEventListener.** Veamos un ejemplo:

```
<p>Hazme clic!< /P>
<script>
        function mensaje(){
                alert("!Me has hecho clic!")
                p.removeEventListener("click",mensaje);
        }

        let p=document.querySelector("p");
        p.addEventListener("click",mensaje);
</script>
```

Dentro de la propia función ***mensaje,*** tras escribir el mensaje en sí, se invoca al método ***removeEventListener*** el cual provoca que no se vuelva a invocar a esta función tras otro clic.

Hay que tener en cuenta que solo podemos retirar funciones de captura que tengan nombre, no podemos anular funciones anónimas asociadas a eventos.

8.2.5 CAPTURA DE EVENTOS EN ELEMENTOS DINÁMICOS

A la hora de implementar capturas de eventos, hay que tener en cuenta que los listeners se asocian a elementos del DOM que existan en ese momento. Eso puede dar lugar a problemas si no se tiene cuidado.

Supongamos que queremos crear un documento en el que consigamos que al hacer clic en todos los párrafos de tipo p, se muestre en un cuadro de alerta, el texto del párrafo. El código podría ser este:

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>
<p>Cuatro< /p>
<script>
let parrafos=document.querySelectorAl 1("p");
for(let párrafo of párrafos){
    párrafo.addEventListener("elick",()=>{alert(párrafo.textContent)});
}
```

La página posee cuatro párrafos, con los textos ***Uno, Dos, Tres*** y ***Cuatro.*** El código JavaScript asocia todos los párrafos a la variable que se llama ***párrafos.*** Pero, para implementar el control de eventos, necesitamos asignar un evento a cada párrafo. Por eso hay un bucle que recorre cada elemento devuelto por el método ***querySelectorAll*** y así vamos asignando a cada párrafo la captura del evento. Si ejecutamos este código, veremos que, efectivamente, al hacer clic en cualquiera de los cuatro párrafos, se nos muestra el mensaje.

Sin embargo, si añadimos un quinto párrafo de tipo p mediante JavaScript:

```
let nuevoParrafo=document.createElement("p");
nuevoParrafo.textContent="Cinco";
document.body.appendChild(nuevoParrafo);
```

Este párrafo aparecerá al final de los otros cuatro. Al hacer clic en él, no ocurrirá nada. La razón es lógica. Se ha creado después del bucle y por ello no se ha ejecutado el método **addEventListener** sobre él. Tendríamos que añadir el código que permita capturar el evento en el nuevo elemento.

8.3 OBJETO DE EVENTO

8.3.1 UTILIDAD Y USO DEL OBJETO DE EVENTO

Cuando se produce un evento, el navegador crea automáticamente un objeto cuyas propiedades pueden ser muy útiles a los desarrolladores. La información fundamental que graban son las coordenadas del cursor del ratón, si hay teclas pulsadas, el elemento que ha producido el evento, etc.

Ya hemos visto que, cuando se produce un evento, se invoca a automáticamente el código de una función. Pues bien, esa función puede tener un parámetro que será una referencia al objeto de evento. Gracias a ese parámetro podremos leer la información del evento.

Una de las propiedades que posee este objeto se llama **target** y es una referencia al elemento que causó el evento.

Veamos un ejemplo usando esta propiedad para entender cómo funciona el objeto de evento:

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>
<script>
	function escribeContenido(evento){
		consolé.log(evento.target.textContent);
	}
	let parrafos=document.querySelectorAl1("p");
	for(let párrafo of párrafos){
		párrafo.addEventListener("elick",escribeContenido);
	}
</script>
```

En este código se asigna a los elementos de los tres párrafos, el código de la función ***escribeContenido*** será ejecutado cuando se produzca un clic en cada párrafo. La función recibe un parámetro en el que se almacenará el objeto con los datos importantes del evento. La función, en este caso, usa la propiedad **target** para llegar al elemento que causó el clic y así acceder al texto de ese elemento que, finalmente, se mostrará en la consola.

8.3.2 OBTENER COORDENADAS DEL EVENTO

Los objetos de evento poseen dos propiedades para obtener las coordenadas del ratón en el momento del evento: **clientX** y **clientY.** La primera obtiene la posición horizontal en píxeles y la segunda la vertical. Ambas lo hacen utilizando como referencia la esquina superior izquierda de la ventana del navegador, que será el punto (0,0) de coordenadas.

Hay dos propiedades similares: **screenX** y **screenY.** La diferencia es que el origen de coordenadas no es la esquina del navegador, sino de la pantalla. Coinciden muchas veces, pero si la ventana no está maximizada, no coincidirán.

ACTIVIDAD 8.1: CARTEL PERSIGUIENDO AL RATÓN

- Es bueno que vayamos practicando eventos para solucionar tareas y así coger bagaje y habilidad en el uso de eventos y la aplicación de sus propiedades.
- Aprovechando estas propiedades, haz la Práctica 8.1 en la página 322, que consigue que un cartel con el texto ***Hola,*** persiga el cursor del usuario.

Otras coordenadas que se almacenan en el objeto de evento son **pageX** y **pageY.** Funcionan como **clientX** y **clientY,** pero no solo toman las de la ventana, tienen en cuenta el desplazamiento realizado en el elemento. Es decir, si hemos avanzado dos pantallas de 500 píxeles de alto usando las barras de desplazamiento y el cursor está a mitad de pantalla, **clientY** nos diría 250, mientras que **pageY** nos diría 1250. Depende de lo que deseemos nos vendrán mejor unas u otras coordenadas.

ACTIVIDAD 8.2: COMPARAR COORDENADAS

- Crea una página con mucho contenido. Puedes hacerlo rápido escribiendo en Visual Code, dentro del apartado body de la aplicación, la abreviatura **p*60>lorem500.** Tras pulsar el tabulador llenaremos la página de contenido.

- Escribe este código JavaScript justo antes de cerrar la etiqueta body:

```
<script>
document.addEventListener("mousemove",function(ev){
        consolé.log('screenX:${ev.screenX}, screenY:${ev.screenY}\n' +
                            'clientX:${ev.elientX}, elientY:${ev.elientY}\n' +
                            'pageX:${ev.pageX}, pageY:${ev.pageY}');
    })
</script>
```

- Haz que la ventana donde se ejecuta la página no esté maximizada. Desplázate hacia abajo en el documento varias pantallas y ahora, mostrando la consola, compara las coordenadas. Verás como ***pageY*** es la más grande porque cuenta todo el desplazamiento, como ***screenY*** es la segunda porque toma la pantalla y que ***clientY*** es la más pequeña.

8.3.3 OBTENER LA TECLA PULSADA

En eventos de teclado, el objeto de evento posee información sobre la tecla pulsada. Son importantes estos datos en aplicaciones como la programación de juegos o en el control de la entrada por el teclado.

La propiedad más importante es key que obtiene el texto que indica la tecla pulsada. Es el más cómodo de programar porque es fácil de entender. Así sus valores fundamentales devueltos son:

- Si es una tecla de carácter, nos retorna el carácter. Por ejemplo: ***"a", "g", "k","ñ",etc.*** También nos indica si hay mayúsculas porque la letra la retornaría mayúscula: ***"A", "G", "K", "Ñ",*** etc.

- Con las teclas numéricas actúa igual, nos retorna el número. Pero con shift pulsada a la vez, nos devuelve el segundo símbolo de la letra (<&, $, %, etc.). Con Alt Gr nos retorna el tercero (|, <©, #, etc.).

- Actúa igual con cualquier tecla que escriba caracteres, simplemente indica qué carácter normalmente sale escrito.

- En el caso de pulsar, sin más, teclas de control, nos devolverá su nombre: ***"Control", "Alt", "Shift", "AltGraph", "F5", "F6", "KeyUp", "KeyDown", "Home", "End",*** etc.

A veces es necesario saber si se pulsó a la vez que la tecla, las teclas de control especial: **Ctrl, Alt** o **Shift.** Tenemos tres propiedades relacionadas con estas teclas que devolverán true si la tecla en cuestión se pulsó. Son: **AltKey, CtrlKey, ShiftKey** y **MetaKey** (tecla de los ordenadores Mac).

ACTIVIDAD 8.3: PRÁCTICA DE CAPTURA DE TECLAS

- Haz la Práctica 8.2 en la página 323 que permite practicar y entender cómo hacer captura de teclas.

Otra propiedad interesante es location que permite saber la localización de la tecla en el teclado. El caso habitual es tener que distinguir una tecla **Shift** de la otra. La propiedad **location** lo consigue, esta propiedad puede devolver los siguientes valores:

Devuelve un número entero que indica la posición del teclado en la que se encuentra la tecla. Valores posibles que devuelve:

VALOR	CONSTANTE RELACIONADA	SIGNIFICADO
0	DOM_KEY_LOCAT1ON_STANDARD	Teclado normal
1	**DOM_KEY_LOCATION_LEFT**	Zona izquierda (para distinguir las dos teclas control por ejemplo)
2	**DOM_KEY_LOCATION_R1GHT**	Zona derecha (para distinguir las dos teclas control por ejemplo)
3	**DOM_KEY_LOCATION_NUMPAD**	Teclado numérico

8.3.3.1 OBTENER LOS BOTONES DEL RATÓN

En los eventos del ratón hay propiedades comunes con las teclas. Por ejemplo, también tenemos las propiedades **AltKey, CtrlKey, ShiftKey** y **MetaKey,** para saber si la pulsación de alguna de estas teclas acompaña al evento de ratón.

Hay otras propiedades de coordenadas muy interesantes (además de las ya vistas). Así **movementX** y **movementY** nos retornan la diferencia en píxeles de las coordenadas X e Y respecto al último movimiento del ratón.

Además, la propiedad **button** devuelve el botón de ratón pulsado en el momento del evento. Los valores posibles para lo propiedad **button** son:

VALOR	SIGNIFICADO
0	Botón principal del ratón.
1	Botón central del ratón (en muchos casos, el botón de la rueda).
2	Botón secundario del ratón (en el caso de personas diestras, es el botón derecho).
3	Cuarto botón. En muchos ratones es el de retroceder página.
4	Quinto botón. En muchos ratones es el de avanzar página.

8.3.4 ANULAR COMPORTAMIENTOS PREDETERMINADOS

Además de propiedades, los objetos de evento poseen métodos. Uno de los más importantes es **preventDefault.** No tiene parámetros y lo que hace es conseguir que si ese evento producía en el elemento un comportamiento concreto por defecto, que ese comportamiento no se produzca.

Es más fácil de entender la idea con un ejemplo. Supongamos que hemos puesto un enlace y deseamos que el usuario confirme que desea ir al destino de ese enlace, de modo que si no lo confirma, no nos moveremos de la página actual. El código podría ser este:

```
<a href="https://es.wikipedia.org/">Ir a la Wikipedia </a>
<script>
        let enlace=document.querySelector("a");
        enlace.addEventListener("elick",function(ev){
                        if(confirm("¿Realmente desea abandonar esta página?"))
                        location="https://es.wikipedia.org/";
        });
</script>
```

Si ejecutamos este código en una página web, efectivamente se nos requiere confirmar la acción, pero aunque no aceptemos el cuadro, la acción se lleva a cabo, porque los enlaces, por defecto envían al usuario al destino. Por ello el código debería ser este:

```
<a href="https://es.wikipedia.org/">Ir a la Wikipedia </a>
<script>
        let enlace=document.querySelector("a");
        enlace.addEventListener("elick",function(ev){
            if(!confirm("¿Realmente desea abandonar esta página?"))
                    ev.preventDefault();
        });
</script>
```

8.3.5 CANCELAR PROPAGACIÓN

Anteriormente (8.2.3 ***"Propagación de eventos",*** en la página 302) hemos explicado cómo funciona la propagación de eventos. Hemos visto que los eventos se lanzan primero en el elemento más interior, luego en el siguiente y así, hasta que al final se lanzan los relacionados con los elementos más exteriores.

Podemos cancelar la propagación mediante un método llamado **stopPropagation.** Veamos este ejemplo:

```
<div>
        Pinta de rojo
        <button>Pinta de verde</button>
</div>
```

```
<script>
        let boton=document.querySelector("button");
        let div=document.querySelector("div");
        boton.addEventListener("elick",(ev)=>{
                document.body.style.backgroundColor="green";
        });
        div.addEventListener("elick",(ev)=>{
                document.body.style.backgroundColor="red";
        });
</script>
```

El botón está dentro de un elemento de tipo div. Hacer clic en el elemento div (con el texto ***Pinta de rojo)*** provoca colorear todo el cuerpo de la página en color rojo. Hacer clic en el botón lo pinta de verde. Sin embargo, al hacer clic en el botón, la página también se pinta de rojo. Si pudiéramos ver a cámara lenta el proceso, veríamos primero el color verde y luego el rojo. Predomina el color rojo, porque el evento clic se propaga al elemento div , después de haber sido capturado por el botón.

Para evitar este efecto podemos el método **stopPropagation:**

```
<div>
        Pinta de rojo
        <button>Pinta de verde</button>
</div>
<script>
        let boton=document.querySelector("button");
        let div=document.querySelector("div");
        boton.addEventListener("elick",(ev)=>{
                document.body.style.backgroundColor="green";
                ev.stopPropagation();
        });
        div.addEventListener("click",(ev)=>{
                document.body.style.backgroundColor="red";
        });
</script>
```

Hay que observar que hemos indicado el fin de la propagación en el elemento div. De esa forma, este elemento se retira del lanzamiento del evento en la fase de burbuja y así solo se ejecuta el código en el clic del botón.

Por supuesto el funcionamiento también podíamos haberlo modificado si la ejecución del código la hubiéramos obligado a realizar en la fase de captura.

8.3.6 LANZAR EVENTOS

JavaScript permite lanzar eventos a voluntad. La idea se basa en crear objetos de evento propios y mediante el método **dispatchEvent** de los elementos, enviar el evento creado.

La creación del evento se basa en el objeto **Event** (o en sus descendientes más específicos como **MouseEvent** por ejemplo) el cual funciona indicando el tipo de evento en la creación. Por ejemplo:

```
let evento=new Event("elick");
```

Después lanzaríamos el evento a cualquier elemento que esté capturando ese evento. Un ejemplo de uso (poco práctico, pero fácil de entender) sería el que proporciona este código:

```
<P>
      <button id="botonl">Coloreo de rojo</button>
      <button id="boton2">Yo hago lo mismo</button>
</p>
<script>
let botonl=document.getElementById("botonl");
let boton2=document.getElementById("boton2");
botonl.addEventListener("click",()=>{
      document.body.style.backgroundColor="red";
});
boton2.addEventListener("click",()=>{
      let el=new Event("elick");
      botonl.dispatchEvent(el);
});
</script>
```

Aparecerán dos botones, ambos realizan la misma acción, porque el evento ***onclick*** del segundo botón realiza un clic sobre el primero.

8.4 LISTA DE TIPOS DE EVENTOS

En todos los ejemplos anteriores hemos utilizado el evento ***click.*** Sin embargo, hay muchos otros eventos que podemos capturar. De detallan los principales a continuación.

8.4.1 EVENTOS DE RATÓN

Los eventos de ratón los produce este dispositivo. Pero también los dispositivos táctiles pueden producir algunos de estos eventos. Por ejemplo, el evento ***click*** ocurre cuando el usuario pulsa y despulsa el botón principal del ratón sobre el objeto que posee el ***listener,*** pero también ocurre si el usuario, en un dispositivo táctil, golpea y suelta el dedo sobre dicho objeto.

Por otro lado, aunque los eventos se refieren al ratón, realmente valen para cualquier dispositivo apuntador capaz de mover el cursor por la pantalla.

La tabla completa de los eventos de ratón es la siguiente:

EVENTO	EXPLICACIÓN
click	El usuario hace clic sobre el botón principal del dispositivo apuntador. Esta acción exactamente consiste en bajar y subir rápidamente el botón principal del dispositivo apuntador (o dar un golpe con el dedo en un dispositivo táctil)
dblclick	El usuario golpea dos veces rápidas sobre el botón principal del dispositivo apuntador (o con el dedo en un dispositivo táctil)
mousedown	El usuario pulsa un botón del dispositivo apuntador. Se produce este evento justo cuando el usuario pulsa el botón, antes de que le suelte.
mouseup	Se produce cuando el usuario suelta el botón del dispositivo apuntador que había pulsado previamente.
mousenter	Se produce cuando el usuario mueve el cursor del dispositivo apuntador dentro del elemento que captura el evento. Solo se produce si el cursor estaba fuera del elemento y el usuario le acaba de mover dentro.
mouseleave	Se produce cuando el usuario mueve el cursor del dispositivo apuntador fuera del elemento que captura el evento. Solo se produce si el cursor estaba dentro del elemento y el usuario le acaba de mover fuera.
mousemove	Se produce cada vez que el usuario mueve el cursor dentro del elemento (estando previamente dentro).
mouseover	Se produce cuando el cursor apuntador entra dentro del elemento o de cualquiera de los hijos del elemento.
mouseout	Se produce cuando el cursor apuntador abandona el elemento o cualquiera de los hijos del elemento.
contextmenu	Se produce cuando el usuario acaba de pedir el menú de contexto. Normalmente ese menú aparece al pulsar el botón secundario del dispositivo apuntador. En dispositivos táctiles suele ocurrir cuando el usuario deja el dedo apretado sobre el elemento 2 o 3 segundos al menos.

ACTIVIDAD 8.4: CAPTURA DE EVENTOS DE RATÓN

- Es muy interesante hacer la Práctica 8.3 en la página 325, para comprender la mayoría de eventos de ratón y cómo distinguir los botones en la práctica.

8.4.2 EVENTOS DE TECLADO

EVENTO	EXPLICACIÓN
keypress	Se produce cuando un usuario pulsa y suelta una tecla.
keydown	Se produce justo cuando el usuario ha pulsado la tecla (antes de que la suelte).
keyup	Se produce cuando el usuario suelta la tecla.

ACTIVIDAD 8.5: CAPTURA DE EVENTOS DE TECLADO

- La Práctica 8.4 en la página 327, facilita mucho el aprendizaje para capturar eventos de teclado. Es muy recomendable hacerla.

8.4.3 EVENTOS DE MOVIMIENTO EN LA VENTANA

EVENTO	EXPLICACIÓN
scroll	Ocurre cuando se ha desplazado la ventana a través de las barras de desplazamiento o usando el dispositivo táctil.
resize	Evento de window que se produce cuando se cambia el tamaño de la ventana.

ACTIVIDAD 8.6: CAPTURA DE SCROLL

- Realizar la Práctica 8.5 en la página 328 para practicar con el evento scroll y las propiedades relacionadas.

8.4.4 EVENTOS SOBRE CARGA Y DESCARGA DE ELEMENTOS

EVENTO	EXPLICACIÓN
load	Se concluyó la carga del elemento. Es uno de los elementos más importantes a capturar para asegurar que el código siguiente funciona con la seguridad de que está cargado lo que necesitamos. Cuando se aplica al elemento **window,** se produce cuando todos los elementos del documento se han cargado.
DOMContentLoaded	Similar al evento ***load.*** Se produce cuando el documento HTML ha sido cargado. A diferencia de ***load,*** se dispara sin esperar a que se terminen de cargar las hojas de estilos, imágenes y elementos en segundo plano. En general, para JavaScript, es más conveniente este evento.
abort	Se produce cuando se anula la carga de un elemento.
error	Sucede si hubo un error en la carga.
progress	Se produce si la carga está en proceso.
readystatechange	Ocurre cuando se ha modificado el estado del atributo readystate, lo cual ocurre cuando se ha modificado el estado de carga y descarga.

ACTIVIDAD 8.7: PANEL DE CARGA

- La Práctica 8.6 en la página 330, es muy interesante para manejar el evento **load.** Se recomienda encarecidamente hacerla.

Concretamente, el evento **load** del objeto **window** es muy importante. Hay que tener en cuenta que desde JavaScript manejamos elementos del DOM continuamente. Además, JavaScript es un lenguaje asincrono, como se ha comentado en este libro varias veces, y eso puede implicar que el código JavaScript intente manipular un componente que aun no se ha cargado.

Por ello es muy habitual que todo el código JavaScript para manipular elementos del DOM, se coloque dentro de la función callback asociada al evento **load** del objeto **window.**

8.4.5 OTROS EVENTOS

Deliberadamente hemos obviado los eventos relacionados con los formularios. Debido a la importancia de los formularios en la creación de aplicaciones, hemos dedicado un apartado completo al final de esta unidad.

Pero aun hay más eventos. Los eventos de esta sección son más específicos, no son de uso habitual, pero nos pueden ser de utilidad en algunas aplicaciones:

8.4.5.1 EVENTOS SOBRE EL HISTORIAL

EVENTO	EXPLICACIÓN
popstate	Se produce si se cambia el historial.

8.4.5.2 EVENTOS RELACIONADOS CON LA REPRODUCCIÓN DE MEDIOS

EVENTO	EXPLICACIÓN
waiting	Sucede cuando el vídeo se ha detenido por falta de datos.
playing	El medio está listo para su reproducción después de que se detuviera por falta de datos u otras causas.
canplay	Ocurre cuando se detecta que un vídeo (u otro elemento multimedia) ya se puede reproducir (aunque no se haya cargado del todo).
canplaythough	Se produce si se estima que el vídeo ha cargado suficientes datos para poderse reproducir sin tener que esperar la llegada de más datos.
pause	El medio de reproducción se ha pausado.
play	Ocurre cuando el medio de reproducción se ha empezado a reproducir tras una pausa.
ended	Se produce si la reproducción del vídeo o audio se ha detenido, sea por haber llegado al final o porque no hay más datos disponibles.
loadeddata	Se produce si se ha cargado el frame actual (normalmente el primero).
suspend	Se lanza si se ha suspendido la carga del medio.
emptied	Sucede si se ha vaciado el medio por un nuevo intento de carga por parte del usuario o por otras razones.
stalled	Sucede ante un fallo en la carga, pero sin que se detenga la misma.

EVENTO	EXPLICACIÓN
seeking	Ocurre cuando se ha iniciado una labor de búsqueda en el medio.
seeked	Ocurre cuando se ha finalizado la labor de búsqueda.
loadedmetadata	Se han cargado los metadatos del medio
durationchange	Sucede si se modifica el atributo duration del medio.
timeupdate	Ocurre si se ha modificado el atributo currentTime del medio.
ratechange	Se produce cuando el ratio del vídeo se ha modificado.
volumechange	Se lanza si el volumen se ha modificado.

8.4.53 EVENTOS DE ARRASTRE

Están relacionados con la interfaz de arrastre que es parte de HTML 5. Para que un elemento pueda ser arrastrable debe tener el atributo **draggable** colocado con valor **true:**

```
<div id="capaArrastrable" draggable="true"> ¡Soy arrastrable!</div>
```

En este tipo de operaciones hay dos capas protagonistas: una capa arrastrable (la que realmente se arrastra) y un posible destino del arrastre.

La capa que se arrastra puede generar estos eventos:

EVENTO	EXPLICACIÓN
dragstart	Se produce cuando el usuario empieza a arrastrar el elemento.
drag	Ocurre, una vez iniciado el arrastre, cada vez que se sigue arrastrando el elemento (cada vez que lo movemos).
dragstop	Se produce cuando el arrastre finaliza.

Eventos que se produce en el elemento destino del arrastre.

EVENTO	EXPLICACIÓN
dragenter	Se produce cuando el elemento que se está arrastrando, entra en el elemento destino.
dragover	Ocurre cada vez que se continúa, tras haber entrado, arrastrando el elemento origen sobre el destino.
dragleave	Ocurre cuando el elemento que se arrastra, sale del destino.
drop	Ocurre cuando el elemento origen se suelta dentro del destino. Para que este evento se pueda capturar hay que eliminar el comportamiento por defecto del evento **dragover.**

ACTIVIDAD 8.8: CAPTURA DE ARRASTRE

- En la Práctica 8.7 en la página 332, podemos observar cómo utilizar estos eventos para poder arrastrar una capa sobre otra.

8.4.5.4 EVENTOS SOBRE ANIMACIONES Y TRANSICIONES

Son eventos que podemos capturar cuando hemos programado animaciones sobre uno o más elementos desde CSS o por transiciones. Tenemos estos eventos posibles a capturar:

EVENTO	EXPLICACIÓN
animationstart	Se produce cuando se inicia una animación sobre el elemento.
animationinteraction	Se produce justo cuando se repite la animación.
animationend	Se produce cuando finaliza la animación.
transitionrun	Se lanza cuando ya se ha preparado para empezar la transición.
transitionstart	Ocurre cuando se inicia una transición.
transitionend	Ocurre al finalizar la transición.

8.4.5.5 EVENTOS DEL PORTAPAPELES

Permiten capturar los eventos de cortar, copiar y pegar. Todos ellos se relacionan con el objeto **clipboard** que es el que permite gestionar el portapapeles del sistema.

EVENTO	EXPLICACIÓN
cut	Se produce cuando el usuario intenta cortar contenido del elemento.
copy	Se produce cuando el usuario intenta copiar contenido del elemento.
paste	Se produce cuando el usuario intenta pegar contenido.

8.4.5.Ó EVENTOS ESPECIALES

EVENTO	EXPLICACIÓN
offline	Solo funciona para el objeto window ***y se*** produce si el navegador se desconecta de la red.
online	Solo funciona para window y se produce si el navegador vuelve a conectarse a la red después de haber estado desconectado.
fullscreenchange	Ocurre cuando un elemento pasa a modo de pantalla completa.
fullscreenerror	Sucede si hay un error al pasar un elemento a modo de pantalla completa.
message	Evento que se asocia a numerosos elementos de envío de mensajes como los que se producen a través de las APIs **WebSockets, WebWorkers** y otras.

8.5 FORMULARIOS

8.5.1 EL FORMULARIO COMO OBJETO DEL DOM

Indudablemente, los formularios son el componente fundamental de captura de eventos en una aplicación web. Nos han sido muy útiles los métodos **alert, prompt** y **confirm,** pero la realidad es que las aplicaciones profesionales no los utilizan nunca. Si hay que mostrar mensajes se

hacen en elementos HTML normales (paneles, capas, etc.) y la introducción de datos por parte del usuario se hace siempre con formularios.

Los formularios son la base de la comunicación de las aplicaciones web con el usuario. El objeto **document** dispone de una propiedad llamada **forms** que retorna una colección con todos los formularios del documento. Así **document.formsfO]** hará referencia al primer formulario del documento. Pero es posible, y más recomendable de hecho, acceder al formulario mediante otros métodos, por ejemplo, por su identificador.

Acceder a los elementos del formulario se hacía, de forma clásica, mediante el atributo **ñame** de los controles del formulario. Este atributo es el que permite dar nombre a la información que se graba con el control y que se enviará cuando se pulse el botón de envío **(submit)** del formulario. Así, si tenemos un control para grabar el primer apellido del usuario y ese control utiliza el atributo **ñame** para darle el nombre de **apellidol,** la forma de acceder a ese control es (suponemos que estamos en el primer formulario del documento):

```
document.forms[0].apel1idol;
```

Igualmente, podríamos hacerlo así:

```
document.forms[0]["apel1idol"];
```

Y, por supuesto, esta otra forma es válida:

```
consolé.log(document.querySelector("[name='apel1idol']"));
```

Si hay varios elementos con el mismo nombre (como ocurre con los botones de radio), esta última forma de acceder debe utilizar **querySelectorAll,** la cual devolverá una colección con todos los botones de radio con ese nombre. Después, tendremos que ir recorriendo cada botón de la colección.

RECOMENDACIÓN

- Damos por hecho que se conocen los entresijos de los formularios HTML tales como: tipos de controles, envío de datos por GET y por POST, atributos de los controles (especialmente **valué, ñame** e **id),** botones **submit** y **reset,** etc. De no ser así, se aconseja un repaso de estos conceptos para aprovechar mejor este apartado.

El objeto de formulario **(form)** posee estas propiedades y métodos:

PROPIEDAD O MÉTODO	USO
elements	Devuelve una colección que contiene todos los controles del formulario.
length	Obtiene el número de controles del formulario.
action	Devuelve el contenido de la propiedad action, que marca la URL destino de los datos del formulario. Permite su modificación.

PROPIEDAD O MÉTODO	USO
method	Retorna el contenido de la propiedad method del formulario, que marca la forma de envío de los datos (get o post). Permite su modificación.
enctype	Obtiene o cambia la forma de codificar los datos del formulario.
acceptCharset	Obtiene o modifica el conjunto de caracteres del formulario.
submit()	Envía los datos del formulario a su destino.
resetO	Deja los valores de los controles del formulario a su estado por defecto.

Esos métodos van a permitir automatizar acciones de forma muy efectiva en los formularios.

8.5.2 EVENTOS DE FORMULARIO

8.5.2.1 EVENTO SUBMIT

Se trata del evento que produce el envío de los datos del formulario a su destino. Es muy habitual validar los datos previamente a su envío para que el usuario detecte los fallos antes de realizar el envío. Este evento se produce justo antes del envío y se asocia al propio formulario (también se puede capturar en el botón de tipo **submit** del formulario). Tras su captura podemos validar los datos, pero teniendo la precaución de que si no cancelamos la acción por defecto de este evento (mediante el **preventDefault** del objeto de evento), la acción se llevará a cabo.

Como ejemplo, supongamos que tenemos un formulario en el que pedimos al usuario su nombre de usuario y que dicho nombre solo pueden ser letras del alfabeto inglés y con un mínimo de 6 caracteres. No enviaremos datos si no se cumple esa premisa y cuando detectemos el fallo, avisaremos del error y no permitiremos el envío. El código podría ser éste:

```
<form action="https://jorgesanchez.net/servicios/ver-datos.php">
        <input type="text" name="usuario" id="usuario"><br>
        <button type="submit">Enviar</button>
</form>
<div id="mensaje"> </div>
<script>
let formulario=document.forms[0];
let tUsuario=document.getElementById("usuario");
let cMensaje=document.getElementById("mensaje");

formulario.addEventListener("submit",function(ev){
        let exp=/^[a-zA-Z]{6,}$/;
        i f(exp.test(tUsuario.va lúe)==false){
                ev.preventDefault();
                mensaje.innerHTML="<p>Error: nombre de usuario no válido</p>";
        }
});
```

```
</script>
```

La captura del evento permite detener el envío de datos, si no se cumple la expresión regular en el valor del cuadro de texto en el que se escribe el nombre de usuario.

8.5.2.2 EVENTO RESET

Este evento se produce cuando se ha ordenado, a través de un botón de tipo **reset** por ejemplo, que el formulario muestre los valores iniciales y borre lo que el usuario hubiera escrito. La captura del evento nos permitirá matizar esta operación o añadir acciones a la misma.

8.5.23 EVENTOS DE ENFOQUE

Son eventos asociados a los controles de formulario en los que el usuario puede escribir o elegir opciones. Un control obtiene el ***foco*** cuando al usar el teclado manejamos dicho control. El enfoque se consigue gracias a la interacción con el dispositivo apuntador (un clic de ratón por ejemplo) o al cambio de control de formulario gracias a la tecla del tabulador.

Un control de formulario lanza el evento **focus** cuando obtiene el foco y lanza **blur** cuando pierde el foco.

8.5.2.4 EVENTO DE CAMBIO DE VALOR

El evento **change** es, seguramente, el evento más importante de los formularios. Se produce cuando modificamos el valor de cualquier control de formulario. Ejemplos de ello son:

- Cambiar el estado de un botón de radio o de tipo ***check.***
- Modificar el valor de un cuadro de texto o de contraseña.
- Elegir otro valor de una lista de opciones.
- Arrastrar un deslizador para modificar su valor.

Realmente, el evento se produce cuando se ha modificado el valor y, además, se ha perdido el foco sobre ese control. Es decir, durante el cambio no se lanza el evento.

8.5.3 PROPIEDADES DE LOS CONTROLES

Para modificar las propiedades de los controles de formulario podemos trabajar sobre sus atributos como hacemos con cualquier otro elemento. Por ejemplo, para obtener el valor de un control de formulario podemos hacer lo siguiente:

```
consolé.log(control.getAttribute("valué"));
```

Suponiendfo que ***control*** es el nombre de un control de formulario, efectivamente aparece el valor. Pero, el valor es algo que se cambia de forma dinámica en los formularios. Si el usuario ha modificado, por ejemplo, el valor de un cuadro de texto porque ha escrito algo en él, ***getAttribute("value")*** no refleja estos cambios. Por ello es mejor usar propiedades que sí reflejen los cambios que hace el usuario. Estas propiedades son:

PROPIEDAD	USO	CONTROLES QUE LA USAN
ñame	Nombre del control.	Prácticamente todos.
type	Valor de atributo **type.**	Controles de tipo **input.**
valué	Devuelve el valor actual del control.	Prácticamente todos.
checked	Puede valer true o false. Índica, sobre un control de activar o desactivar, si el control está activado o no.	Botones de radio v botones de tipo **check.**
defaultChecked	Indica el estado inicial de la propiedad **checked** en el control.	Botones de **radio** y botones de tipo **check.**
disabled	Índica, con **true** o **false,** si el control está deshabilitado o no.	Prácticamente todos.
readonly	Índica, con **true.o false,** si el control es de solo lectura o no.	Prácticamente todos.
required	Indica con **true** o **false** si es obligatorio o no cambiar el valor del control.	Controles de entrada de texto o listas.
maxLength	Permite ver y modificar la propiedad que define la anchura máxima de texto.	Controles de entrada de texto.
min	Valor mínimo posible para el control.	Input de tipo numérico o de fecha.
max	Valor máximo posible para el control.	Input de tipo numérico o de fecha.
step	Mínimo valor de cambio del control. Si el valor es 1, los valores avanzan de uno en uno como poco.	Input de tipo numérico o de fecha.
selectionStart	En controles de entrada de texto indica el índice dentro del texto general en el que comienza la selección actual sobre el texto. Si no hay nada seleccionado, indica la posición del cursor en el texto.  El cuadro de texto con selección que se muestra en la imagen anterior tendría estos valores en las propiedades: ■ **seleccionStart:** 6 ■ **seleccionEnd:** 14	Controles de entrada de texto.

PROPIEDAD	USO	CONTROLES QUE LA USAN
selectionEnd	En controles de entrada de texto indica el índice dentro del texto general en el que finaliza la selección actual sobre el texto. Si no hay nada seleccionado, indica la posición del cursor en el texto.	Controles de entrada de texto

8.5.4 MÉTODOS DE LOS CONTROLES

Los controles también ofrecen métodos con los que realizar acciones que simulan la acción del usuario. La lista es la siguiente:

MÉTODO	USO	CONTROLES
focus()	Fuerza a que el control obtenga el foco.	Prácticamente todos.
blur()	Provoca la pérdida de foco en el control.	Prácticamente todos.
selectf)	Selecciona todo el texto en el control y también deja el foco en él.	Controles de entrada de texto.
setSelectionRange(inicio,fin)	Selecciona el texto que va desde la posición inicio hasta la posición fin (sin incluir esta última).	Controles de entrada de texto.

8.6 PRÁCTICAS SOLUCIONADAS

Práctica 8.1: Cartel que persigue al ratón

- Crea una página web con bastantes párrafos.
- Haz que en cada movimiento de ratón sobre la página, un pequeño cartel con fondo amarillo y el texto Hola, siga al cursor del usuario o usuaria.

SOLUCIÓN: PRÁCTICA 8.1

Podemos empezar creando una serie de párrafos de muestra que cubren mucho contenido. Por ejemplo, 50 párrafos con el texto de prueba ***Lorem ipsum...*** Esto en Visual Studio Code lo podemos conseguir escribiendo dentro del apartado **body:**

```
p*50>lorem
```

Después pulsamos inmediatamente la tecla tabulador.

Además, crearemos una capa con posicionamiento de tipo fixed (y por lo tanto inmune al desplazamiento) y que tendrá un tamaño adecuado para el texto ***Hola.*** Esa capa se identificará como ***cHola.*** Un resumen del código HTML y CSS sería este:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Cartel perseguidor</title>
    <style>
        #cHola{
            background-color: yellow;
            color:black;
            position:fixed;
            left:0;
            top:0;
            padding:4px;
            width:100px;
            height:20px;
            text-align: center;
        }
    </style>
</head>
<body>
<div id="cHola">Hola</div>
<p>Lorem ipsum dolor sit amet consectetur, adipisicing elit. Sunt debitis esse corrupti autem repellat. Praesentium similique, excepturi repudiandae facilis dignissimos, cupiditate unde cumque voluptatum sit fugit
```

```
corporis tempore. Aliquam, amet. Aspernatur ipsum in nostrum praesen-
tium reprehenderit odit maiores iure nesciunt animi amet soluta quam
quis tenetur dolores, similique fugit sed molestias temporibus facere!
Dignissimos ea omnis ipsum libero amet dolores.</p>
 <p>Esse, in impedit optio facilis repellendus sed quia voluptates itaque,
sint, voluptatum eos quo ducimus? Accusantium aspernatur nesciunt in quas
blanditiis recusandae, quam magnam voluptatum ipsam pariatur officiis
corporis similique laboriosam ex. Ipsa explicabo obcaecati totam tenetur
cum, facilis blanditiis ipsam aliquid quidem commodi ullam, corrupti fuga
necessitatibus soluta porro quas cupiditate nam accusamus deleniti in
eius eaque? Molestias, quis.</p>
. . .
<p>Accusantium labore ut esse nam ullam iusto aut illum quidem laborum.
Beatae, ea. Quibusdam quod quo voluptatum in alias, dolor consequuntur
nam nobis, est id aliquid amet beatae quas dolorem cumque velit sapiente
iure quos ipsa? Libero cum i lio nam optio saepe. Vel necessitatibus repu-
diandae distinctio. Incidunt, ut soluta harum exercitationem quos qui vel
illum non eveniet porro esse saepe?</p>
<script src="acción.js"></script>
</body>
</html>
```

No se han colocado todos los párrafos para evitar ocupar mucho espacio, sin que apenas aporte demasiado la información. Se invoca al archivo JavaScript **accion.js,** el cual contiene este código:

```
document.body.addEventListener("mousemove",function(ev){
        let cHola=document.getElementById("cHola");
        cHola.style.left=ev.elientX+"px";
        cHola.style.top=ev.elientY+"px";
});
```

Se captura el evento **mousemove** que se produce cada vez que el usuario mueve el ratón, en este caso es el evento ideal a capturar. Lo único que hacemos es tomar las coordenadas clientX y clientY del evento y dárselas a las propiedades **left** y **top** de la capa, añadiendo la palabra ***px*** para indicar la unidad a CSS.

Práctica 8.2: Tecla que pone imagen de fondo

- Crea una página web que tenga un texto que indique que al pulsar Alt+F12, podremos colocar una imagen de fondo. El texto tiene que salir centrado.
- La idea es que inicialmente aparezca una pantalla negra con el texto y hasta que el usuario no pulse esa tecla, la imagen no se muestre.



Figura 8.2: Imagen de la aplicación antes de pulsar la combinación de teclas Alt+F12

- Tras pulsar Alt+F12 una imagen ocupará el fondo completo. Se aconseja usar una imagen aleatoria del servicio gratuito ***unsplash*** (https://source.unsplash.com/random).



Figura 8.3: Ejemplo de apariencia de la aplicación tras pulsar la combinación de teclas Alt+F12

SOLUCIÓN: PRÁCTICA 8.2

Curiosamente, lo más difícil de esta práctica es el formato CSS. Porque el código JavaScript es muy sencillo. Se presenta el código HTML que incluye tanto el CSS como el JavaScript de esta práctica:

```
<!DOCTYPE html>
<html lang="es">
<head>
        cmeta charset="UTF-8">
        <title>Imagen tras tecla</title>
        <style>
                #imagen{
                        background-color: black;
                        position:fixed;
                        left:0;
                        top:0;
                        width:100%;
                        height:100%;
```

```
            display:flex;
            justify-content: center;
            align-items: center;
            background-size:cover;
            background-position: center;

        }
        hl{
            color:white;
            text-align: center;
        }
</style>
</head>
<body>
    <div id="imagen”>
        <hl>Pulsa <kbd>Alt + F12</kbd> para cambiar la imagen de fondo</hl>
    </div>
    <script>
        document.body.addEventListener("keyup",function(ev){
            if(ev.altKey && ev.key=="F12"){
                let capa=document.getElementById("imagen");
                capa.style.backgroundImage=
                                        "url('https://source.unsplash.com/random')";
            }
        })
    </script>
</body>
</html>
```

Práctica 8.3: Capa *rollup*

- Crear una aplicación web que muestre una capa centrada que ocupe el 50% del ancho y el alto de la ventana.
- Inicialmente la capa será blanca y solo se la verá el borde.
- Al arrimar el ratón se colorea de verde.
- Al hacer clic encima con el botón principal se colorea de rojo, pero solo mientras el botón principal está abajo, si está arriba, entonces se quita el color rojo.
- Al hacer clic encima con el botón secundario ocurre lo mismo que en el caso anterior, pero se muestra el color azul. No se mostrará en ningún caso el menú de contexto.

SOLUCIÓN: FIGURA 8.3

En este caso hemos dividido el código CSS, elñ HTML y el JavaScript. El código HTML simplemente tiene, en el cuerpo, una capa que se identifica precisamente como ***capa.***

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="UTF-8">
      <title>Control de ratón</title>
      <1ink rel="stylesheet" href="estilos.css">
</head>
<body>
    <!DOCTYPE html>
    <html lang="es">
    <head>
        <meta charset="UTF-8">
        <title>cambio de color</title>
    </head>
    <body>
        <div id="capa"></div>
    </body>
    <script          src="acción.js"></script>
    </html>
</body>
</html>
```

Se destaca la carga de la hoja de estilos, la capa y la carga del archivo JavaScript. El archivo **estilos.css** podría tener este contenido:

```
#capa{
      position:fixed;
      width:50%;
      height:50%;
      left:25%;
      top:25%;
      border:lpx solid black;
}
```

Simplemente coloca la capa con el tamaño adecuado en el centro.

Finalmente el código JavaScript del archivo accion.js se encargará de capturar cada evento de ratón que nos interesa, además de utilizar la propiedad **button** del evento para distinguir los dos botones del ratón, y finalmente de anular el comportamiento del evento **contextmenu** para que no aparezca el menú del botón derecho,

```
var capa=document.getElementById("capa");

//El ratón entra en la capa
capa.addEventListener("mouseenter",(ev)=>{
      capa.style.backgroundColor="green";
});
```

```
//El  ratón sale de la capa
capa .addEventListener("mouseleave",(ev)=>{
      capa.style.backgroundColor="transparent";
});
//Se  aprieta el botón principal sobre la capa
capa .addEventListener("mousedown",(ev)=>{
      i f(ev.button==0)
            capa.style.backgroundColor="red";
      else if(ev.button==2)
            capa.style.backgroundColor="blue";
});
//Se  suelta el botón principal sobre la capa
capa .addEventListener("mouseup",(ev)=>{
      capa.style.backgroundColor="green";
});
//Se intenta sacar el menú de contexto, acción que se provoca
//□on el botón secundario, pero que se debe anular aparte
capa.addEventListener("contextmenu",(ev)=>{
      ev.preventDefault();
});
```

Práctica 8.4: Control de velocidad

- Crea una aplicación web que muestre el texto: ***Velocidad 0.***
- Si pulsamos la tecla de la flecha hacia arriba del teclado, la velocidad se incremente en uno.
- Si es la flecha hacia abajo, la velocidad baja en uno.
- La velocidad no puede sobrepasar los 120 ni descender de cero.
- Experimenta con los eventos de teclado para investigar cuál es el más apropiado en este caso.

SOLUCIÓN: PRÁCTICA 8.4

Se trata de una práctica muy sencilla. Es importante, para aprender la diferencia, hacer otras soluciones sustituyendo el evento **keydown** (que es el aconsejable en este caso) por los eventos **keyup** y **keypress.**

Este podría ser el código:

```
<!DOCTYPE html>
chtml lang="es">
<head>
      <meta charset="UTF-8">
      <title>Control de velocidad</title>
      <style>
            p{
```

```
                font-size:2em;
            }
        </style>
</head>
<body>
        <p>Velocidad <span>0</span> </p>
        <script>
            const TOPE_SUPERIOR=120;
            const TOPE_INFERIOR=0;
            let velocidad=document.querySelector("span");
            document.body.addEventListener("keydown",(ev)=>{
                let v=Number(velocidad.textContent);
                if(ev.key=="ArrowUp"){
                    if(v<TOPE_SUPERIOR)
                        velocidad.textContent = v + 1;
                }
                else if(ev.key=="ArrowDown"){
                    if(v>TOPE_INFERIOR)
                        velocidad.textContent = v - 1;
                }
            });
        </script>
</body>
</html>
```

Práctica 8.5: Cierre de panel con scroll

- Crea una capa que ocupe el 50% del ancho y el alto de la ventana y que en ella se muestre una gran cantidad de texto a la que podemos acceder mediante las barras de desplazamiento.
- Cuando hayamos conseguido llegar al final del desplazamiento, se mostrará un botón con el texto ***"Cerrar".*** Haciendo clic en él, se elimina la capa anterior.

SOLUCIÓN: PRÁCTICA 8.5

Se trata de un ejemplo muy parecido al que utilizan muchas aplicaciones para mostrar las condiciones de uso del software. Al llegar al final de la lectura, se permite aceptar esas condiciones y seguir con la instalación. En este caso, al llegar al final del panel, simplemente cerramos el panel.

Empezamos creando el HTML que lo único que hace es cargar el CSS del archivo **estilos.css** y el javaScript **accion.js.** Además crea un primer panel con el nombre **capaScroll** y una segunda capa llamada capaBoton en la que se mostrará el botón cuando lleguemos al final del scroll. Hemos creado el contenido de forma automatizada, como viene siendo habitual en muchas otras prácticas.

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
        <meta charset="UTF-8">
        <title>Scrol1</title>
        <1ink rel="stylesheet" href="estilos.css">
</head>
<body>
        <div id="capa">
                <p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
Fuga ipsa delectus pariatur, quasi nobis eos perspiciatis harum asperna-
tur assumenda porro animi velit, eaque debitis laborum sit? Dignissimos
reprehenderit corrupti deserunt?</p>
                <p>Reprehenderit amet omnis inventore quod, reiciendis moles-
tias impedit quo rerum? Incidunt consequuntur sit laboriosam esse commodi
atque soluta, fugit, tempore possimus illo labore omnis nihil ex quibus-
dam nesciunt cum animi!</p>
                <p>Expedita quam ab harum. Eius tempore dolore ducimus commodi,
porro quis accusantium expedita dolores in molestias assumenda nobis
ipsum possimus sapiente! Vitae dolorum debitis beatae dolore nesciunt
quasi doloribus porro?</p>
. . .
                <p>Consectetur qui voluptate placeat dolores quo illum quia num-
quam consequuntur laudantium saepe, quas dolorum laboriosam eum quidem
vero accusamus quisquam aperiam, vitae inventore. Dolorum quisquam obcae-
cati eum nostrum, sit suscipit.</p>
        </div>
        <div id="boton">
                <button>Cerrar</button>
        </div>
        <script src="acción.js"></script>
</body>
</html>
```

El código CSS oculta la capa del botón y deja el tamaño de la primera capa y la marca para que se pueda acceder a su contenido con barras de desplazamiento:

```
#capa{
        position:fixed;
        width:50%;
        height:50%;
        left:25%;
        top:25%;
        border:lpx solid black;
        overflow: scroll;
}
wbotonf
        position:fixed;
```

```
        width:100%;
        height:100px;
        bottom:0;
        left:0;
        display:none;
        text-align: center;
        background-color: black;
}
«boton buttonf
        height:50px;
        width:50%;
        margin-top:25px;
}
```

Finalmente, el código JavaScript es muy sencillo gracias a las propiedades de scroll del DOM. El evento permite obtener la coordenada de desplazamiento vertical **(scrollTop)** y de la propia capa la propiedad **scrollHeight** nos permite saber el tamaño completo, y **clientHeight** el tamaño de la capa en la ventana del usuario actual. La suma de scrollTop y clientHeight, cuando hemos llegado al final, tiene que llegar al tamaño de scrollHeight. Esa es la base del código.

```
let capaScroll=document.getElementById("capa");
let capaBoton=document.getElementById("boton");
let boton=document.querySelector("«boton button");
capa.addEventListener("scroll",(ev)=>{
        //para saber si hemos llegado al final
        //hay que sumar la anchura del elemento
        //a la propiedad scrollTop y ver si redondeando
        //hacia arriva, hemos llegado a scrollHeight
        if(Math.cei 1(capaScrol1.scrollTop+capaScrol1.elientHeight)> =
                        capaScrol1.scrol1Height){
                capaBoton.style.display="block";
        }
        else{
                capaBoton.style.display="none";
        }
});
boton.addEventListener("click",(ev)=>{
        document.body.removeChild(capaScrol1);
});
```

Práctica 8.6: Panel con el mensaje "cargando"

- Crea una página que descargue de Internet 10 imágenes de alta calidad
- Se mostrará un panel con el mensaje ***"Cargando..."*** durante la carga.
- El panel se retira automáticamente cuando se cargan todas las imágenes

SOLUCIÓN: PRÁCTICA 8.6

En el código HTML hemos utilizado varios servicios de Internet que permiten descargar imágenes de forma gratuita y aleatoria. Con esa carga, la página tarda un poco en mostrarse.

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8"        >
        <title>Panel de carga</title>
        <1ink rel="stylesheet" href="estilos.css">
</head>
<body>
        <div id="cargando">
                <hl>Cargando</hl>
        </div>
        < img src="http://lorempixel.com/1920/1048">
        <img src="https://source.unsplash.com/random">
        <img src="https://picsum.photos/1920">
        <img src="https://www.fi 1lmurray.com/640/360">
        <img src="https://loremf1ickr.com/1920/1048">
        <script src="acción.js"></script>
</body>
</html>
```

El código CSS permite colocar la capa tapando todo el contenido que se va cargando. Además, se crea una clase llamada ***ocultar*** que oculta la capa haciendo una animación que la hará ver cómo desciende de arriba a abajo, descubriendo el contenido de la página.

```
#cargando{
        background-color: black;
        position:fixed;
        left:0;
        top:0;
        width:100%;
        height:100%;
        display:flex;
        justi fy-content: center;
        align-items: center;
        background-size:cover;
        background-position: center;
        z-index:10;
}
.ocultar{
        top: 110% !important;
        transition:ls;
}
```

```
hl{
        color:white;
        text-align: center;
}
```

Lo más sencillo es el código JavaScript:

```
window.addEventListener("load",(ev)=>{
        let cargando=document.getElementById("cargando");
        cargando.classList.add("ocultar");
});
```

Práctica 8.7: Arrastre de una capa en otra

- Crea una aplicación que muestre dos capas del mismo tamaño concretamente 200 píxeles de ancho por 100 de alto.
- La primera capa mostrará el texto ***"Soy arrastrable"*** y tendrá fondo amarillo y la segunda muestra ***"Soy el destino" y*** tiene fondo blanco. Ambas dejan un borde de un píxel, negro.
- La aplicación permite arrastrar la primera sobre la segunda.
- Durante el arrastre, la primera capa se mostrará con una opacidad del 50%. Al arrastrar sobre la segunda, esta (el destino) se muestra con fondo rojo. Al soltar en esa segunda capa, la primera desaparece y en la segunda aparecerá el texto ***"¡Lo has logrado!"-***

SOLUCIÓN: PRÁCTICA 8.7

Esta práctica la vamos a resolver usando la interfaz de tipo ***drag and drop*** de HTML5. Esta interfaz se basa en eventos de arrastre, los cuales funcionan en elementos con el atributo **draggable** colocado en **true.**

El código HTML es muy sencillo:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Arrastre de una capa en otra</title>
        <1ink rel="stylesheet" href="estilos.css">
</head>
<body>
        <div id="capal" draggable="true" style="background-color: yellow">
                Soy arrastrable
        </div>
        <div id="capa2">Soy el destino</div>
        <script src="acción.Js"> </script>
</body>
</html>
```

El código CSS (archivo **estilos.css)** sería:

```
div{
        border:lpx solid black;
        padding:lem;
        position:fixed;
        top:20px;
        width:200px;
        height:100px;
}
ítcapal {
        left:50px;
}
#capa2{
        left:400px;
}
```

Finalmente, el código JavaScript **(accion.js):**

```
let                capal=document.getElementById("capal")
let               capa2=document.getElementById("capa2")
capal.addEventListener("drag",(ev)=>{
        capal.style.opacity=.5;
});
capal.addEventListener("dragend",(ev)=>{
        capal.style.opacity=l;
        capal.style.backgroundColor="transparent";
});
         capa2.addEventListener("dragenter",(ev)=>{
                capa2.style.backgroundColor="red";
});
capa2.addEventListener("dragleave",(ev)=>{
        capa2.style.backgroundColor="transparent";
});
         capa2.addEventListener("dragover",(ev)=>{
        ev.preventDefault();
});
capa2.addEventListener("drop",(ev)=>{
                document.body.removeChild(capal);
        capa2.textContent=" ¡Lo has logrado!";
        capa2.style.backgroundColor="yellow";
});
```

Es muy importante tener en cuenta qué eventos hay que escuchar en la capa de arrastre y cuáles en la capa destino. También es fundamental cancelar la acción predeterminada para el evento **dropover** en la capa de destino.

Práctica 8.8: Validación de formularios

• Crea una aplicación con esta apariencia inicial:



Figura8.4: Imagen inicial, ejemplo, de la Práctica 8.8

- La imagen de la izquierda se descargará de Internet (en páginas que permiten la descarga sin restricciones de derecho de uso). Realmente necesitamos tres: una relacionada con el área de distribución, otra con el área de oficina y una tercera para el área de producción.
- El cuadro combinado ***Tipo de incidencia*** contiene esas mismas tres entradas: distribución, oficina y producción. Eligiendo cada opción, se modifica automáticamente la imagen de la izquierda.
- Cuando pulsamos en el botón con el texto ***Mostrar descripción,*** aparece un área de texto en el que podemos describir la incidencia. Tras el botón la apariencia es esta:

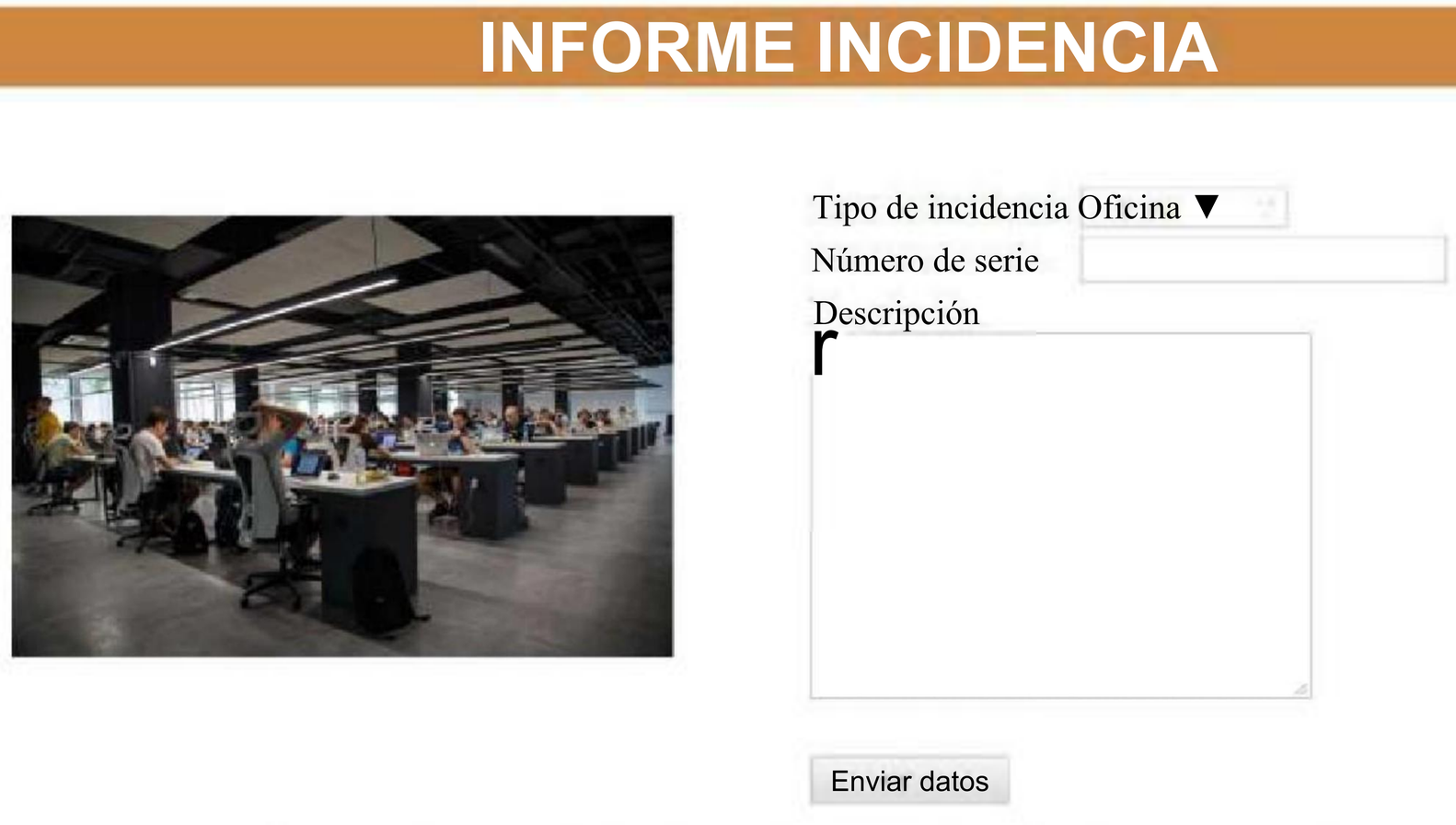


Figura 8.5: Imagen de la Práctica 8.8 tras pulsar el botón de mostrar descripción

- El número de serie debe de cumplir estas reglas:
  - Empezar con tres números
  - Seguir con 4 letras mayúsculas
  - Acabar con los números 1 o 2; o bien con la letra A
- Se valida el número de serie al pulsar enviar. Si es incorrecto, se avisará con un mensaje de error (fondo rojo y letra blanca). Además se marcará la etiqueta y el cuadro de texto con un borde fino de color rojo. Tanto el mensaje como los bordes, se quitan cuando el usuario se dispone a modificar de nuevo el número de serie.
- Los datos, realmente, no se enviarán a ningún destino.

SOLUCIÓN: PRÁCTICA 8.8

Esta práctica es un buen ejemplo de validación, donde el botón de enviar se controla para que no envíe nada si hay errores. Además, hay elementos de formulario que modifican elementos del DOM. Todo ello permite una práctica ya bastante real de cómo usar formularios en las aplicaciones web para comunicarse con el usuario.

Supones que hemos creado tres directorios: **img** para las imágenes, **css** para el código CSS y **js** para JavaScript.

Lo primero es ver el HTML. Es extenso debido a la maquetación en forma de tabla de la parte del formulario:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8"        >
        <title>Informe de Incidencias</title>
        <1ink rel="stylesheet" href="css/estilos.css">
</head>
<body>
<header>
        <hl>INFORME INCIDENCIA</hl>
</header>
< figure>
        <img src="img/distribucion.jpg" alt="oficina" id="imagen"/>
</figure>
<main>
        <form action="#">
                <table>
                        <tr>
                                <td>
                                        <label for="tipo">
                                                Tipo de incidencia
                                        </label>
```

```
                </td>
                <td>
                    <select name="tipo" id=”tipo">
                        <option value="distribución" selected>
                            Distribución
                        </option>
                        <option value="oficina">Oficina</option>
                        <option va 1ue="produce ion">Produce ion</optión>
                    </select>
                </td>
            </tr>
            <tr>
                <td>
                    <label id="1abelSerie" for="serie">
                        Número de serie
                    </label>
                </td>
                <td>
                    <input type="text" name="serie" id="serie">
                </td>
            </tr>
            <tr>
                <td colspan="2">
                    <button type="button" id="mostrarDescripcion">
                        Mostrar Descripción
                    </button>
                    <div id="fDescripcion">
                        <label for="descripción" id="labelDescripción">
                            Descripción
                        </label> <br>
                        <textarea name="descripción" id="descripción"
                                  cols="30" rows="10">
                        </textarea>
                    </div>
                </td>
            </tr>
            <tr>
                <td colspan="2">
                    <br>
                    <button id="enviar">Enviar datos</button>
                </td>
            </tr>
        </table>
        <div id="capaError">

        </div>
```

```
            </form>
</main>

<script src="js/accion.js"> </script>
</body>
</html>
```

La hoja CSS (estilos.css) es la siguiente:

```
hl{
    text-align: center;
    color:white;
    font-family:            sans-serif;
    background-color: perú;
}

figure{
        position:absolute;
        top:100px;
        left:0;
        width:300px;
}

figure img{
        width:100%;
}

main{
        position:absolute;
        top:100px;
        left:400px;
}
*fDescripcion{
        display:none;
}
#capaError{
        background-color:       red;
        color:white;
        margin-top:2em;
}

.error{
    border:1px solid red;
}
```

Finalmente, el código JavaScript **(accion.js):**

```
//Todo el código le colocamos de forma que tengamos
//asegurada toda la carga del DOM
window.addEventListener("load",function(ev){
        //asignación de elementos
        var imagen=document.getElementById("imagen");
        var combo=document.getElementById("tipo");
        var serie=document.getElementById("serie");
        var 1abelSerie=document.getElementById("labe1Serie");
        var                    labelDescripcion=document.getElementById("labe1Descripción");
        var enviar=document.getElementById("enviar");
        var          mostrarDescri          peion=document.getElementById("mostrarDescripcion");
        var capaError=document.getElementById("capaError");
        var fDescripcion=document.getElementById("fDescripcion");

        //Cuadro combinado que cambia la imagen
        combo.addEventListener("change",function(ev){
                imagen.setAttribute("src",'img/${combo.valuej.jpg');
                labelSerie.removeAttribute("class");
                serie.removeAttribute("class");
        });

        //Si se coloca el foco en el número de serie
        //quitamos los errores
        serie.addEventListener("focus",function(ev){
                capaError.textContent="";
                labelSerie.classList.remove("error");
                serie.classList.remove("error");
        });
        //Validación del número de serie al intentar enviar los datos
        enviar.addEventListener("click",function(ev){
                if(/A[0-9]{3}[A-Z]{4}[12A]$/.test(serie.value)==false){
                        ev.preventDefault();
                        capaError.textContent="Código no válido";
                        labelSerie.classList.add("error");
                        serie.classList.add("error")
                }
        });
        //Click en el botón de descripción, muestra una capa
        //con el área de texto y la etiqueta
        mostrarDescripcion.addEventListener("click",function(ev){
                fDescripcion.style.display="iniine-block";
                this.style.display="none";
        });
});
```

8.7 PRÁCTICAS RECOMENDADAS

Práctica 8.9: Repasar Modelos de Red

- Crea una aplicación web con esta apariencia aproximada:

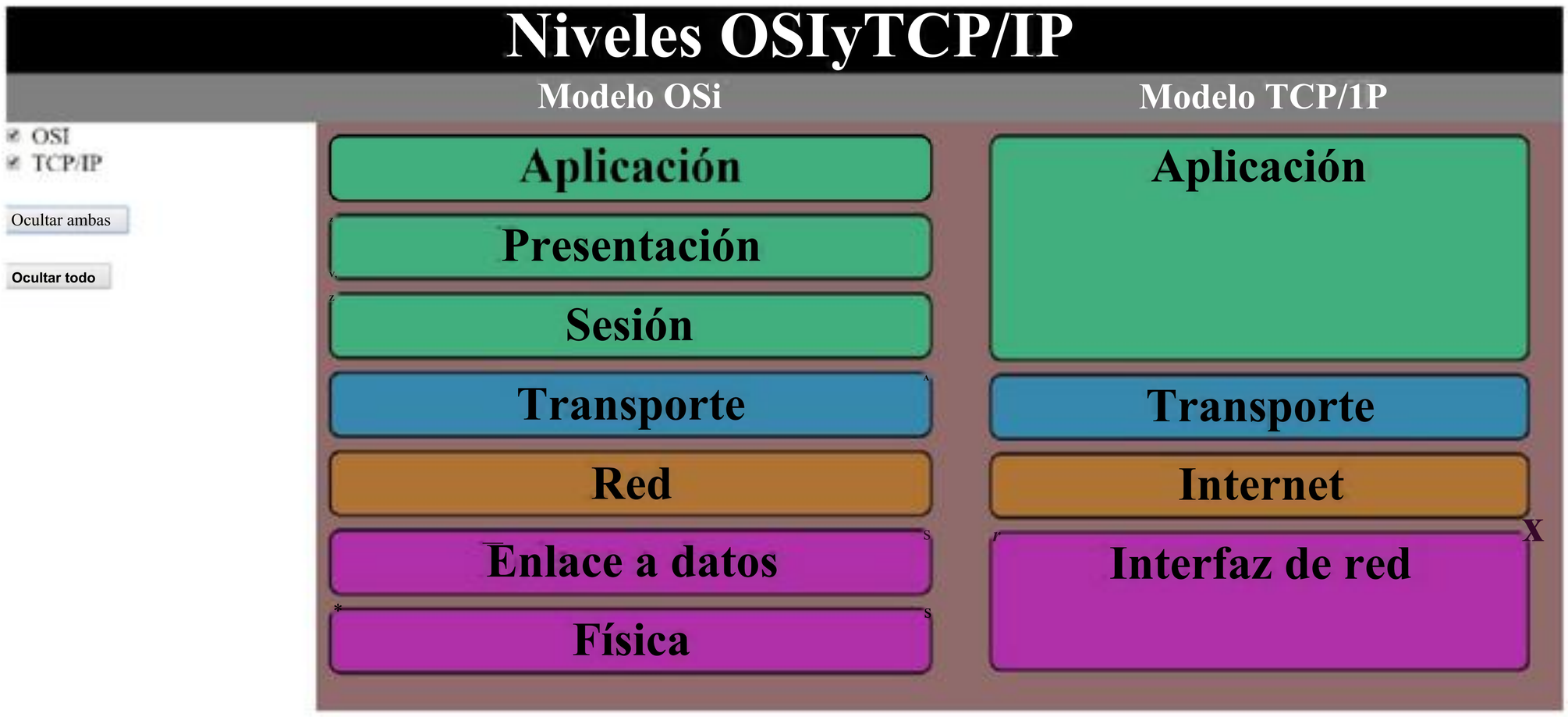


Figura 8.6: Aspecto de la aplicación con los dos modelos a la vista

- Tanto el **modelo OSI** como el **TCP/IP** salen cuando se activan las casillas de verificación correspondientes. Inicialmente tanto las casillas como los modelos están desactivados
- El botón **Ocultar ambas,** oculta ambos modelos y además desactiva las casillas.
- El botón **Ocultar todo** tapa toda la página con un velo negro. Ese negro sale desde arriba con una animación hasta que oculta toda la página. Tras aparecer y quedar todo en negro, podemos hacer clic en el velo y vuelve a su posición original (también mediante una animación).

Práctica 8.10: Lista de tareas

- Crea una aplicación web con la apariencia aproximada de la Figura 8.7.
- El botón "+" añade tareas, pero solo si hay contenido en el cuadro
- Cada botón con el texto **"quitar",** quita la tarea correspondiente. En cuanto se añade una tarea, este botón aparece en ella para poder realizar esta labor.
- Al hacer clic en una tarea (en el nombre de la misma), se quita de donde está, y pasa a ser la primera de la lista.
- El único tipo de letra utilizado es **Montserrat** (letra de **Google Font)** en grosores 400 y 700.
- Colores: **teal, brown** y **red.**

• Consigue que las tareas se vayan grabando en una cookie que dure una semana, así al entrar de nuevo, con el mismo navegador, veremos las tareas que habíamos añadido.
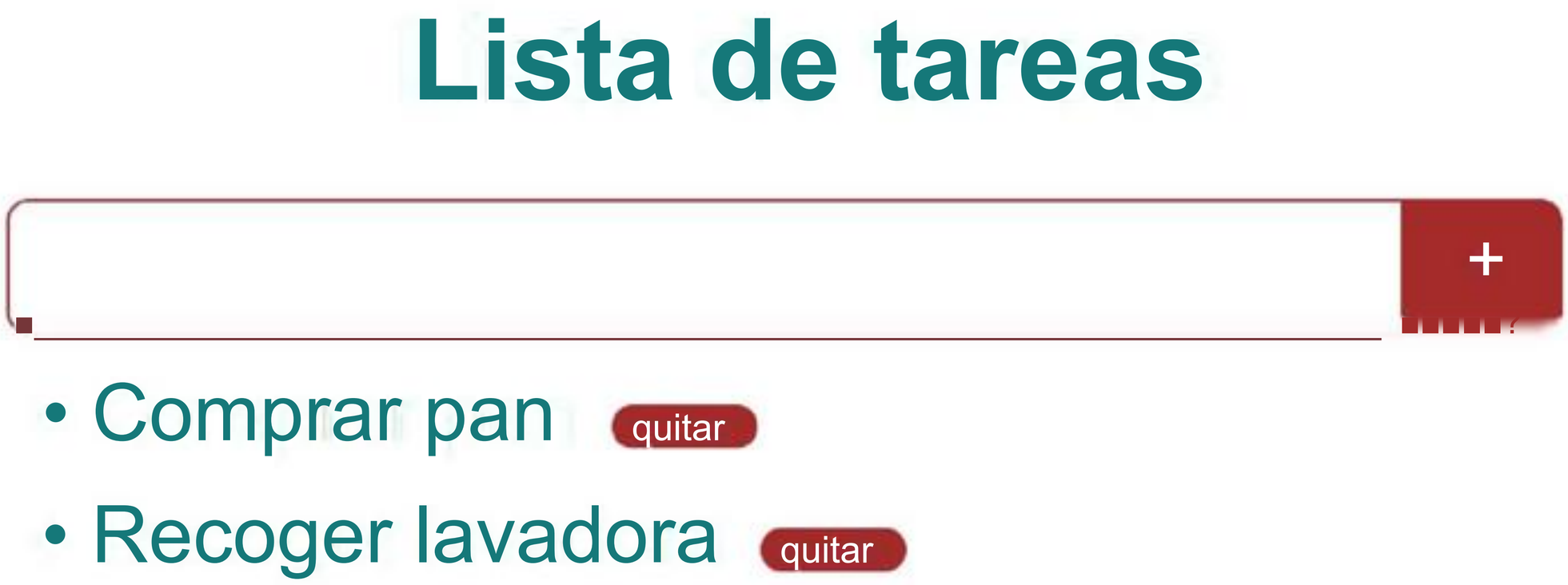


Figura 8.7: Aspecto de la aplicación con dos tareas añadidas

Práctica 8.11: Calculadora

• Crea una aplicación web con esta apariencia aproximada:

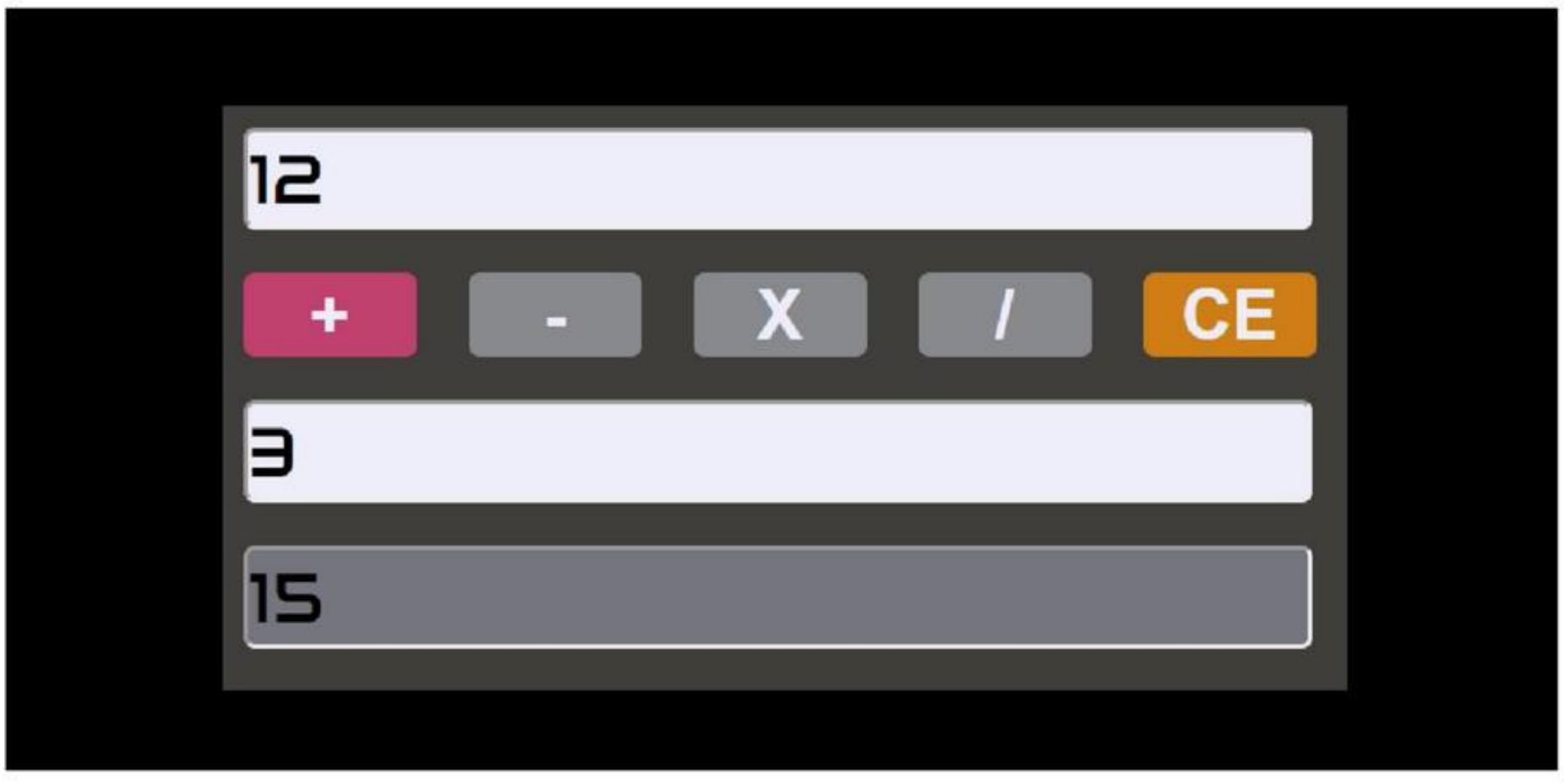


Figura 8.8: Aspecto de la aplicación tras sumar los números 12 y 3

• Se trata de una calculadora que debe de funcionar perfectamente con los botones que aparecen.

- La letra de los cuadros numéricos es de Google Fonts y se llama **Audiowide.**
- El cuadro debe de salir totalmente centrado en la pantalla independientemente de la resolución
- Los colores usados son:
  - negro
  - #3F3e3b (fondo calculadora)
  - #87898D (fondo botón)
  - #C0416E (fondo al arrimar el ratón)
  - #EDEEF9 (color letra botones y fondo cuadros numéricos)
  - # 74757e (fondo control de resultado)
- Hay que conseguir que no se pueda escribir en el cuadro de texto en el que aparece el resultado y que cuando se pulsen los botones correspondientes se calcule el resultado en base a los valores escritos.

Práctica 8.12: Buscaminas

- Consigue crear el juego del buscaminas. Básate en lo realizado en la Práctica 6.9 en la página 243 y en la Práctica 5.11 en la página 198.
- Ahora el tablero será interactivo, al hacer clic veremos si hay una mina o no. SI no la hay, se nos informará de cuántas minas tiene esa casilla alrededor.
- Si descubrimos todas las casillas sin mina, habremos ganado la partida

---

Práctica 8.13: Juego del helicóptero

- Consigue crear un juego en el que un helicóptero intenta posarse en una torre.
- Las imágenes para crear el juego están disponibles en https://github.com/jorgesancheznet/libro-dwec/tree/master/unidad-08/practica-08-13/img
- Una capa muestra una imagen de un helicóptero, otra una imagen de una plataforma de aterrizaje y otra un marcador con la velocidad. El fondo lo cubre completamente la imagen del cielo
- Las flechas permiten mover el helicóptero en la dirección deseada.
- Todos los movimientos hacen efecto de acelerar y frenar en horizontal y vertical. Es decir, si el helicóptero se mueve a la izquierda y va a 10, la flecha derecha no cambia de dirección, solo aminorará la marcha. Pero, si seguimos pulsando derecha llegaremos a velocidad 0 y a partir de ahí el helicóptero va hacia la derecha.

- Lo mismo vale para subir y bajar, si el helicóptero sube, la flecha de bajar aminora la marcha, pero si insistimos en bajar, la velocidad será negativa.
- Hay que contar con la fuerza de la gravedad que siempre tira hacia abajo. En vertical es como si continuamente alguien está pulsando la flecha hacia abajo,
- Si el helicóptero sobrepasa los límites de la pantalla o choca con la plataforma aparece la imagen **boom,** en la zona donde ha chocado.
- El helicóptero tiene que posarse despacio, si se posa a mucha velocidad ocurre el ¡boom!
- Cuando posa bien, el juego termina y se indica con el texto en grande **¡bravo!**

8.8 RESUMEN DE LA UNIDAD

- Los eventos son uno de los elementos más importantes de la programación de aplicaciones web.
- Gracias a los eventos podemos comunicar la aplicación con el usuario y también reaccionar ante situaciones que se producen de forma asincrona.
- La programación basada en eventos consiste en asignar el código de una función de tipo callback al evento concreto. Cada evento, además, se asocia a un elemento concreto del DOM.
- Hay varios métodos para programar eventos:
  - Usar atributos HTML como **onclick** por ejemplo.
  - Usar métodos de tipo onclick, onload, etc. Que son métodos de los elementos del DOM.
  - Usar el método **addEventListener** que es el más recomendado y que asocia el evento a una función callback de forma más abierta, además de poder asociar más de una función.
- Los eventos se propagan en dos fases: primero de los elementos más grandes a los más pequeños (fase de captura) y luego de los pequeños a los grandes (fase de burbuja). Por defecto, el código asociado al evento se lanza en la fase de burbuja, pero podemos cambiar este comportamiento.
- AI ocurrir un evento se crea un objeto especial que posee propiedades y métodos relacionadas con el evento. Estas propiedades y métodos dependen del tipo de evento, pero son detalles como las coordenadas del ratón, la tecla que hemos pulsado, la posibilidad de anular el comportamiento por defecto del evento, etc.
- Podemos incluso crear nuestros propios eventos y ejecutarlos para su captura igual que los demás eventos.
- Hay numerosos tipos de eventos, lo que nos permite usar este tipo de programación en prácticamente cualquier circunstancia.
- Los formularios alcanzan una nueva dimensión con el apoyo de los eventos. Hay eventos especialmente preparados para los formularios. Además, podemos modificar casi cualquier propiedad de los controles del formulario desde JavaScript, lo que convierte a los formularios en la vía prioritaria para comunicarse con el usuario.

8.9 TEST DE REPASO

1.- **¿Qué método es el prioritario para capturar eventos actualmente en JavaScript?**

a) Usar atributos asociados al evento de HTML.

b) Usar propiedades de los elementos con el del evento.

c) Usar el método **addEventListener.**

d) Usar el método **on.**

2.- Observa este código

```
capa.addEventListener("click",
  ()=>{
    consolé.log("Hola")
},true);
```

¿En qué fase se lanza el código asociado al evento?

a) En la de captura.

b) En la de burbuja.

c) En ambas.

d) En ninguna.

3.- **¿Para qué sirve el método "target" de los objetos de evento ?**

a) Para llegar a las propiedades del objeto de evento.

b) Para obtener el tipo de evento que se ha producido.

c) Para obtener el elemento en el que se ha producido el evento.

d) Para obtener el método de captura del evento.

4.- **Queremos distinguir si hemos pulsado el botón izquierdo o el derecho del ratón (posiciones de diestro) ¿Qué propiedad del objeto de evento lo permite?**

a) location

b) mouse

c) key

d) button

e) No se puede averiguar qué botón se ha pulsado

5.- **¿Qué método permite dejar de escuchar un evento?**

a) removeEventListener

b) addEventListener

c) preventDefault

d) cancelEvent

6.- **Queremos crear un nuevo evento de tipo "dblclick" ¿Qué instrucción lo permite?**

a) new Event("dblclick")

b) window.createEvent("dblclick")

c) elemen.createEvent("dblclick")

d) elemen.
addEventListener("dblclick")

7.- **¿Qué evento se produce antes?**

a) click

b) dblclick

c) mousedown

d) mouseup

8.- **¿Qué atributo hay que activar en los elementos que deseamos arrastrar mediante eventos *drag?***

a) allowDrag

b) useDrag

c) drag

d) draggable

9.- **¿Qué evento es el que debemos comprobar para saber si el usuario está situado en un control de formulario?**

a) click

b) keydown

c) blur

d) focus

10.- **¿Qué controles producen eventos de tipoo *change?***

a) ***input*** de tipo ***text***

b) ***select***

c) ***textarea***

d) Todos los anteriores

UNIDAD 9

CONCEPTOS AVANZADOS

OBJETIVOS

— Reconocer la importancia de la gestión de errores en las aplicaciones web

— Distinguir entre los conceptos de error, excepción y aviso

— Aplicar la estructura try..catch..finally para capturar los errores y gestionarlos adecuadamente

— Analizar el funcionamiento de los objetos de error

— Asimilar las ventajas del uso de módulos

— Aplicar la carga dinámica de módulos en aplicaciones web

— Distinguir las diferencias entre la programación síncrona y asincrona

— Identificar el problema del exceso de llamadas a funciones callback

— Reconocer el funcionamiento de las promesas como mecanismo de control de acciones asincronas

— Aplicar correctamente los métodos **then** y **catch** de las promesas

— Asimilar el funcionamiento de las funciones **async** y el operador **await**

CONTENIDOS

9.1 JAVASCRIPT AVANZADO

9.2 CONTROL DE ERRORES

9.2.1 INTRODUCCIÓN AL CONTROL DE ERRORES

9.2.2 ERRORES, EXCEPCIONES Y AVISOS

9.2.3 JAVASCRIPT MÁS ESTRICTO

9.2.4 CREAR Y LANZAR ERRORES PROPIOS

9.2.5 GESTIONARLAS EXCEPCIONES. BLOQUE TRY...CATCH

9.3 MÓDULOS

9.3.1 USO DE MÓDULOS Y PAQUETES

9.3.2 CARGA Y CREACIÓN DE MÓDULOS

9.4 PROGRAMACIÓN ASINCRONA

9.4.1 PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA

9.4.2 CALLBACK HELL

9.4.3 PROMESAS

9.4.4 FUNCIONES **ASYNC**

9.5 PRÁCTICAS RESUELTAS

9.6 RESUMEN DE LA UNIDAD

9.7 TEST DE REPASO

9.1 JAVASCRIPT AVANZADO

El lenguaje JavaScript ha mejorado enormemente su sintaxis para acomodarse a las necesidades planteadas por los desarrolladores de aplicaciones web. Esta unidad presenta algunos de los elementos más celebrados en las mejoras del lenguaje. Es una unidad teórica cuya aplicación se verá reforzada en el tema siguiente, cuando se describa la forma de realizar peticiones AJAX mediante la API Fetch.

Todo desarrollador debe de tener en cuenta que el lenguaje JavaScript es un lenguaje vivo que se sigue ampliando y mejorando continuamente. Las novedades tardan un tiempo en ser adoptadas al 100% por los navegadores, las que aquí se presentan ya han sido adoptadas por todos ellos, salvo los que, como Internet Explorer, no siguen actualizándose.

9.2 CONTROL DE ERRORES

9.2.1 INTRODUCCIÓN AL CONTROL DE ERRORES

Sin duda una de las tareas más importantes, a la hora de programar aplicaciones, es la de solucionar los errores que van apareciendo. Todos los profesionales del mundo del desarrollo han pasado horas y horas, al menos alguna vez, tratando de solventar un error que habían detectado en el programa. Lo peor, es que hay veces que los errores se detectan mucho después porque ocurren circunstancias en el uso de la aplicación que no se habían tenido en cuenta.

La habilidad de encontrar errores se va consiguiendo a través de los años de experiencia, pero hay que conocer los aspectos y las técnicas fundamentales desde el primer momento. Los errores que ocurren en un programa pueden ser:

- **Errores al escribir código por parte del programador.** Son **errores de sintaxis,** errores por cosas como: expresiones incorrectas al escribir el código, cierres de llaves olvidados, palabras clave mal escritas, etc. Son los más fáciles de detectar porque cuando se interpreta el código, se nos indica el error. En el caso de que el código se esté creando en un entorno de trabajo avanzado, hay errores que aparecen marcados en el mismo instante en el que hemos escrito el código. En el caso de las aplicaciones web, el error aparece también marcado en la consola del panel de depuración del navegador.

  En realidad hay dos tipos:

  - **Errores detectables en tiempo de escritura.** Son fallos de sintaxis evidentes que la mayoría de entornos de desarrollo (incluido **Visual Studio Code)** pueden marcar antes de probar el código. Muchas veces estos entornos lo que hacen es subrayar en rojo el código erróneo.
  - **Errores de ejecución.** Son errores que solo se pueden detectar cuando el código se intenta ejecutar para probar la aplicación. Un ejemplo de error de este tipo es cuando en el código se invoca a una función que aún no ha sido definida. Solamente cuando tratamos de ejecutar el código se puede detectar que esa función no existe,

- Hay errores por mala lógica al desarrollar la aplicación. Son **errores lógicos,** en los que la sintaxis es correcta, pero el programa no funciona como debería. Ninguna herramienta nos avisa automáticamente del error, aunque sí hay herramientas especiales que facilitan su detección, es el desarrollador el que detecta que la aplicación no funciona como debería. Puede ser también, que el error lo detecten los usuarios y se lo comuniquen a los desarrolladores.

- Hay errores por causas externas. Son **errores del sistema,** circunstancias que provocan el error pero que están fuera del control del programador: fallo en la conexión de red, caída de un servicio que estábamos utilizando, etc. No podemos controlar estos fallos la mayoría de veces, pero al menos sí podemos matizar el daño que causan a nuestra aplicación.

- **Errores de usuario.** Son los provocados por acciones inesperadas que realiza el usuario y que causan un error en tiempo de ejecución. Por ejemplo, pedir al usuario un número y recibir un texto. En realidad, son errores lógicos que ocurren por no prever estas situaciones.

9.2.2 ERRORES, EXCEPCIONES Y AVISOS

Estas tres palabras se refieren al control de errores, pero es muy importante diferenciarlas para saber controlar y decidir debidamente cómo actuar si ocurre cualquiera de ellas.

Un **error** es un fallo que produce el programa y que tiene como consecuencia que la aplicación se detenga. Los errores no están controlados y provocan todo tipo de situaciones indeseadas en la ejecución de la aplicación.

Una **excepción** es un error que podemos controlar para que se gestione adecuadamente. Las excepciones permiten manejar objetos especiales que contienen los detalles del error para poder lidiar con el mismo de la mejor manera posible.

Un **aviso** ***(warning)*** es un error que se considera leve. No impide la ejecución del programa pero, al menos, intenta avisar del problema al desarrollador o desarrolladora para que conozca la situación. Los avisos pueden ser vitales para detectar, de forma temprana, errores complejos de resolver.

9.2.3 JAVASCRIPT MÁS ESTRICTO

JavaScript se ideó bajo una capa de rapidez y dinamismo que hizo que este lenguaje no tuviera una sintaxis rígida. Las ventajas de este hecho son: la facilidad para empezar a desarrollar aplicaciones con este lenguaje y el dinamismo que se consigue en los resultados con poco esfuerzo de escritura de código.

El problema es que, a medida que las posibilidades de las aplicaciones JavaScript han aumentado, el control de errores se ha hecho cada vez más importante y el lenguaje original, en este sentido, no ayudaba mucho.

Un ejemplo de variante más estricta es el lenguaje creado por **Microsoft** con el nombre de **TypeScript.** Este lenguaje tiene una sintaxis más formal y estricta que es más del gusto de muchos desarrolladores, es famoso porque añade muchos tipos de datos al lenguaje y fuerza que se utilicen los elementos del lenguaje de forma más estricta, especialmente en todo lo referente al uso de tipos de datos.

TypeScript no tuvo un gran éxito hasta que se convirtió en el lenguaje base del framework **Angular** de **Google** que es ampliamente utilizado por una enorme comunidad de desarrolladores. TypeScript es más potente para detectar errores, pero es más pesado de escribir al ser mucho más rígido. Ningún navegador entiende TypeScript, por lo que su código se debe convertir a JavaScript con ayuda de un software especial.

No obstante, desde la versión ES5 del estándar, se puede activar en JavaScript el llamado ***modo estricto,*** que es un modo más exigente con los errores. Por ejemplo, si ejecutamos este código por consola:

```
x=9;
consolé.log(x);
```

Simplemente veremos el número 9 por consola. Si viéramos los avisos, aparecería un aviso indicando que la variable ***x*** no se ha definido. La razón es que no hemos declarado la variable ***x,*** pero no se considera un error, es un ***warning.*** El programa funcionará porque actúa con la variable como si realmente se hubiera declarado.

El modo estricto se activa escribiendo el término entrecomillado **'use strict'** en la primera línea del archivo. En ese caso, el código siguiente:

```
'use strict';
x=9;
consolé.log(x);
```

Ahora sí se produce un **error,** indicando que ***x*** no está definida. El modo estricto se puede activar para todo un archivo, como es el caso del ejemplo anterior, pero se puede activar para el código interior a funciones concretas:

```
function f(param){
   'use strict';
   ….
}
```

La función f usará el modo estricto en su código, fuera de la función no se usará este modo.

9.2.4 CREAR Y LANZAR ERRORES PROPIOS

Los errores se crean por parte del intérprete de JavaScript cuando ocurren. Pero podemos crear nuestros propios errores:

```
let miError=new Error("Se esperaba un número");
```

La variable ***miError*** es una referencia a un objeto que representa un error. El objeto ***Error*** representa errores genéricos, pero hay seis objetos que sirven para crear errores propios de tipo más específico.

TIPO DE ERROR	USO
EvalError	Error al intentar usar la función **eval()** de JavaScript.
RangeError	Error numérico 0 de rango de valores incorrecto.
ReferenceError	Referencia a objeto no válido.
TypeError	Error por un mal uso de los tipo de datos .
URI Error	Error al codificar 0 decodiñcar la URL.

En base a la tabla anterior sería más correcto:

```
let miError=new RangeError("Se esperaba un número");
```

Crear un objeto de error no provoca ningún error. Lanzar el error en sí y provocar una excepción, se hace con la palabra clave **throw:**

```
let miError=new RangeError("Se esperaba un número");
throw miError;
```

Se provoca un error que podrá ser analizado por la consola.

9.2.5 GESTIONAR LAS EXCEPCIONES. BLOQUE TRY...CATCH

JavaScript aporta una estructura llamada **try..catch.** Esta estructura trabaja con esta sintaxis:

```
try{
    …
    código que puede provocar un error
    …
}catch(objetoError){
    …
    código que se ejecuta si hay error
    …
}
```

En el bloque encabezado por la palabra **try** se coloca el código que puede provocar un error. Si ese error se produce, el flujo del programa pasa al apartado catch, dejando el resto de líneas del try posteriores a la que produce el error, sin ejecutar.

Veamos este sencillo ejemplo:

```
try{
    consolé.log(e);
    consolé.log("aquí");
}
```

```
catch(error){
        let e=l;
        consolé.log(e);
}
```

Si el archivo solo tiene este código, se va a intentar escribir por consola el valor de una variable llamada ***e*** que no existe porque no se ha declarado. Con lo cual el código **console.log(e)** produciría un error. Al estar en un bloque **try,** se crea el objeto de error y se envía dicho objeto al apartado **catch.** Las líneas dentro del catch pasan a ejecutarse: se declara la variable **e** con valor **1** y se escribe por pantalla. Veremos simplemente, al ejecutar el código, que se muestra el número 1 por consola.

Todo cambia si ***e*** se declara antes de la instrucción ***try:***

```
let e=12;
try{
   console.log(e);
   console.log("aquí");
}
catch(error){
    let e=1;
    console.log(e);
}
```

Se muestra por pantalla:

```
12
aquí
```

No se muestra el código del catch, no hay error.

Hay posibilidad de añadir un bloque **finally** cuyo contenido se muestra tanto si se produce la excepción, como si no:

```
try{
      consolé.log(e);
      consolé.log("aquí");
}
catch(error){
        let e=l;
        consolé.log(e);
}
finally{
      consolé.log("Yo salgo siempre");
}
```

Muestra:

```
1
Yo salgo siempre
```

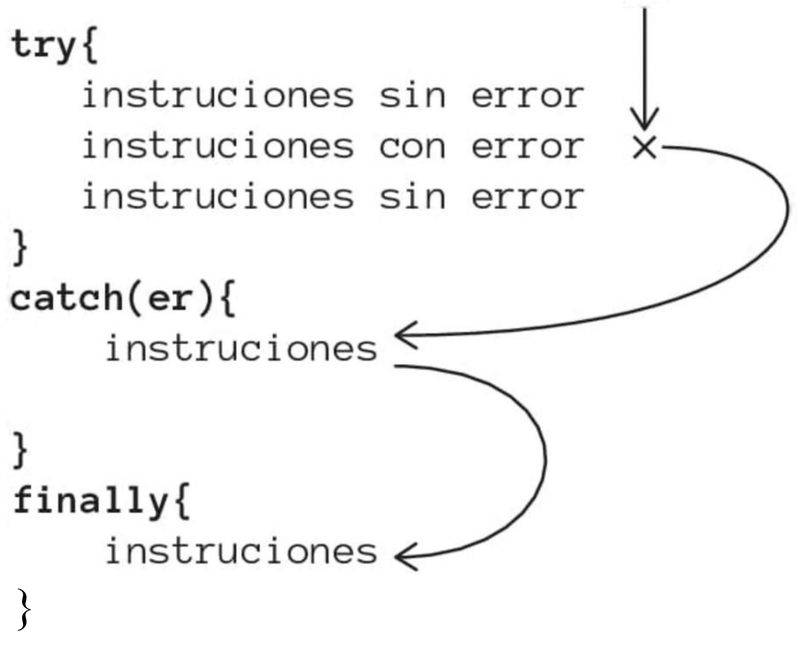


Figura 9.1: Flujo de instrucciones en una estructura try..catch..finally con un error capturado.

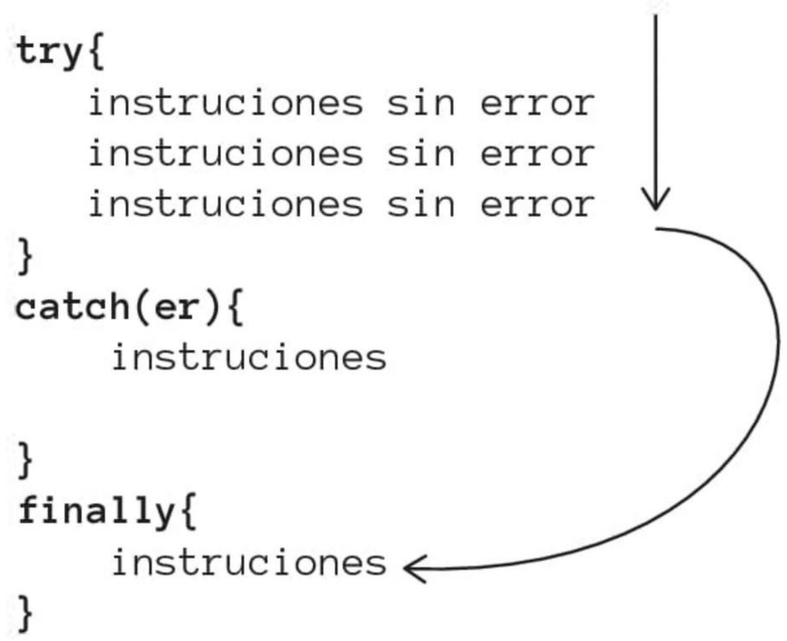


Figura 9.2: Flujo de instrucciones en una estructura try..catch..finally sin que ocurra un error

9.3 MÓDULOS

9.3.1 USO DE MÓDULOS Y PAQUETES

La programación de aplicaciones para **Node.js** ha conseguido influenciar mucho las nuevas normas sobre JavaScript. Una de sus más llamativas aportaciones al JavaScript clásico ha sido el uso de módulos. Hemos visto en temas anteriores (2.4.4 ***"npm",*** en la página 29), como node incorpora un gestor de paquetes llamado **npm.** En la programación front-end el uso de npm está orientado principalmente a la instalación de utilidades interesantes que nos ayuden en el proces de desarrollo.

Cada paquete integra una serie de módulos. Hay que entender que un módulo es un conjunto de funciones, constantes, objetos, etc, que se usan como una librería que podremos reutilizar para facilitar el desarrollo de aplicaciones.

Sin pretender profundizar más en la creación de aplicaciones con node.js (que no son el centro del estudio de este libro), lo cierto es que los módulos permiten de forma eficaz tanto utilizar módulos de librerías de terceros, como crear módulos propios que pueden reutilizarse en todas

nuestras aplicaciones. Esto permite condensar más nuestras aplicaciones y facilitar la reusabilidad del código.

Las aplicaciones de node.js hacen uso de módulos desde hace mucho tiempo. Pero la cuestión es ¿qué pasa con el JavaScript creado para ser ejecutado en un navegador? Es decir ¿qué pasa con el JavaScript del lado del cliente?

Lo cierto es que en el lado del cliente no se usaban módulos, estaban fuera de la norma. Si deseamos usar librerías de funciones y otros elementos, estas se cargaban mediante etiquetas script:

```
<script      src="1ibrerial.js"></seript>
<script      src="1ibreria2.js">              </seript>
<script      src="1ibreria3.js"> </seript>
...
<script>
... código que usa funciones de las librerías anteriores....
</script>
```

El código anterior muestra la forma de trabajar cuando queremos reutilizar código JavaScript o utilizar librerías de terceros (como **jQuery,** por ejemplo). La instrucción script, cuando carga archivos externos, genera una petición http por cada librería y eso puede ralentizar la ejecución de la aplicación.

Especialmente preocupante es el hecho de que, de algunas librerías estándar, solo usemos ciertas funcionalidades, ya que la etiqueta script carga el archivo entero. Y esto no es nada eficiente.

En definitiva, con los años, muchos desarrolladores empiezan a echar mucho de menos un sistema de módulos como el de node.js para el desarrollo de aplicaciones en el lado del cliente.

En atención a ello han aparecido numerosas utilidades para añadir algún mecanismo de carga de módulos para el JavaScript del navegador. Ejemplos de ello son, la librería **Require.js** [1] y también **System.js**[2] que requieren de la carga de un archivo JavaScript con las funciones que aportan para poder importar módulos en nuestro código. Es decir, son librerías que nos proporcionan una forma (no estándar) de importar módulos.

Otras opciones pasan por usar precompiladores que permiten que en nuestro código usemos instrucciones de carga de módulos y a través de herramientas de compilación, ese código se traduce a un código final que dispondrá de los módulos cargados. Ejemplos de esta técnica son las utilidades **Browserify** o **webpack.**

Pero la carencia en el estándar seguía y de hecho, en el momento de escribir estas líneas, sigue habiendo problemas con la carga de módulos usando el estándar. Por ello, las opciones comentadas antes se siguen usando en muchos casos. Si en este libro hemos optado por comentar la carga estándar de módulos es porque parece claro que en el futuro será una práctica habitual a la hora de programar aplicaciones web.

---

1 Disponible en https://requirejs.org/

2 Véase https://github.com/systemjs/systemjs

9.3.2 CARGA Y CREACIÓN DE MÓDULOS

La norma ES2015 (también llamada ES6) al fin incorporó el uso de módulos en JavaScript. Con esta norma aparecieron las palabras claves **export** e **import.**

9.3.2.1 CREACIÓN DE MÓDULOS

Un módulo no es más que un conjunto de funciones, variables, objetos y todo tipo de elementos que puedan ser reutilizados en otro código. Cuando deseemos en nuestro código alguno de los elementos del módulo bastará con indicar qué queremos importar el módulo y qué elementos concretos deseamos de él (o bien importar todo).

Por otro lado, cuando se crea el módulo también hay que indicar qué cosas son importables. Supongamos que estamos creando un módulo llamado ***geometria.js*** en el que deseamos colocar funciones de cálculo de áreas y perímetros de figuras. En ese archivo hemos creado la función **areaCirculo** en base a lo siguiente:

```
export function areaCirculo(radio){
     return Math.PI * radio * pradio;
}
```

La palabra **export** hace referencia a que esa función es exportable en otro archivo JavaScript.

Cada función o variable que queremos exportar debe tener por delante la palabra export:

```
export const PI_CUADRADO=Math.PI*Math.PI;

export function areaCirculo(radio){
      return radio * PI_CUADRADO;
}
export function areaCuadrado(lado){
      return lado ** 2;
}
```

También podemos acumular todo lo que queremos exportar en una sola instrucción **export:**

```
const PI_CUADRADO=Math.PI*Math.PI;

function areaCirculo(radio){
      return radio * PI.CUADRADO;
}
function areaCuadrado(lado){
      return lado ** 2;
}
export{
      PI.CUADRADO,
      areaCirculo,
      areaCuadrado
}
```

Aquellos elementos del módulo que no estén en la instrucción **export** se considerarán privados, y, por lo tanto, no exportables.

93.2.2 CARGA DE MÓDULOS

La importación de módulos se realiza con la instrucción **import.** Esta instrucción debe de ser la primera (puede haber varias instrucciones **import)** del código JavaScript. Si queremos cargar un elemento del módulo, podremos hacer lo siguiente:

```
import { areaCirculo } from "./geometría.js";
consolé.log(areaCirculo(5));
```

El código anterior carga la función ***areaCirculo*** del módulo que se ha configurado en el archivo ***geometría.js.*** Si queremos importar varios elementos del módulo, estos se separan con comas:

```
import { areaCirculo, PI.CUADRADO } from /geometría.js";
```

Podemos renombrar los elementos importados:

```
import { areaCirculo as circulo, areaCuadrado as cuadrado }
                                                    from "./geometría.js";


consolé.log(circulo(9)); //Escribe 88.82643960980423
consolé.log(cuadrado(4)); //Escribe 16
```

También podemos importar todos los elementos de un módulo, pero tenemos que asignar un nombre que se utilizará que usar como **espacio de nombre.** Los espacios de nombre son un identificador que se usa como prefijo delante del nombre de cada elemento importado(función, método, variable, etc.) La razón de su uso es diferenciar los nombres de elementos pertenecientes a dos módulos distintos.

```
import * as geom from /geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
```

Un detalle muy importante es que, en el código HTML, si nuestro código JavaScript utiliza módulos, la etiqueta script que contiene ese código o que carga el código desde un archivo debe utilizar el atributo **type** con el valor **module.** Es decir, el código anterior completo, sería:

```
<script type="module">
import * as geom from "./geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
</script>
```

9.4 PROGRAMACIÓN ASÍNCRONA

9.4.1 PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA

Las funciones de tipo callback (5.4.3 ***"Funciones Callback",*** en la página 183) son uno de los elementos del lenguaje JavaScript que ayudan a entender por qué es un lenguaje asincrono. Pero es momento ya de dedicar un tiempo a las bases de la programación asincrona.

La mayoría de lenguajes de programación son síncronos. Esto significa que, por ejemplo, la línea número 9 del código no se ejecuta si la tarea que realiza la línea 8 no ha terminado. La dificultad que tienen muchos desarrolladores, incluso experimentados, para entender y aplicar bien el lenguaje JavaScript se debe a esta circunstancia: a que antes han sido programadores de lenguajes síncronos. Que JavaScript sea asincrono implica que una instrucción puede no haber terminado su labor cuando ya se está ejecutando la siguiente.

Pero la cuestión es por qué es interesante, incluso fundamental, que JavaScript trabaje de forma asincrona. Un ejemplo típico es una página que necesita cargar dos elementos independientes que muestran información procedente de dos servicios en Internet. Por ejemplo, podríamos crear una página que, en una capa, muestre la temperatura prevista para hoy, proporcionada por un servicio externo que proporciona las temperaturas, y en otra, un mapa que muestre cómo llegar a nuestra oficina, usando un servicio de mapas en Internet.

La temperatura y el mapa tardan en cargar. Si la programación fuera síncrona, el mapa no se carga si antes no ha llegado la información de la temperatura. No nos interesa este efecto, es más eficiente que ambos componentes se carguen de forma independiente. Si el segundo espera al primero, la carga es más lenta y el usuario de la aplicación lo notaría. Lo ideal es que la carga sea asincrona.

En lenguaje síncronos la única manera de conseguir este tipo de efectos es crear varios hilos independientes en los que cada uno realiza una tarea. Así funciona, por ejemplo, el lenguaje Java para conseguir dos procesos que consigan resultados de forma independiente. JavaScript es un lenguaje de un único hilo, no se pueden programar dos hilos. Pero las operaciones sobre la red y otras operaciones de entrada/salida (como consultar bases de datos) se lanzan de forma independiente.

Pero también este efecto tiene sus problemas. ¿Y si deseamos colorear el mapa? El problema es que, aunque tengamos el código para colorear el mapa, debemos asegurarnos de que el mapa ha llegado antes de empezar a colorear. Esto requiere sincronización, algo del tipo: ***cuando el mapa llegue, coloreamos.***

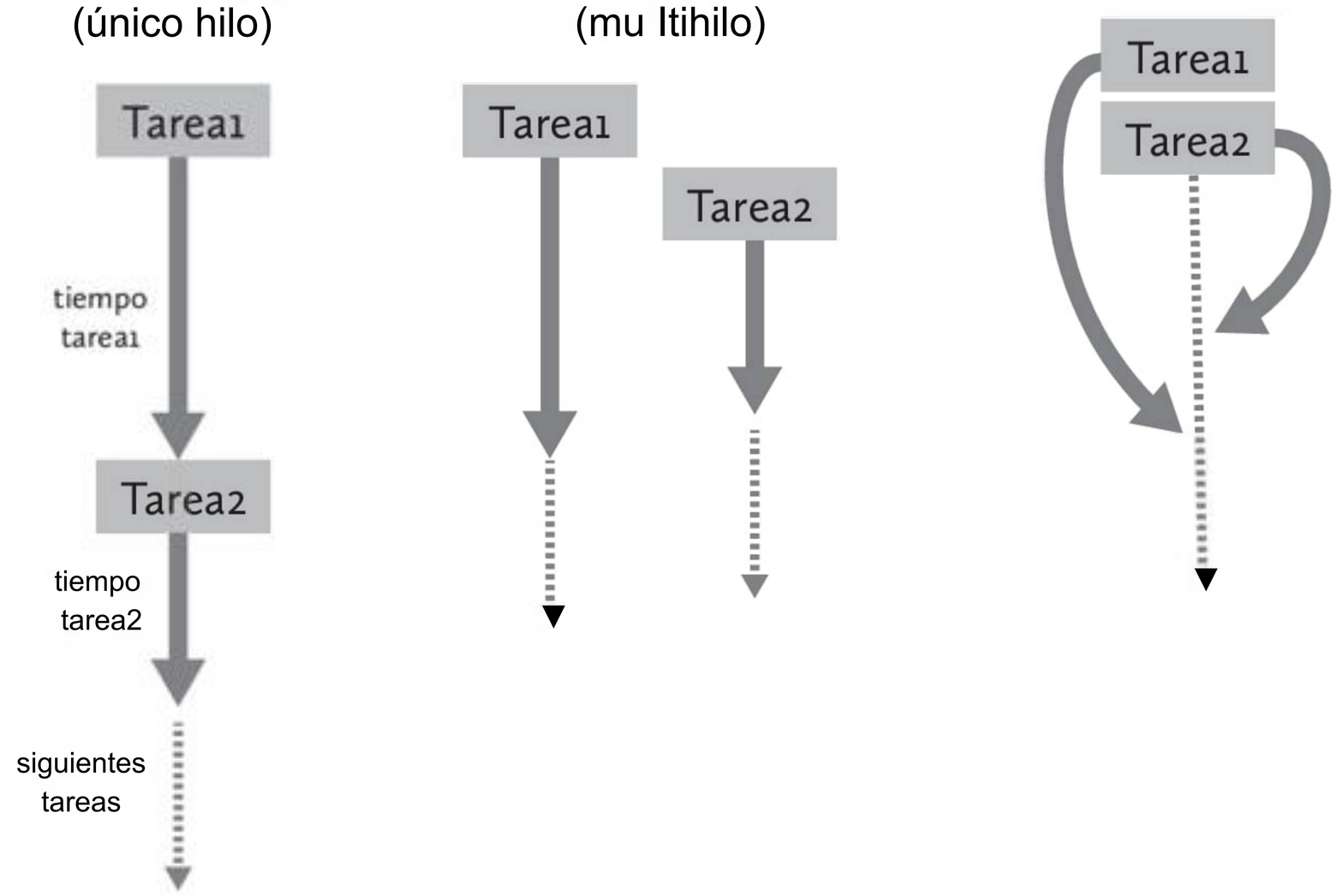


Figura 9.3: Comparación entre los modelos síncronos y asincronos

Solución a esto ya tenemos, puesto que conocemos las funciones callback. Mediante funciones callback podemos hacer que el código de colorear se lance justo cuando se ha cargado el mapa. Algo, como esto:

```
componente.haCargado(()=>{colorear()});
```

Se invoca a colorear cuando el componente se ha cargado.

9.4.2 CALLBACK HELL

Este término que podemos traducir como el infierno de los **callback,** se popularizó hace unos años cuando se percibió el problema de tener que sincronizar numerosas acciones. Esto provoca un exceso de funciones callback que van desplazando el código a su derecha haciéndole poco legible y mantenible.

Siguiendo con el ejemplo anterior del componente de mapas, supongamos que ahora deseamos que el mapa se cargue cuando hagamos clic a un botón concreto. Además, tras colorear el mapa, deseamos realizar una animación sobre el mismo. Pero, todo ello, solo si hemos podido realizar correctamente y sin error las tareas anteriores. El código podría ser:

```
botón.addEventListener("elick",function(ev){
        cargarMapa(componente,function(error){
                if(error){
                        consolé.log("Error al cargar el mapa");
```

```
        }
        else{
            colorear(componente,function(error){
                if(error){
                    consolé.log("Error al colorear");
                }
                else{
                    animar(componente);
                }
            })
        }
    })
});
```

Con tantas llamadas de tipo callback, el código se desplaza mucho a la derecha. En este código hemos hecho que las funciones callback reciban una variable que marca si hubo error en la operación. El hecho de valorar errores es fundamental para saber si el proceso se hizo bien o mal, es muy importante en labores de programación asincrona. Pero esa valoración, que es habitual, desplaza aún más el código.

Ciertamente, no hay duda de que la invocación a ***animar(componente)*** realizará la animación tras ***colorear.*** Por otro lado, colorear se realiza tras la carga del mapa, y la carga no se inicia si no se ha hecho clic en el botón. El planteamiento es correcto, pero la legibilidad es terrible. Un infierno, como dice el título de este apartado.

9.4.3 PROMESAS

9.4.3.1 INTRODUCCIÓN A LAS PROMESAS

La solución al problema anterior es una estructura que permite controlar de forma más organizada las tareas asincronas sin tener que usar tantas funciones **callback** anidadas. Esa nueva estructura es lo que se conoce como **promesa.**

Las promesas, como tantas veces ocurre en JavaScript, se implementaron primero en librerías de terceros. Las más famosas son **Q**[3]**, When**[4]**,** el objeto Promise de la librería **WinJS** de **Microsoft** y la que ha resultado más influyente **RVSP.js**[5]**.** Se siguen usando por parte de algunos desarrolladores y algunas son respetuosas con el estándar (como RVSP.js) pero aportan métodos extras que algunos desarrolladores agradecen mucho.

La norma ES2015 finalmente incorporó las promesas en el estándar y, desde entonces, se utilizan extensamente por parte de la comunidad de desarrolladores.

---

3 https://github.eom/kriskowal/q

4 http s ://github. com/cui o i s/whe n

5 https://github.com/tildeio/rsvp.is

Una promesa permite invocar a una función o tarea, cuya labor requiere una ejecución asincrona. Podremos determinar lo que ocurre en el caso de que esa tarea concluya correctamente y lo que haremos en el caso de concluir mal. En las dos situaciones, podremos recoger información al respecto.

Las promesas pueden tener uno de estos estados:

- **Cumplida *(resolved* o *fulfdled).*** Si la tarea relacionada con la promesa se ha finalizado con éxito.
- **Rechazada *(rejected).*** Si la tarea no finaliza con éxito.
- **Pendiente de finalizar *(pending).*** En proceso de finalización.
- **Finalizada *(settled).*** Independiente de si se ha cumplido o no, la tarea relacionada con la promesa ha finalizado.

Las promesas son objetos de tipo **Promise.** Estos objetos son los que hacen la labor de relacionar la labor asincrona con las acciones a tomar en caso de éxito o fracaso. Para ello, proporcionan varios métodos y, sobre todo, una función de construcción de objetos Promise que es donde se constituye realmente la promesa.

9.43.2 CREACIÓN DE PROMESAS

La creación de promesas sigue esta estructura formal:

```
let promesa = new Promise(function(resolver, rechazar) {

  código de la tarea asíncrona

  if (condición que valida que la tarea fue exitosa) {
    resolver(información sobre el éxito);
  }
  else {
    rechazar(información sobre el fracaso);
  }
});
```

Para explicar el código anterior veamos lo siguiente:

Las promesas son objetos que se construyen indicando una función de tipo callback.

Esa función callback acepta dos parámetros, los cuales son dos funciones. La primera (se la suele llamar **resolve** o **resolver)** se invoca cuando se ha verificado que la operación ha finalizado de forma correcta. La segunda (se suele llamar **reject** o **rechazar)** se invoca cuando se ha determinado que el proceso no ha finalizado correctamente.

Ambas funciones reciben parámetros. A cada una se la envía un objeto que contiene la información de la resolución en caso de éxito (función ***resolve)*** o de fracaso ***(reject).*** Para

la función de rechazo el parámetro suele ser un objeto de error, ya que permite la gestión de errores de forma más conveniente.

La creación del objeto de promesa (la creación de la promesa) implica lanzar la tarea en segundo plano.

9.4.33 MÉTODO **THEN**

Pero la cuestión es cuándo recogemos los resultados del éxito o el fracaso. Ahí interviene el método **then.** Este método acepta una función callback, que será invocada cuando la tarea de la promesa finalice con éxito. Hay un segundo parámetro opcional, que también es una función, cuyo código se invoca si el resultado es erróneo. La sintaxis es la siguiente:

```
promesa.thenífunctionfresultado)!          '...}, function(error          ){  ..
```

La primera función recibe un parámetro. Ese parámetro es el que hayamos pasado a la función ***resolver,*** vista en el apartado anterior, durante la creación de la promesa. La segunda función recibe el parámetro indicado en la función ***rechazar.*** Es poco habitual usar la segunda función, en su lugar se usa el formato de captura de errores que veremos a continuación.

9.43.4 MÉTODO **CATCH**

Como ya hemos dicho, en el caso de la función ***rechazar,*** es habitual lanzar un error. Por eso, hay un método llamado **catch** que permite gestionar el rechazo en la promesa. Es un formato mucho más coherente y que se asemeja a la estructura **try..catch.** Además, se permite encadenar ambos métodos, porque el resultado de los métodos **then** y **catch** es el propio objeto de la promesa(objeto **Promise).** La sintaxis completa es:

```
promesa.then(function(resultado){
    ...
}).catch(function(resultado){
    ...
});
```

En ese código podemos encadenar ambas acciones. Incluso hay un tercer método llamado **finally** que recibe una función callback sin parámetros, cuyo código se ejecuta tanto si la promesa es exitosa como si no.

9.4.33 EJEMPLOS DE PROMESAS

La idea es simple, pero es compleja de entender dada su novedosa sintaxis. Por ello, vamos a empezar haciendo una promesa simple. Haremos una promesa que consista en que, si dos variables valen lo mismo, por pantalla salga un mensaje diciéndolo, y si no así, lanzaremos un error. Evidentemente para una acción como esta, no hace falta usar promesas, pero usamos esta idea para entender el funcionamiento de las promesas.

```
var promesa=new Promise((resolver,rechazar)=>{
    let nl=2;
    let n2=2;
    if(nl==n2) resolver( " ¡Son iguales!");
```

```
        else rechazar(Error("Algo raro ha pasado"));
});

promesa.then((respuesta)=>{
        consolé.log(respuesta);
});
```

Instantáneamente aparece el texto ***"¡Son iguales!"*** porque la condición es verdadera y eso provoca lanzar la función ***resolver*** enviando como parámetro el texto ***"¡Son iguales!".*** El método **then** lanza la función callback que usa como parámetro el texto anterior y ese texto sale por pantalla, ya que la función callback del método **then** solo hace la labor de mostrar el parámetro que reciba por consola.

Si cambiamos los valores de ***ni*** y ***n2*** para que no sean iguales, se provocará una excepción que paraliza el programa y muestra el error. Como no hemos capturado errores, la ejecución de la aplicación finaliza.

Para capturar el error y gestionarlo, es para lo que se usa **catch.** La estructura completa sería esta:

```
var promesa=new Promise((resolver,rechazar)=>{
        let nl=3;
        let n2=2;
        if(nl==n2) resolver(" ¡Son iguales!");
        else rechazar(Error("Algo raro ha pasado"));
});

promesa.then((respuesta)=>{
        consolé.log(respuesta);
}).catch((error)=>{
        consolé.log(error.message);
});
```

Los valores de ***ni*** y ***n2*** son distintos, lo que provoca que invoquemos a la función para el rechazo. A esa función le pasamos como parámetro un objeto de error. El método **catch** ejecuta su función callback a la que se le pasa el objeto de error (sin que la excepción se lleve a cabo) y entonces, simplemente (gracias al método **message** de los objetos de error) se mostrará por consola el mensaje ***"Algo raro ha pasado".***

Veamos este otro ejemplo:

```
var promesa=new Promise((resolver,rechazar)=>{
        throw new Error("La he liado parda");
});

promesa.then((respuesta)=>{
        consolé.log("Me ha ido bien");
}).catch((error)=>{
```

```
        consolé.log("Me ha ido mal");
});
```

En la creación de la promesa, a las primeras de cambio, lanzamos un error. Aunque no se ha gestionado nada con las funciones ***resolver y rechazar,*** lo cierto es que se ha generado un error, y ese error provocará que se ejecute la función callback del método catch. Es decir, por pantalla aparece el texto ***Me ha ido mal.***

Veamos un ejemplo que usa temporizadores:

```
var promesa=new Promise(function(resolver,rechazar){
        let n=0;
        let intervalo=setInterval(function(){
                n++;
                if(n==10) {
                        resolver("Han pasado 10 segundos");
                        clearInterval(intervalo);
                }
        },1000);
});

promesa.then(function(mensaje){
        consolé.log(mensaje);
};
```

En este código, en la creación de la promesa se genera un intervalo cuya función interna es invocada cada segundo. Un contador (n) va incrementándose en cada llamada. Cuando este contador llega a 10, invoca a la función **resolver** con el mensaje ***han pasado 10 segundos.*** El método then será invocado entonces y mostrará ese mensaje por consola.

9.43.6 ENCADENAMIENTO DE MÉTODOS

Se puede invocar varias veces a las funciones de resolución y de rechazo. Pero cada método **then** o **catch** solo puede responder a una. Lo que sí se permite es encadenar los métodos then y catch las veces que haga falta. Eso es una buena manera de estructurar nuestras aplicaciones evitando un exceso de control de funciones callback.

Un ejemplo sencillo de esta idea sería este:

```
var promesa=new Promise(function(resolver,rechazar){
        let n=0;
        let intervalo=setInterval(function(){
                n++;
                if(n==10) {
                        resolver("Han pasado 10 segundos");
                        clearInterval(intervalo);
                }
        },1000);
});
```

```
promesa.then(funetion(mensaje){
        consolé.log(mensaje);
        return "Se ha cerrado el temporizador";
}).then((mensaje)=>console.log(mensaje));
```

A los 10 segundos se muestra ***Hemos llegado a 10 y*** justo debajo el texto ***Se ha cerrado el temporizador.*** El hecho de que el primer **then** use un **return,** provoca que se devuelva una nueva promesa, cuya función **then,** en este caso, se resuelve al instante, mostrando el resultado que hemos comentado.

Con estos códigos parece que estemos simplemente complicando el lenguaje, pero sin utilidad alguna. Será en la unidad siguiente cuando de verdad veamos una utilidad práctica real de las promesas y el encadenamiento de las mismas. Pero no podíamos acabar este apartado sin adelantar de manera conceptual la idea.

Volvamos al ejemplo de colorear un mapa. No podemos colorear el mapa si no hemos podido cargarlo. Supongamos, además, que debemos cargar para colorear, una plantilla de colores de Internet. Sin esa plantilla no podemos colorear. Podríamos crear una promesa asociada al éxito de cargar o no el mapa. Si se carga de forma correcta el mapa, entonces lanzamos el segundo método de cuyo éxito depende el coloreado final.

La idea con promesas sería la siguiente:

```
cargarMapa()
        .then((mapa) = >cargarPlanti 1 la(mapa))
        .then((mapa)=>colorear(mapa));
        .catch(throw new Error("Error en la carga"));
```

La limpieza de este código es maravillosa. Pero la potencia la dan las funciones **cargarMapa, cargarPlantilla** y **colorear.** El resultado de las dos primeras tiene que ser una promesa que indique la correcta finalización de la tarea o su fallo al finalizar.

9.4.37 MÉTODOS DEL OBJETO PROMISE

El objeto **Promise** aporta un método estático llamado **Promise.resolve,** que crea una nueva promesa resuelta y enviando (como si se hubiera invocado al método resolver en la creación de la promesa) el objeto que se envíe como parámetro.

Ejemplo de uso:

```
let promesa=Promise.resolve("Ha funcionado todo");
promesa.then((mensaje)=>{consolé.log(mensaje)});
```

Se muestra: ***Ha funcionado todo.*** Luego, lo que hace este método es crear una promesa cumplida.

El método contrario es **Promise.reject:**

```
let promesa=Promise.reject(Error("No ha funcionado nada"));
promesa
```

```
    .then((mensaje)=>{consolé.log(mensaje)})
    .catch((error)=>{consolé.log(error.message)});
```

Muestra el mensaje ***"No ha funcionado nada".*** Luego, lo que hace este método es crear una promesa rechazada.

Otro método interesante es **Promise.all.** Este método devuelve una promesa cumplida si todas las promesas, de la colección que recibe como parámetro, son cumplidas. Si alguna se rechaza, entonces, el método devuelve una promesa rechazada. Ejemplo:

```
let promesal=Promise.resolve("Estoy resuelta");
let promesa2=new Promise((resolver)=>{
      setTímeout(()=>{resolver("Resuelvo en 3s")},3000);
});
let promesa3=new Promise((resolver)=>{
      setTimeout(()=>{resolver("Resuelvo en 6s")},6000);
});
let promesaConjunta=Promise.al1([promesal,promesa2,promesa3]);

consolé.log("Empezando");

promesaConjunta.then((resultados)=>{
   let n=l;
   for(let resultado of resultados){
        consolé.log('Promesa n2 ${n}: Mensaje:${resultado}');
        n++;
   }
});
```

Se crean tres promesas llamadas ***promesal, promesa2 y promesa3.*** La primera genera una resolución positiva al instante, la siguiente tras tres segundos y la tercera tras seis segundos. Hay que recordar que **setTimeout** es asincrona y realiza sus acciones en segundo plano.

La promesa conjunta, creada con el método **Promise.all,** exige del cumplimiento de las tres promesas para ser considerada una promesa resuelta. En este caso pasarán seis segundos antes de saber que las tres están resueltas. Lo cual significa que este código muestra el texto ***Empezando y*** seis segundos después, de golpe, muestra este texto:

```
Empezando
Promesa  n2  1:  Mensaje:Estoy resuelta
Promesa  n2  2:  Mensaje:Resuelvo  en  3s
Promesa  n2  3:  Mensaje:Resuelvo en 6s
```

Hasta que no se resuelve la tercera, no podremos saber si las anteriores se han resuelto. Esto significa que este método es fantástico para sincronizar acciones asincronas.

Es importante resaltar que, si alguna promesa se rechazara o provocara un error, instantáneamente se generaría el rechazo, sin esperar al resto de promesas de la lista.

9.4.4 FUNCIONES **ASYNC**

Hay otras posibilidades en JavaScript de sincronización más avanzadas. En la norma ES2017 apareció una esperada mejora, conocida como **await/async.** Actualmente, salvo Internet Explorer como es habitual, todos los navegadores actuales reconocen esta estructura que explicaremos a continuación.

Hay funciones especiales que podemos declarar anteponiendo la palabra **async.** Esto declara un tipo de función especial, que internamente es un objeto de tipo **AsyncFunction** que devuelve una promesa implícita. Pero lo que nos interesa realmente, es que en las funciones de tipo async podemos utilizar la palabra clave **await** para poder sincronizar varios elementos asincronos.

Es decir, la función puede requerir terminar un proceso antes de iniciar un segundo proceso que dependa de él. Hasta ahora como mecanismos disponíamos de las funciones callback y de los métodos **then** y **catch** de las promesas. No es que ya no necesitemos estos elementos, es que ahora les podemos integrar con un operador que aporta más legibilidad al código. El operador **await** requiere de una promesa, si se cumple, entonces la siguiente línea se ejecutará, si no, se mantiene en espera. Veamos un ejemplo:

```
let promesa1=Promise.resolve("Estoy resuelta");
let promesa2=new Promise((resolver)=>{
        setTimeout(()=>{resolver("Resuelvo en 3s")},3000);
});
let promesa3=new Promise((resolver)=>{
        setTimeout(()=>{reso1ver("Resuelvo en 6s")},6000);
});

async function esperarTiempos(){
        let mensaje1=await promesa1;
        consolé.log(mensaje1);
        let mensaje2=await promesa2;
        consolé.log(mensaje2);
        let mensaje3=await promesa3;
        consolé.1og(mensaje3);
}

esperarTiempos();
```

Las tres promesas iniciales son las que vimos en el apartado anterior para explicar el método **Promise.all.** La primera se resuelve al instante, la segunda tarda tres segundos y la tercera tarda seis segundos. La función **esperarTiempos** se marca con la palabra clave **async** y eso permite que haya tres variables ***(mensaje1, mensaje2*** y ***mensaje3)*** que graben el resultado del ***resolver*** de las promesas pero haciendo que se espere ese resultado antes de pasar a la siguiente línea de código. Eso provoca que el primer mensaje sale al instante, el segundo tres segundos desde que se inició el programa y el tercer mensaje saldrá a los seis segundos. Los segundos no se acumulan, porque las promesas se han creado previamente.

Diferente habría sido este otro código:

```
async function esperarTiempos(){
        let mensajel=await Promise.resolve("Estoy resuelta");
        consolé.log(mensajel);
        let mensaje2=await new Promise((resolver)=>{
                        setTimeout(()=>{resolver("Resuelvo en 3s")},3000);
        });
        consolé.log(mensaje2);
        let mensaje3=await new Promise((resolver)=>{
                        setTimeout(()=>{resolver("Resuelvo en 6s")},6000);
        });
        consolé.log(mensaje3);
}
esperarTiempos();
```

El código es similar, pero ahora cada promesa se genera tras esperar la finalización de la anterior. A la vista ocurre que el primer mensaje sale inmediatamente, el segundo tarda tres segundos desde que se mostró el primer mensaje y para ver el tercer mensaje hay que esperar seis segundos más, es decir, el último mensaje aparece a los nueve segundos.

Lo interesante de await es la facilidad que aporta para la sincronización. Además, es muy versátil, no obliga a que lo que tiene a su derecha sea un objeto de tipo **Promise,** si es una expresión normal, la convierte en promesa y funciona también:

```
async function escribeYa(){
    let texto-await "ya esta!";
    consolé.log(texto);
}
escribeYa(); //Escribe: ya está!
```

Hay otros tipos de objetos con los que trabaja al estilo de las promesas. Son objetos que tienen implementados un método **then.** En estos objetos, el método ***then*** tiene que estar creado al estilo del método ***then*** de las promesas. Es decir, tiene que recibir una función callback para resolver y, opcionalmente, otra para rechazar. Se les llama a estos objetos, objetos ***thennables,*** simplemente son objetos con un método then correcto implementado.

Otra cuestión es qué pasa si alguna de las promesas (u objetos ***thennables)*** son rechazadas. ¿Cómo capturamos el rechazo? Lo lógico para ello es que el rechazo signifique lanzar un error, por lo que basta con que dispongamos una estructura de tipo **try..catch.**

```
async function falla(){
    try{
        let resultado=await Promise.reject(Error("Promesa rechazada"));
    }
    catch(error){
        consolé.log(error.message);
    }
}
falla(); //Escribe: Promesa rechazada
```

Incluso es posible usar el método **catch** obviando la estructura **try..catch.**

```
async function falla(){
    let resu1tado=awa it Promise.reject(
                                    Error("Promesa rechazada")
                            ).catch(error=>{consolé.log(error.message)});
}
falla(); //Escribe: Promesa rechazada
```

9.5 PRÁCTICAS RESUELTAS

Práctica 9.1: Temporizador hecho con promesas

- Crear una función que reciba un número de milisegundos y genere un temporizador que escriba el texto ***"Tiempo concluido"*** cuando pasen dichos milisegundos.
- Realmente la función no escribe, sino que genera una promesa cuyo resultado es dicho texto.
- En el caso de que no se cumpla la promesa, el resultado será el texto ***"El tiempo no va bien".***
- Se decidirá que la promesa no se ha cumplido cuando pase el doble de tiempo que el indicado en los milisegundos.

SOLUCIÓN: PRÁCTICA 9.1

Esta práctica busca que nos familiaricemos con la programación de promesas. Aunque, con saber cómo usarlas es suficiente para los objetivos principales de este libro, fabricar promesas nos ayudará a que esos objetivos los alcancemos de forma más óptima. Además, la creación de promesas ayuda a mejorar la forma de crear programas asincronos.

Los temporizadores en JavaScript se crean con las funciones **setInterval** y **setTimeout** (véase "Temporizadores"en la página 279) En este caso, la idónea es **setTimeout,** porque solo necesitamos que se ejecuta una vez.

El código de la función sería el siguiente:

```
function temporizador(tiempo){
    var promesa=new Promise((resolver, rechazar)=>{
        var temp=setTimeout(()=>{
            clearTimeout(temp2);
            resolver("Tiempo concluido");
        },tiempo);
        var temp2=setTimeout(()=>{
            rechazar("El tiempo no va bien");
        },tiempo*2);
    });
    return promesa;
}
```

La función crea dos temporizadores, uno con el tiempo que recibe la función como parámetro y el segundo con el doble. El primero invoca a la función callback ***resolver*** porque considera cumplida la promesa. El segundo invoca a ***rechazar.***

Suponiendo que estamos en una página web vacía, escribimos los resultados de la promesa como texto completo de la página. Invocaremos a la función con 500 ms.

```
temporizador(5000).then((texto)=>{
        document.body.innerHTML=
        "<p>"+texto+"</p>";
}).catch((mensaje)=>{
        document.body.textContent=mensaje;
});
```

Práctica 9.2: Módulo de funciones de temporizadores

- Crea un archivo llamado **temporizador.js,** el cual será un módulo que podremos importar a nuestro código JavaScript principal.
- Este módulo constará de dos funciones:
  - La función **temporizador** creada en la práctica anterior.
  - La función **cuenta.** Esta función será capaz de escribir una cuenta atrás, y mostrarla en un elemento de HTML. Los parámetros son:
    * El número con el que se inicia la cuenta atrás.
    * El elemento en el que escribiremos la cuenta atrás. Por defecto usará el elemento **body.**
    * El intervalo en milisegundos en el que cambia cada número. Por defecto valdrá 1000.
    * Una función callback, cuyo código se ejecuta cuando la cuenta finalice.
- Crea, además, una aplicación web que cree en dos párrafos de modo que en el primero se cuente atrás desde el 6, pasando de segundo en segundo. En el segundo se contará desde el 60 pero cada número se moverá de décima en décima. Además, al llegar a cero, queremos que escriba ***"Fin".***

SOLUCIÓN: PRÁCTICA 9.2

El módulo de las dos funciones sería el siguiente:

```
/**
  *   Crea una promesa que genera un mensaje en el
  *   tiempo indicado
  *   ©param {Number} tiempo ms
*/
export function temporizador(tiempo){
        var promesa=new Promise((resolver, rechazar)=>{
                var temp=setTimeout(()=>{
                        clearTimeout(temp2);
                        resolver("Tiempo concluido");
                },tiempo);
                var temp2=setTimeout(()=>{
                        rechazar("El tiempo no va bien");
                },tiempo* *2);
        });
```

```
        return promesa;
}
/**
  *   Crea una cuenta atrás en el elemento indicado
  *   @param {number} numero Desde dónde contamos
  *   @param {*} elemento En que elemento HTML escribimos
  *   ©param {*} tiempo ms en los que cambia cada número
  *   @param {*} f Función opcional que queremos que se ejecute al final
  */
export        async        function        cuenta(numero,elemento=document.body,tiempo=1000,f)
{
        try{
                for(let innúmero;i>=0;i —){
                        await temporizador(tiempo);
                        elemento.textContent=i;
                }
                if (í) f();
        }
        catch(e){
                consolé.log("Error ”+e);
        }
}
```

La función de la cuenta atrás es asincrona, usa la función **await** y la captura de errores con **try..catch.** De esta forma, el código es más legible y usa de forma eficaz la función del temporizador creada anteriormente. Aunque lo que hace esta función es complejo, su código final resulta sencillo.

Gracias a este módulo, la página en sí resuelve el problema en apenas cuatro líneas de JavaScript:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <meta name="viewport" contenta"width=device-width,
initial-scale=l.0">
        <meta http-equiv="X-UA-Compatible" contenta"ie=edge">
        <title>Documente/title>
        <style>
                p{
                        font-size:2em;
                }
        </style>
</head>
<body>
        <p> </p>
        <p> </p>
```

```
<script type="module">
        import { temporizado!", cuenta } from " ./temporizador. js" ;
        let pl=document.querySelector("p:first-of-type");
        let p2=document.querySelector("p:last-of-type");
        cuenta(6,pl)
        cuenta(60,p2,100,()=>{
                p2.textContent="Fin";
        });
</script>
</body>
</html>
```

Las invocaciones a la función ***cuenta,*** hacen muy sencillo el código.

```
cuenta(6,pl)
```

Con esta invocación, decimos que queremos contar desde el número ***6*** y que se muestre en el elemento ***pl.*** Como no indicamos tiempo, se cambiará de número en cada segundo. Al no indicar función alguna, al final no se hará nada.

```
cuenta(60,p2,100,()=>{
     p2.textContent="Fin";
});
```

Este otro código, cuenta desde el número ***60,*** se muestra la cuenta en el elemento ***p2,*** se cambiará de número cada 100 milisegundos (una décima) y al final se ejecuta el código de la función flecha que escribirá la palabra ***Fin*** en el propio elemento ***p2.***

9.6 RESUMEN DE LA UNIDAD

- JavaScript ha evolucionado en estos años, incorporando nuevas estructuras en el lenguaje y mejoras que han permitido dar un nuevo impulso para manejar de forma más eficientes elementos como el control de errores, las promesas o la carga dinámica de módulos.
- El control de errores es fundamental para conseguir aplicaciones correctas y eficientes.
- Los errores se producen por distintas causas, principalmente por fallos en la programación (que se pueden descubrir de forma temprana o después de diversas pruebas) o por otro tipo de circunstancias inesperadas.
- La forma de trabajar de JavaScript no facilita el control de errores, por ello disponemos del modo estricto que se hace más sensible a los errores.
- La gestión de excepciones (errores que cortan la ejecución del programa) en JavaScript se puede gestionar con la ayuda de la estructura **try..catch,** la cual permite que si una excepción se produce, el flujo sea controlado por la estructura catch, la cual manejará de la mejor forma el error ocurrido.
- La instrucción **catch** recibe un objeto de error que contiene los detalles del mismo. El propio programador puede crear objetos de error e incluso lanzarlos con ayuda de la instrucción **throw.**
- La estructura **try..catch** puede incorporar un bloque **finally** cuyas instrucciones se ejecutan tanto si hay error, como si no.
- La carga de módulos permite un uso de librerías de funciones y objetos más eficientes que la carga con la etiqueta **script** de HTML.
- Podemos crear módulos utilizando la palabra clave **export** para indicar qué elementos son exportables del módulo. Al cargar el módulo, mediante la instrucción **import** podemos decidir qué elementos concretos importamos.
- JavaScript es un lenguaje asincrono, lo que implica que hay acciones que finalizan en segundo plano. Eso es una ventaja para la programación de aplicaciones web.
- Necesitamos en muchas ocasiones poder sincronizar estas acciones. Un primer mecanismo fueron las funciones callback, pero tienden a complicarse enormemente produciendo el llamado ***callback hell,*** cuando encadenamos varias acciones a sincronizar.
- Las promesas permiten valorar una acción y decidir si finaliza bien o mal. Disponen de un método **then** cuyo código (envuelto en una función callback) se ejecuta en caso de que la promesa se resuelva y de un método llamado **catch,** cuyo código se ejecuta en caso de que la promesa no se resuelva.
- El uso de promesas permite el encadenamiento de las mismas encadenando métodos then y catch. Esta estructura es más legible que el uso anidado de funciones callback para controlar acciones asincronas.

Los objetos Promise aportan métodos para generar promesas cumplidas y rechazadas, así como para crear promesas a partir del cumplimiento de una serie de varias promesas.

La norma ES2017 trajo la posibilidad de crear funciones **async,** las cuales admiten el uso del operador **await** para generar código síncrono, basado en promesas, aun más legible y eficiente.

9.7 TEST DE REPASO

1.- **Una acción asincrona de JavaScript va a tardar 2 segundos en realizarse. Justo detrás, en el código, se lanza otra acción que tardará en completarse 1 segundo,. ¿Cuánto tardan, en teoría, en JavaScript en completarse ambas acciones?**

a) 3 segundos.

b) 2 segundos.

c) 1 segundos.

d) Instantáneamente.

2.- **Se programan las acciones anteriores de forma síncrona. ¿Cuánto tardan en completarse ambas acciones?**

a) 3 segundos.

b) 2 segundos.

c) 1 segundos.

d) Instantáneamente.

3.- **Hemos creado un módulo y sabemos que está correctamente hecho. Sabemos que la instrucción de importación también. EL código está dentro de una etiqueta script de HTML y se está ejecutando en la última versión de Mozilla Firefox. ¿Qué puede pasar?**

a) Que solo funciona la importación si el código está en un archivo externo.

b) Que Firefox no acepta la importación dinámica de módulos.

c) Que no hemos usado type="module" en la etiqueta script.

d) Las dos últimas son correctas.

4.- **Queremos importar las funciones colorear y recortar del archivo paint.js. ¿Qué código lo hace posible?**

a) **import colorear,recortar from paint.js**

b) **import colorear,recortar from "./paint,js"**

c) **import { colorear,recortar } from "./paint,js"**

d) Todas son correctas

5.- **¿Qué diferencia hay entre error y excepción?**

a) La excepción es controlable, el error no.

b) El error es controlable, la excepción no.

c) Ninguna.

6.- **En qué norma del estándar ECMAScript apareció la estructura async/await**

a) ES2015

b) ES2016

c) ES2017

d) ES2018

7.- **En qué norma del estándar ECMAScript aparecieron las promesas**

a) ES2015

b) ES2016

c) ES2017

d) ES2018

8 - ¿Qué escribe este código?

```
try{
    consolé.log("punto 1");
    throw new Error("mi error");
    consolé.log("punto 2");
}
catch(error){
    consolé.log("punto 3");
}
```

a) punto 1
punto 2
punto 3

b) punto 2
punto 3

c) punto 1
punto 3

d) punto 1
punto 2

e) punto 3

f) punto 1

9.- **¿Qué escribe este código?**

```
try{
      consolé.log("punto 1");
      throw new Error("mi error");
      consolé.log("punto 2");
}
catch(error){
       consolé.log("punto 3");
}
finally{
       consolé.log("punto 4");
}
```

a) punto 1
punto 2
punto 3
punto 4

b) punto 2
punto 3
punto 4

c) punto 1
punto 2
punto 3

d) punto 1
punto 3

e) punto 1
punto 4

f) punto 1
punto 3
punto 4

10.- **Hemos visto que Promise.all recibe una lista de promesas y devuelve una nueva promesa. ¿Cuándo se cumple la promesa?**

a) Cuando se cumple cualquier promesa de la lista

b) Cuando se cumplen todas las promesas de la lista

c) Cuando se cumplen o se rechazan todas las promesas de la lista

d) Cuando se cumple o se rechaza al menos una promesa de la lista

11.- **¿Qué escribe este código?**

```
Promise.resolve(3!=3)
       .then((o)=>consolé.log(l))
       .catch((o) = >consolé.log(2));
```

a) 3!=3

b) 1

c) 2

d) Nada, hay un error.

12.- **¿Qué escribe este código?**

```
async function f(){
  try{
    let b=await Promise.
                                  resolve(3!=3);
    consolé.log(l);
  }catch(e){
       consolé.log(2)
  }
}
f();
```

a) 3!=3

b) 1

c) **2**

d) Nada, se queda esperando indefinidamente

13.- **¿Qué escribe este código**

```
new Promise((a,b)=>{
       if(2==2) b("ok");
       else a("fallo");
}).then((o)=>{
       consolé.log(o);
}).catch((o)=>{
       consolé.log(o);
});
```

a) ok

b) fallo

c) Nada, ocurre un error

UNIDAD 10

AJAX

OBJETIVOS

— Asimilar el funcionamiento de los mecanismos de comunicación cliente/ servidor utilizando AJAX

— Reconocer las ventajas y desventajas del uso de AJAX

— Aplicar los métodos del API Fetch para realizar peticiones AJAX y procesar sus resultados

— Personalizar las peticiones AJAX modificando las propiedades de los paquetes http

— Identificar los formatos habituales de envío y recepción de datos mediante AJAX

— Procesar los resultados de las peticiones AJAX dependiendo del tipo de datos deseado

— Enviar datos a los servicios de destino de las peticiones usando el formato apropiado

CONTENIDOS

10.1 COMUNICACIÓN CLIENTE/ SERVIDOR

10.2 INTRODUCCIÓN A AJAX
- 10.2.1 ¿QUÉ ES AJAX?
- 10.2.2 VENTAJAS DE AJAX
- 10.2.3 DESVENTAJAS DE AJAX

10.3 PETICIONES AJAX
- 10.3.1 USO DE UNA API PARA ACCEDER A UN SERVICIO WEB
- 10.3.2 CORS
- 10.3.3 APIS EN JAVASCRIPT PARA EL USO DE AJAX

10.4 REALIZAR PETICIONES MEDIANTE FETCH

10.5 MANIPULAR LA RESPUESTA. OBJETO RESPONSE
- 10.5.1 PROPIEDADES Y MÉTODOS DEL OBJETO DE RESPUESTA
- 10.5.2 DATOS DE LA RESPUESTA
- 10.5.3 PROCESAMIENTO DE LAS RESPUESTAS

10.6 PERSONALIZAR LA PETICIÓN. OBJETO REQUEST
- 10.6.1 PROPIEDADES Y MÉTODOS DE REQUEST
- 10.6.2 ESTABLECER LA CABECERA. OBJETO HEADERS

10.7 ENVIAR DATOS CON LA PETICIÓN
- 10.7.1 ENVÍO DE DATOS USANDO GET
- 10.7.2 ENVÍO DE DATOS DE FORMULARIO
- 10.7.3 ENVÍO DE PARÁMETROS EN FORMATO JSON

10.8 USO DE AWAIT/ASYNC CON FETCH

10.9 PRÁCTICAS SOLUCIONADAS

10.10 PRÁCTICAS RECOMENDADAS

10.11 RESUMEN DE LA UNIDAD

10.12 TEST DE REPASO

10.1 COMUNICACIÓN CLIENTE/SERVIDOR

En la primera unidad (1.1 ***"Creación de aplicaciones Web",*** en la página 2) expusimos el funcionamiento general de las aplicaciones web. Hemos también diferenciado lo que se ejecuta en el lado del cliente **(front-end)** y lo que se ejecuta en el lado del servidor **(back-end).**

La explicación dada en esa unidad obviaba una cuestión que se produce cada vez más habitualmente: el hecho de que el propio código front-end pueda realizar peticiones en segundo plano a un servidor.

Supongamos que hemos creado una aplicación que, entre las diversas cosas que ofrece, indica en un cuadro la temperatura que hace en la zona que vive el usuario. Podríamos acudir a un servicio de terceros que, enviando la localidad o las coordenadas GPS, nos devuelva la temperatura en un formato que nos permita recogerlo desde JavaScript y así actualizar el panel.

El resto del contenido no depende de esa petición. Lo ideal es que se realice en segundo plano, sin que el resto de tareas tengan que esperar a que esta termine. Si el servicio de temperaturas cae, lo único que percibirá el usuario es que no se muestra la temperatura, pero el resto de la página funcionará con normalidad.

En definitiva, una aplicación web, desde este prisma, se puede ver como un conjunto de componentes independientes, algunos de los cuales son clientes que realizan peticiones a servidores de Internet. Este paradigma es la realidad más habitual de las aplicaciones web actuales.

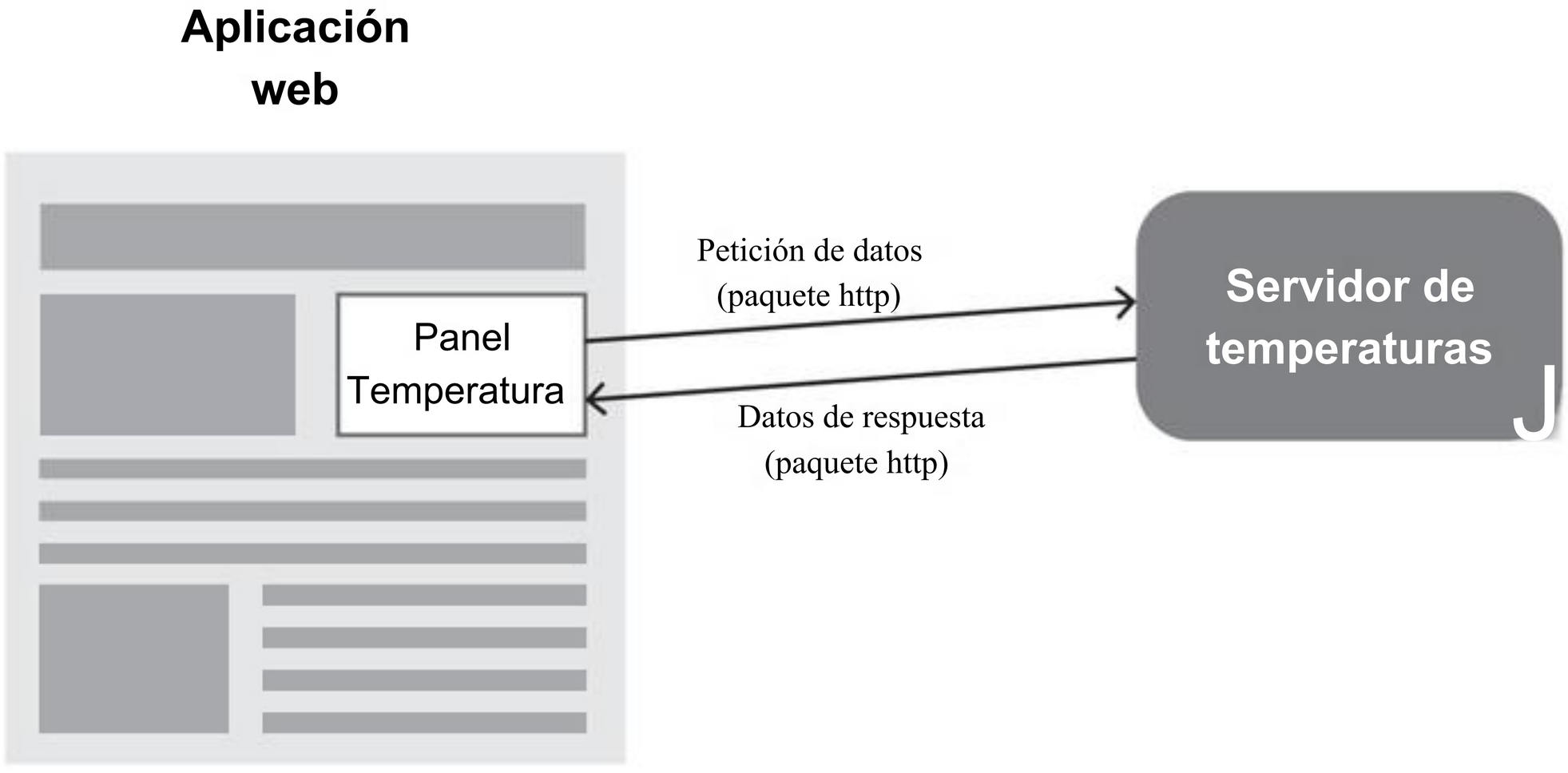


Figura 10.1: Funcionamiento básico de las peticiones AJAX

10.2 INTRODUCCIÓN A AJAX

10.2.1 ¿QUÉ ES AJAX?

Asynchronus JavaScript and XML es el acrónimo que se esconde bajo el término AJAX. Lo más importante de esas siglas es la idea de comunicación asincrona, que soluciona la idea expresada en el punto anterior, ya que la comunicación asincrona permite que la petición y la respuesta se ejecute sin interferir en el resto de instrucciones. El hecho de que haga referencia a XML es porque cuando se ideó esta tecnología, XML era el formato documental predominante y los datos se enviaban en este formato. Actualmente el formato predominante es JSON por lo que algunas personas hablan de **AJAJ** en lugar de **AJAX.**

El nacimiento oficial de esta tecnología se considera que ocurrió cuando en la versión 5.0 de Microsoft Internet Explorer se dio compatibilidad a un nuevo objeto llamado **XMLHttpRequest.** lnicialmente solo se podía utilizar desde componentes **ActiveX** (una tecnología propietaria de Microsoft para enriquecer las aplicaciones web). Más adelante pasó a ser un objeto integrado en todos los navegadores y ahora ya es parte de la especificación oficial de JavaScript y el DOM.

Realmente el término AJAX fue propuesto por **Jesse James Garret** en un artículo que publicó sobre cómo crear nuevas aplicaciones web y que se basaba en el funcionamiento de **Google Pages.** La tecnología ya existía, pero fue este artículo el que consiguió que se adoptara el nombre oficial.

AJAX ha supuesto una revolución enorme en la creación de aplicaciones web, ya que permite que la aplicación mute y se modifiquen sus elementos en base a las acciones del usuario, sin que este sufra tiempos de espera, ya que la carga de componentes cuyos datos proceden de bases de datos o de otros servicios de Internet se produce en segundo plano. Tanta ha sido la influencia de AJAX que, por parte de muchas personas, se la considera la principal responsable de la creación de la llamada **web 2.0** hace unos años. Lo que es indudable es que ha tenido, y tiene, una enorme influencia en numerosas técnicas de creación de aplicaciones web.

Quizá el ejemplo que mejor explica AJAX son las páginas que usan el llamado **scroll infinito,** que determinó las experiencias de usuario de numerosas aplicaciones web orientadas a redes sociales como **twitter, Facebook, lnstagram** y, sobre todo por ser la precursora de esta técnica, **Pinterest.**

En estas aplicaciones el usuario ve la información ordenada de más reciente a más antigua. Lógicamente no se cargan todos los datos a la vez (en algunos casos sería miles y miles de entradas), sino los primeros, los más recientes. Cuando la aplicación detecta (mediante el evento **scroll)** que el usuario se acerca al final de los datos actuales, se cargan en segundo plano las siguientes entradas. El usuario percibe que antes estaba al final de la barra de desplazamiento y ahora la barra ha crecido y puede seguir descendiendo para ver cada vez más datos.

Un scroll infinito bien hecho hace que el usuario tenga la percepción de que jamás llegará al final, de ahí el nombre de esta técnica.

Hay muchos más ejemplos, todos se basan en que se carga información que procede de otro servidor y que esa carga se hace en segundo plano. En muchas páginas en lugar de un solo panel de aviso tipo ***"Cargando..."*** avisando de que hay muchos datos por cargar. Hay un aviso de **car-**

gando por cada componente que se esté cargando. Aunque no todos estén cargados, podremos trabajar perfectamente con los que sí.

Las peticiones se hacen enviando paquetes **http** y la respuesta es también un paquete http cuyo cuerpo contiene datos. Hace unos años los datos estaban siempre en formato XML, originalmente solo se daba cabida a este tipo de documento. En la actualidad **JSON** es el formato habitual para los datos, dada su excelente compatibilidad con JavaScript, pero, en realidad, se pueden enviar los datos en muchos otros formatos.

10.2.2 VENTAJAS DE AJAX

AJAX aporta importantes ventajas al campo de la creación de aplicaciones web. Entre ellas:

- **Carga de contenido remoto en segundo plano,** permitiendo que la aplicación funcione sin ralentizarse por realizar dicha operación de carga.
- Aporta una **gran facilidad para aplicar paradigmas de interfaces de usuarios novedosas,** eficientes y muy estéticas como páginas con **scroll infinito,** aplicaciones web de página única **(SPA, *Single Page Applications),*** páginas de integración de múltiples servicios, páginas con información de noticias en tiempo real, navegabilidad independiente de los botones de adelante y atrás, paginación dinámica, etc.
- **Facilidad y versatilidad** para comunicar con APIs de terceros para integrar sus servicios en nuestras aplicaciones.
- **Resolución de problemas reales** de forma sencilla y lógica.
- **Validación de formularios eficaz** al facilitar la detección de errores de forma temprana (a medida que el formulario se va rellenando) y de relleno automático de controles dependientes. Por ejemplo, en muchos formularios tras elegir el país en el que vivimos, en segundo plano se rellena una segunda lista con las provincias de ese país concreto. Esa lista de provincias se rellena tras precisar el país, lo que implica una consulta a otro servicio web en segundo plano.
- **Mejor uso del ancho de banda del usuario.** No se satura, ya que se hacen peticiones pequeñas de datos cada vez. Esto es discutible porque no siempre, por razones del propio protocolo http que no es veloz con peticiones pequeñas de datos, es más eficiente.
- En la programación **back-end,** saber que un servicio puede ser destino de peticiones AJAX, hace que **los servicios se programen de forma más eficiente e independiente,** animando a la creación de Interfaces de Programación de Aplicaciones **(APIs)** que son la base de la programación basada en servicios. Este tipo de paradigma (conocido como **SaaS, *Software as a Service)*** ha impulsado un mayor desarrollo y eficiencia de las aplicaciones web.
- Facilita la **compatibilidad con servidores back-end de todo tipo de tecnologías.** No importa si el servidor se ha programado en **PHP, ASP.Net, JSP** o cualquier otra tecnología. Solo importa que acepte peticiones **http** y que envíe los datos en formatos estándares como son **XML** o **JSON.**

10.2.3 DESVENTAJAS DE AJAX

Es importante saber que AJAX tiene sus desventajas. Las principales son:

- Dificultad, en bastantes casos, para que los buscadores indexen todos los contenidos que integramos en nuestras aplicaciones procedentes de peticiones AJAX. Un caso emblemático es el de la paginación dinámica. Si el usuario recibe contenidos de gran tamaño, lo normal es paginarlos para no provocar una gran espera en la llegada de los datos. Si el avance a la segunda página no abandona la actual y recarga los contenidos mediante AJAX, la URL de la aplicación es la misma, y sin embargo, lo que muestra es distinto.

  Es más fácil que un buscador indexe bien el contenido si el avance a la segunda página es mediante un enlace normal. Aunque, muchos buscadores actuales saben lo que ocurre en segundo plano y son capaces de realizar cierta indexación de los contenidos dinámicos. Pero, esto último solo funciona ante peticiones http predecibles en segundo plano.

- Aunque hemos dicho que los conceptos en el uso de AJAX no son difíciles, lo cierto es que la implementación tiene sus dificultades porque requiere de un manejo avanzado de JavaScript en aspectos como funciones callback, promesas, programación asincrona, programación basada en eventos, etc. En definitiva, manejar el lenguaje JavaScript de forma avanzada. Para nosotros, a estas alturas, esto ya no debería ser un problema.

- Las peticiones AJAX siguen siendo visibles para los usuarios, por lo que son sensibles a ataques malintencionados. No se deben pasar datos críticos como contraseñas en las peticiones. El hecho de que los datos lleguen en segundo plano no significa que sean más seguros, pero nos pueden dar esa sensación dando lugar a problemas de seguridad al programar.

- La barra de navegación (botones de adelante y atrás) no permiten ir al estado anterior en la página. Las peticiones AJAX no se reflejan en la navegación. Lo mismo ocurre con los marcadores, cuando el usuario enlaza en los marcadores la página, solo guarda la raíz de la URL, no el estado actual. Se pierde, pues, una funcionalidad clásica de las aplicaciones web.

- Puede incrementar el tráfico hacia los servidores de Internet. El hecho de que podamos realizar peticiones http a servicios de Internet, puede facilitar la creación de aplicaciones que hagan peticiones http constantemente. Ante este problema muchos servidores utilizan protección anti CORS, de la que hablaremos más adelante.

10.3 PETICIONES AJAX

10.3.1 USO DE UNA API PARA ACCEDER A UN SERVICIO WEB

API es el acrónimo de ***Application Programming Interface,*** interfaz de programación de aplicaciones. Se trata del conjunto de métodos y otros elementos, que un servicio pone a disposición de las aplicaciones para comunicarse con él.

En el caso de AJAX, es muy habitual que servicios públicos de Internet ofrezcan datos para ser usados en las aplicaciones web, pero exijan una forma concreta de comunicarse. Esas especificaciones son la API del servicio. Lo que la API ofrece, a cambio de usarse correctamente, es un conjunto de datos que, hoy en día, suele estar en formato JSON.

Es posible que el servicio esté disponible a cambio de una suscripción, en cuyo caso se nos pedirá autentificarnos antes de resolver la petición de datos. Esa autentificación suele implicar el envío de claves o tokens al servidor.

10.3.2 CORS

CORS es el acrónimo de ***Cross-Origin Resource Sharing,*** Intercambio de Recursos de Origen cruzado. Las peticiones AjAX son fáciles de automatizar, lo que permitiría realizar cientos o miles de peticiones con unas pocas líneas de código, lo que podrá provocar el colapso del servidor. Para que los servicios de Internet se protejan y solo resuelvan peticiones procedentes de dominios validados, se ideó una norma que ahora es parte del protocolo http. Los datos sobre CORS se colocan en la cabecera de los paquetes http y permiten una vía de comunicación entre el navegador y el servidor web (aunque la petición se haga mediante AjAX) que asegura que se cumplen las normas establecidas.

Una política CORS habitual es que solo se resuelvan peticiones procedentes del dominio en el que está el servicio. Pero hay políticas más sofisticadas. Software de servidor web como **Apache** o **nginx** tienen capacidad para modificar esas políticas.

La cabecera http que protege a los servidores de vulnerabilidades causadas por CORS es **Access-Control-Allow-Origin.** Si a esta directiva se le asigna el valor * (asterisco), entonces admite cualquier origen para las peticiones. Se pueden especificar, en su lugar, dominios concretos y así solo se admiten peticiones de estos dominios (serán dominios confiables). Hay otras cabeceras que permiten matizar aún más estas políticas.

10.3.3 APIS EN JAVASCRIPT PARA EL USO DE AJAX

El manejo de AJAX clásico consiste en trabajar con el objeto **XMLHttpRequest.** De hecho, la mayoría de aplicaciones web siguen realizando peticiones mediante ese objeto. En otros casos, no utilizan directamente el objeto sino que utilizan frameworks como **jQuery** que permiten utilizar funciones más sencillas que abstraen y facilitan la forma clásica de realizar estas peticiones.

Hace unos años los dos organismos de estándares de la web **(WHATWG** y **W3C)** incluyeron el uso del objeto XMLHttpRequest en el estándar HTML 5.

Pero, lo cierto es que la API clásica ha sido superada por otra mucho más poderosa y que es compatible con el uso de promesas, lo que la permite aprovechar las nuevas capacidades de JavaScript a la par que adapta el uso de AJAX a los servicios actuales e implementa un modelo de trabajo más sencillo y mantenible. Esta API se llama Fetch.

Fetch es un estándar vivo, forma parte de la norma actual, que se mejora continuamente. Actualmente, todos los navegadores actuales ya han implantado la API Fetch (salvo el obsoleto Internet Explorer).

Cuando decimos que Fetch es una API, lo que queremos decir es que Fetch añade al lenguaje JavaScript una serie de funciones, objetos y elementos de todo tipo que hay que conocer para realizar correctamente la labor de conectar con servicios de las redes. El funcionamiento de esta API será el objeto de estudio el resto de esta Unidad.

10.4 REALIZAR PETICIONES MEDIANTE FETCH

La interfaz de Fetch proporciona un método global que se llama precisamente **fetch.** Este método requiere, al menos, indicar la URL del destino de la petición. El resultado de fetch es una promesa, que se considera resuelta cuando se reciben resultados sin error del destino.

```
fetch(direccionServicio)
   .then(función que recibe el objeto respuesta si la petición
                finaliza bien)
   .catch(función que recibe el error producido durante la petición);
```

Ejemplo:

```
fetch("https://jorgesanchez.net/servicios/ni                    faleatorio.php")
.then(response=>{
      consolé.log(response.status);
})
.catch(error=>{
      consolé.log("Error: " + error);
});
```

Este código realiza una petición **http** de tipo GET a la dirección indicada, la cual proporciona un servicio que devuelve números N1F calculados de forma aleatoria y que se pueden usar para hacer pruebas en bases de datos y otras aplicaciones.

Como **fetch** retorna una promesa, el método **then** permite procesar el resultado. El resultado de la promesa es un objeto de respuesta que se suele conocer como ***response*** (aunque el nombre le podamos cambiar a voluntad) que representa el paquete (o paquetes) http que proporcionan los datos requeridos. Más adelante podremos profundizar en las posibilidades del objeto de respuesta. En este caso, salvo que haya un problema con el servicio, la propiedad **status** del objeto response devolverá 200, código http que significa Ok y ese será el mensaje que veremos en la consola.

El método catch recoge el error ocurrido si la petición no concluye bien. Observemos este código:

```
fetch("http://noexiste.com")
.then(response=>{
      consolé.log(response.status);
})
```

```
.catch(error=>{
        consolé.log("Error:              "+error);
});
```

Este código retornará un código de error porque la petición no se puede resolver. Concretamente el texto que aparece por consola será:

```
Error: TypeError: Failed to fetch
```

10.5 MANIPULAR LA RESPUESTA. OBJETO RESPONSE

Como hemos visto en el apartado anterior, cuando una petición http realizada con **fetch** finaliza correctamente, el método then recibe un objeto conocido como **objeto de respuesta** o **response.** Vamos a ver en este apartado las capacidades de este objeto para poder utilizar de la mejor forma posible los datos de respuesta de la petición realizada.

10.5.1 PROPIEDADES Y MÉTODOS DEL OBJETO DE RESPUESTA

Se enumeran en la siguiente tabla las propiedades y métodos de este objeto:

PROPIEDAD O MÉTODO	USO
headers	Obtiene un objeto que contiene las cabeceras http del paquete de respuesta.
body	Cuerpo de la respuesta http. Contiene los datos en sí de la respuesta.
status	Devuelve el código de respuesta de la petición http. Un valor de 200 indica respuesta correcta. Los códigos se corresponden a los estándares del protocolo http[1].
statusText	Devuelve un texto descriptivo del código de respuesta de la petición. Por ejemplo, para el código con valor ***200,*** devuelve ***Ok.***
ok	Con valor **true,** indica que la petición se resolvió sin problemas. Más concretamente, el valor true significa que la respuesta contiene un código de respuesta http que está entre los valores del **200** al **299.**
redirected	Con valor true, indica que la respuesta es el resultado de una redirección.
url	Devuelve la URL de la que procede la respuesta.

1 La lista de códigos http de respuesta está disponible en la URL: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

PROPIEDAD O MÉTODO	USO
type	Indica el tipo de respuesta de la petición. Posibles valores: ■ **basic:** Respuesta coherente con la petición original. Muestra todas las cabeceras salvo las relacionadas con cookies. ■ **cors:** Índica que la respuesta procede de un origen CORS admitido. Hay cabeceras que no se pueden ver. ■ **error:** Hay error de red. No se describe el error ni disponemos de cabeceras de la respuesta http. ■ **opaque:** Respuesta para una petición marcada como **no-cors;** es decir, sin usar CORS. Es una respuesta muy restringida que no permite leer la mayoría de cabeceras.
redirect(URL)	Método que redirige la respuesta a otra URL.
clone()	Clona la respuesta en otro objeto.
error()	Clona la respuesta generando un error de red.
text()	Método que sirve para obtener un flujo de texto de la respuesta. Devuelve una promesa que, cumplida, resuelve obteniendo los datos de la respuesta en formato texto.
jsonO	Método similar al anterior, pero que trata de convertir la respuesta en un objeto de tipo JSON. Para ello crea una nueva promesa donde, si finaliza de forma correcta, la resolución es dicho objeto JSON.
blobO	Genera una promesa que es resuelta como un objeto binario en el caso de que finalice correctamente.

10.5.2 DATOS DE LA RESPUESTA

Cuando se realiza una petición a un servicio, el resultado deseado siempre es una serie de datos. El formato de esos datos puede ser de muchos tipos. Hoy en día, lo habitual es que se utilice JSON para datos que contienen información etiquetada con metadatos, que son los habituales. Pero se pueden devolver en otros muchos formatos: XML, texto puro, datos binarios, HTML, etc.

10.5.2.1 TIPOS MIME

Los paquetes http disponen de una cabecera llamada **Content-Type** que permite indicar el tipo **MIME** de los datos que van en el paquete. Los tipos MIME ***(Multipurpose Internet Mail Extensions,* Extensiones de Propósito Múltiple para Correos de Internet)** son una convención aceptada por todos los servidores que indica el tipo de datos.

Su formato habitual se basa en indicar un tipo, seguido de un subtipo separados por una barra de dividir. Ejemplos de tipos MIME habituales usados en peticiones AJAX son:

- **text/plain.** Texto puro.

- **text/html.** Datos en formato HTML.
- **application/json.** Datos en formato JSON.
- **application/javascript.** Datos en formato JavaScript. Históricamente este tipo se usaba también para JSON.
- **application/xml.** Datos en formato XML.

Hay muchos más tipos MIME, la lista completa de tipos MIME es extensísima. La lista oficial la mantiene la IANA en la dirección:

https://www.iana.org/assignments/media-types/media-types.xhtml

10.5.3 PROCESAMIENTO DE LAS RESPUESTAS

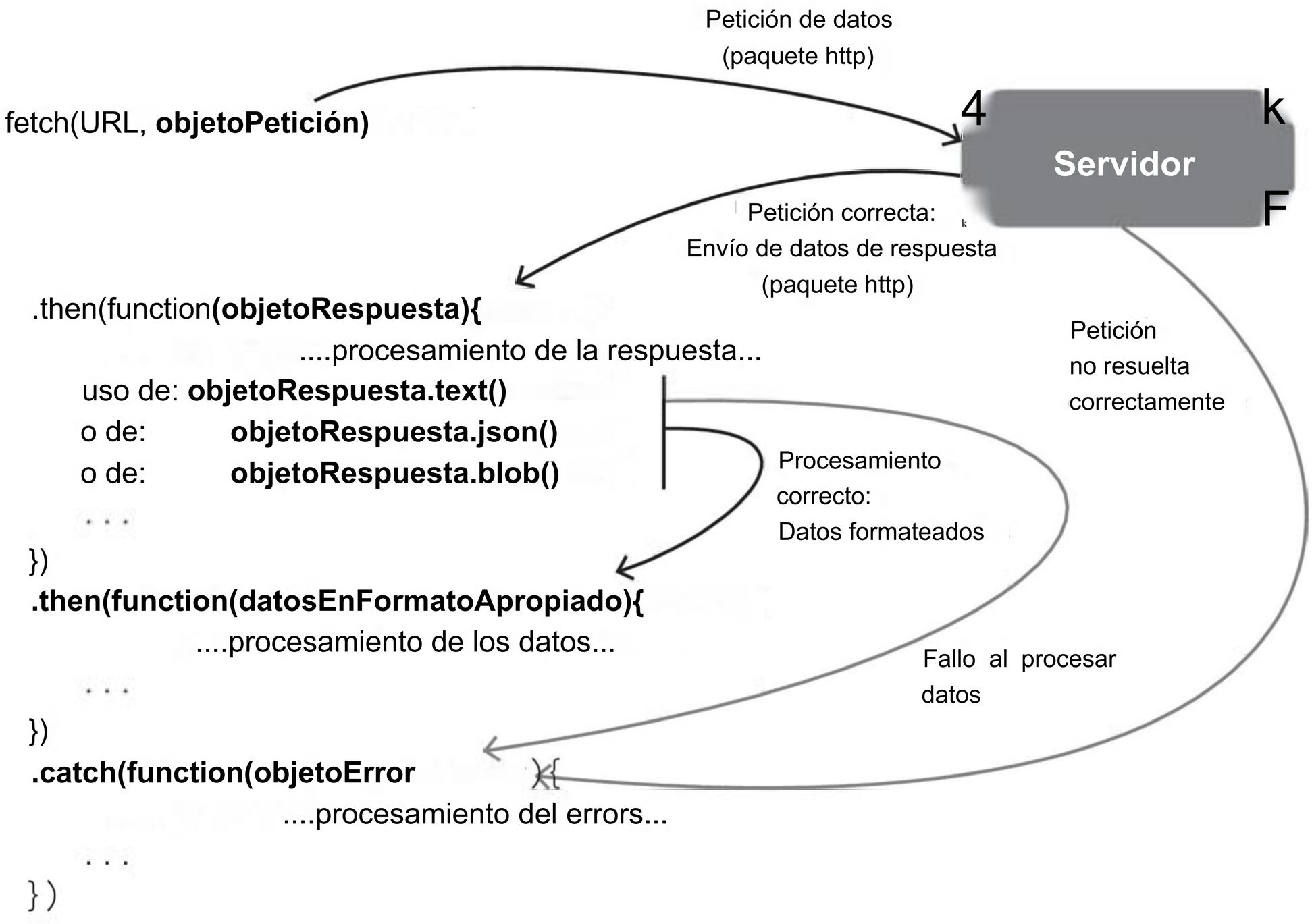


Figura 10.2: Proceso habitual de peticiones AJAX mediante el API Fetch

El proceso de la respuesta con la API Fetch sigue, normalmente, el siguiente proceso:

[1] Invocar a la función fetch indicando la URL a la que realizamos la petición y un objeto ***request*** (objeto de petición), si es el caso, en el que matizaremos aspectos avanzados de la petición como el método http a utilizar **(GET, POST, PUT, DELETE,** etc.). Del objeto de petición hablaremos más adelante.

[2] Invocamos de forma encadenada al método ***then*** el cual usa como parámetro una función callback que recibe como parámetro el objeto de respuesta.

[3] En la función callback del método **then** procesamos la respuesta obteniendo los datos en el formato deseado a través de los métodos que hacen esa labor: **text, json** o **blob.**

[4] Invocamos de forma encadenada a otro método **then** cuya función callback recibe el resultado de la conversión de datos realizada por los métodos text, json o blob (si los datos son correctos). En el cuerpo de esta función es donde realmente procesaremos los datos.

[5] Si las promesas anteriores fallan (falla la petición o el procesamiento de los datos) un método **catch,** a través de una función callback, permite procesar el error.

10.5.3.1 MANIPULAR RESPUESTAS EN FORMATO TEXTO

El tipo de datos más sencillo es el texto. Para entender cómo funcionan, en la dirección https://iorgesanchez.net/servicios/poesia.php se permite obtener el texto de un poema aleatorio. Cada vez que se realiza un petición de tipo GET a esa dirección, se obtiene un poema aleatorio (puesto que es aleatorio, incluso se puede volver a obtener el mismo).

Como veremos, el problema del texto plano es que no podemos delimitar los datos para clarificarlos. En el caso del poema, se entrega el autor, el título del poema y después los versos. Pero, como el texto no permite distinguir unos datos de otros, solo podremos saber el autor si nos fijamos en la respuesta y vemos que el autor aparece en la primera línea devuelta, el título en la siguiente y después, y hasta el final, aparece el texto del poema.

La API Fetch proporciona un método llamado **text** en el objeto de respuesta. El resultado de este método es el cuerpo de la respuesta pero ya formateada como texto plano. Por supuesto, solo en el caso de que el origen de la petición devuelva los datos como texto, podremos formatearlos a ese formato. De otro modo, no se cumplirá la promesa.

Por lo tanto, podemos crear una aplicación web que muestre un poema aleatorio obtenido de la URL comentada anteriormente, de esta forma:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Poesía del día</title>
</head>
<body>
<header>
        <hl>La poesía del día</hl>
</header>
<main>
        <pre>

        </pre>
</main>
```

```
<div id="error"></div>
<script>
    let pre=document.querySelector("pre");
    let capaError=document.getElementById("error");
    fetch("https://jorgesanchez.net/servicios/poesia.php")
        .then(respuesta=>respuesta.text())
        .then(texto=>{
            pre.innerHTML=texto;
        })
        .catch(error=>{capaError.innerText=error});
</script>
</body>
</html>
```

Lo interesante de esta aplicación es el código:

```
.then(respuesta=>respuesta.text());
```

Este código usa una función flecha que devuelve el resultado de la función **text,** la cual crea otra promesa. Si la promesa se cumple de forma exitosa, el texto ya interpretado en forma de texto se pasa a la segunda función **then.** Esa respuesta se usa para rellenar un elemento de tipo **pre** ya que, en este caso, es el ideal para respetar los saltos de línea originales del texto, fundamentales en el caso de la poesía.

10.5.3.2 MANIPULAR RESPUESTAS EN FORMATO JSON

El método JSON, del objeto de respuesta, genera una promesa que, en caso de resolverse positivamente, resuelve entregando los datos ya como objeto JSON. El formato JSON es el más utilizado actualmente para entregar datos a peticiones AJAX debido a su facilidad de manipulación desde JavaScript y a su versatilidad.

Como ejemplo, la dirección https://jorgesanchez.net/servicios/nifaleatorio.php permite obtener en formato JSON un N1F aleatorio (puede obtener una serie de ellos utilizando parámetros, como veremos más adelante). Este servicio está creado para hacer pruebas pero, evidentemente, no se pueden usar esos números para otra finalidad. Este podría ser un ejemplo de la respuesta del servicio:

```
[
   {
      "tipo": "dni",
      "numero": "74147019X"
   }
]
```

Como vemos, lo que devuelve es un array de objetos (aunque solo devuelve un elemento del array) en el cada elemento es un objeto con dos propiedades: tipo y número. Para crear una aplicación que requiere un N1F aleatorio y le muestre, el código sería:

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
        <meta charset="UTF-8">
        <title>NIF aleatorio</title>
</head>
<body>
<p id="nif"x/p>
<script>
var capaNIF=document.getElementById("nif");
fetch("https://jorgesanchez.net/servicios/ni faleatorio.php")
        .then(response=>{
                if(response.ok){
                        return response.Json();
                }
                else{
                        throw new Error("Los datos no llegaron bien");
                }
        })
        .then(1istaNIFs=>{
                capaNIF.textContent=listaNIFs[0]["numero"];
        })
        .catch(error=>{
                capaNIF.textContent="Error: "+error;
        });
</script>
</body>
</html>
```

Como podemos observar, el segundo método **then** recibe los datos ya en formato directamente manipulable desde JavaScript tras haber ejecutado el método **response.json().** Al parámetro de este segundo **then** le hemos llamado ***listaNIFs,*** ya sabiendo que efectivamente lo que se recibe es un array de objetos. Simplemente mostramos entonces la propiedad ***numero*** del primer elemento de dicho array.

10.5.3.3 MANIPULAR RESPUESTAS EN FORMATO BINARIO

Los datos de tipo **Blob** ***(Binary Large OBjects)*** se corresponden a datos inmutables procedentes de ficheros. Se utiliza cuando la respuesta a una petición son datos procedentes de ficheros. El caso más típico es recoger una imagen de un servidor que es capaz de enviarla como respuesta a una petición http.

Es el caso de **Unsplash,** servicio muy popular de Internet para imágenes de alta calidad que permite obtener una imagen aleatoria de la dirección usando la siguiente URL:

https://source.unsplash.com/random

Normalmente este tipo de peticionesse rechazan cuando son de tipo CORS. Pero en este caso, el servicio **unsplash** sí las acepta.

La respuesta se debe procesar con el método **blob()** del objeto de respuesta, el cual genera un objeto de tipo **Bíob.** Estos objetos disponen de varios métodos. El más interesante es **type** que obtiene el tipo MIME del objeto, por ejemplo: ***imagen/jpeg.*** Por otro lado el objeto predefinido y disponible desde JavaScript, **URL,** dispone de un método llamado **createObjectURL** al que se le pasa un objeto Blob y devuelve un texto que representa la URL virtual con la que podemos acceder a ese recurso como si fuera una imagen que se carga desde Internet. De esa forma, se permite acceder a la imagen desde el propio código HTML.

Veamos como poner en práctica todo lo comentado:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Imagen del día</title>
</head>
<body>
<main>
        <p>Esperando datos...</p>
</main>

<script>
var main=document.querySelector("main"); //Capa main
fetch("https://source.unsplash.com/random")
        .then(response=>{
                if(response.ok){
                        return response.blob();
                }
                else{
                        throw new Error("Los datos no llegaron bien");
                }
        })
        .then(blob=>{
                consolé.log(blob.type);
                let imagen=document.createElement("img");//nuevo elemento img
                //Asignamos al atributo src del elemento creado, la URL
                //virtual de la imagen descargada
                imagen.setAttribute("src",URL.createObjectURL(blob));
                main.innerHTML="";//borramos mensaje de "Esperando"
                main.appendChild(imagen);//colocamos la imagen en la capa main
        })
        .catch(error=>{
                main.textContent=error;
        });
</script>
</body>
```

```
</html>
```

El primer método **then** comprueba que el código de respuesta es ***Ok*** y usa el método blob para generar una promesa que obtenga el objeto **Blob** devuelto por **Unsplash.**

El segundo método **then** recoge ese objeto, una vez que se cumple la promesa, y por consola escribirá el tipo MIME del objeto (debería ser ***imagen/jpeg).*** Además, crea un nuevo elemento de tipo **img** y modifica el atributo **src** de ese elemento para que recoja la URL que representa al objeto Blob obtenido. Para ello, se usa **URL.createObjectURL(blob).** Antes de colocar la imagen se borra el contenido de la capa main **(main.innerHTML="")** para que desaparezca el mensaje con el texto ***Esperando datos....*** Inmediatamente después añadimos la imagen a la capa main.

10.6 PERSONALIZAR LA PETICIÓN. OBJETO REQUEST

Cuando se realizan peticiones mediante el método **fetch,** además de la URL, podemos pasar como parámetro un objeto de tipo Request. Este objeto permite modificar el paquete http que se envía con la petición para que recoja aspectos que nos interesan y que la petición sea lo más ajustada posible a nuestros intereses.

10.6.1 PROPIEDADES Y MÉTODOS DE REQUEST

Las propiedades de las que disponemos en los objetos request son:

PROPIEDAD	USO
url	Propiedad obligatoria que contiene la URL destino de la petición.
method	Método http que se usará en la petición. Es el comando http de la petición. Normalmente se usa **GET** o **POST,** pero es válido cualquier comando http: **PUT, DELETE, HEADER, DELETE,** etc. Por defecto el método que se usa es GET.
headers	Permite indicar un objeto de tipo **Headers** que permite modificar las cabeceras http de la petición.
mode	Indica si se usa CORS en la petición. Posibilidades: ■ **same-origin.** Solo se acepta respuesta del mismo dominio en el que se ha realizado la petición. Si la petición se dirige a otro dominio, se devuelve un error. ■ **cors.** Se permiten respuestas de dominios cruzados. Es el valor por defecto, ■ **no-cors.** Solo se admiten métodos GET, POST o PUT y limita las cabeceras http que se pueden usar. ■ **navigate.** Índica que la petición se usa como URL del navegador y no para ser usada como parte de una petición AJAX.

PROPIEDAD	USO
cache	Índica el modo de caché de la respuesta. Los navegadores cachean las respuestas a peticiones previas para acelerar la navegación. Esta propiedad permite calibrar cómo deseamos el cacheado. Posibilidades: ■ **default.** Valor habitual. El navegador comprueba si la respuesta a la petición está en caché. Si es así y el caché es reciente, no realiza realmente la petición y recoge los datos del caché. ■ **no-store.** Obliga siempre a traer la respuesta del servidor. No cachea ninguna respuesta. ■ **reload.** Obliga a traer la respuesta del servidor y no de la caché, pero la respuesta se guarda en caché por si se desea usar en las siguientes peticiones, ■ **no-cache.** Si la respuesta a la petición ya sa había cacheado, pide al servidor una comprobación de si la respuesta es la misma, si es así se toma de la caché. ■ **force-cache.** Fuerza a usar la caché. Si hay respuesta a esa petición en la caché, se usa. Si no, se pide al servidor y se cachea para la siguiente vez. ■ **only-if-cached.** Solo se admiten los datos si proceden de la caché. En otro caso, si la respuesta no está cacheada, devuelve un código 504 de petición.
redirect	Indica cómo se deben de procesar las respuestas en el caso de que procedan de redirecciones. Posibilidades: ■ **follow.** Se siguen las redirecciones sin problemas. ■ **error.** Si hay redirecciones, se retorna un error. ■ **manual.** Se manejan de manera manual por el usuario (debería hacer un clic sobre la respuesta antes de la redirección). En peticiones AJAX, solo las dos primeras nos interesan.
credentials	Permite especificar si se admiten cookies en la petición. Posibilidades: ■ **omit.** Nunca se envían ni reciben cookies. ■ **same-origin.** Es el valor por defecto, se admiten si el origen y el destino de la petición están en el mismo dominio. ■ **inelude.** Siempre se admiten cookies.
integrity	Almacena una cadena de integridad creada con un algoritmo hash de criptografía (se admite **sha256, sha384** y **sha512),** para validar la integridad de la petición.

Ejemplo de petición usando objeto de tipo ***request:***

```
fetch("http://direccionnovalida.com/",{
    method:"POST",
    mode:"cors",
    cache:"no-cache"
})...
```

Además los objetos request disponen de los mismos métodos, prácticamente, que los objetos response: **json(), blob(), text(), clone(),** etc. Sin embargo, no suele ser necesario su uso porque lo que hacen es procesar el cuerpo de la petición y ese proceso se requiere de las respuestas, no de las peticiones.

10.6.2 ESTABLECER LA CABECERA. OBJETO HEADERS

La propiedad **headers** de la petición (objeto **request)** permite establecer todas las cabeceras http que nos proporciona el protocolo. Tanto el objeto de respuesta como el de petición poseen una cabecera de tipo headers.

Para crear objetos de tipo **Headers** disponemos de varios constructores. Por ejemplo, podemos indicar un objeto que permite indicar propiedad http y valor de la misma:

```
var cabecera=new Headers({
        "Content-Type":"application/json",
        "Accept-Charset":"utf-8"
});
```

Además, los objetos de cabecera disponen de estos métodos:

PROPIEDAD	USO
append	Añade una entrada al objeto de cabecera. Es decir, añade una nueva cabecera http. Para ello recibe el nombre de la cabecera y su valor. Ejemplo: objeto.**append('Content-Type', 'application/json');** Si el nombre de cabecera http no es correcto, se provoca una excepción de tipo **TypeError.**
delete	Retira una cabecera del objeto de cabeceras. Ejemplo: objeto.**delete('Content-Type');**
get	Devuelve el valor de una cabecera en el objeto. Por ejemplo: objeto.**get('Content-Type');** Podría retornar, por ejemplo: ***'application/json'.*** Si la cabecera de la que se pide el valor no está en el objeto, devuelve **nuil.**

PROPIEDAD	USO
set	Modifica una cabecera existente y la otorga un nuevo valor: **objeto.set (' Content-Type' , ' text/xml);** Si la cabecera no existe, se añade (como hace **append).** Si el nombre de cabecera http no es válido, se provoca una excepción de tipo **TypeError.**
has	Método que comprueba si el objeto de cabeceras contiene una entrada concreta. De ser así, devuelve **true.** **objeto. has([1] Content-Type');** Si el objeto tiene esa cabecera, devuelve **true.**
entries	Devuelve un objeto iterable que permite recorrer todas las cabeceras del objeto: **for(let** [clave,valor] **of** objetoHeader){ consolé.log(clave+", "+valor); }

Hay que tener en cuenta que también las respuestas poseen este mismo objeto y que, lógicamente, sus propiedades y métodos son los mismos.

10.7 ENVIAR DATOS CON LA PETICIÓN

En muchas ocasiones, los servidores de Internet que otorgan servicios, requieren o permiten enviar datos para determinar de una forma más ajustada los datos que debe devolver. De forma clásica, esos datos son pares nombre/valor, listas parámetros con valores concretos. Pero actualmente se admite enviar datos incluso en formato de texto JSON.

Los documentos HTML siempre han permitido el envío de este tipo de datos a servidores externos a través de los formularios. De manera clásica, los datos se envían usando los comandos http GET o POST. El método GET envía los datos en la URL y POST lo hace en el cuerpo del mensaje http. Esa diferencia puede hacer pensar que POST es más seguro al ocultar más los datos que envía, pero realmente no lo es. Los datos viajan de forma plana y solo si los ciframos estarán realmente protegidos.

Actualmente, los comandos http se asocian a verbos que requieren un tipo concreto de acción. Así **GET** se asocia a una petición de datos y **POST** a un envío de datos a un servidor. Otros comandos http son: PUT (modificación de datos), DELETE (borrado de datos), PATCH (modificación parcial de datos) o HEAD (petición de cabecera).

Evidentemente hay que conocer la API del servicio final para saber qué método usar, cómo enviar los datos y cómo debemos recibirlos, dependiendo de lo que el servicio es capaz de aceptar.

10.7.1 ENVÍO DE DATOS USANDO GET

En el caso de las peticiones **GET,** se envían los parámetros en la URL en la parte conocida como **cadena de consulta.**

Por ejemplo, en el caso del servicio que vimos en apartados anteriores y que permite mostrar una lista de números de identificación fiscal españoles (N1F) aleatoria. Si la petición se hace sin enviar parámetros devuelve un solo N1F. Pero, si indicamos un parámetro llamado ***n*** con un valor positivo, por ejemplo **10,** se nos devolverán diez NIFs aleatorios. Además, un segundo parámetros llamado ***tipo*** admite los valores: ***dni, nie*** o ***todo.*** Si indicamos el valor ***nie*** en el parámetro ***tipo*** se nos devolverán diez números de identificación de extranjeros (NIE) producidos de forma aleatoria:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>NIEs aleatorios</title>
</head>
<body>
<main>
        <ul>

        </ul>
</main>
<script>
        let 1ista=document.querySelector("ul");
        fetch("https://jorgesanchez.net/servicios/ni faleatorio.php" +
                        " ?n=10&tipo=nie",{
                method:"get",
                mode:"cors"
        })
        .then((resultado)=>resultado.json())
        .then(datos=>{
                for(let nie of datos){
                        let 1i=document.createElement("1 i");
                        1 i.textContent=nie.numero;
                        lista.appendChild(1 i);
                }
        })
        .catch((error)=>{
                document.querySelector("main").textContent=
                        "Error: "+error;
        });
</script>
</body>
</html>
```

En caso de que la petición se resuelva correctamente, se reciben los 10 números en formato jSON. Tras procesarlos, se muestran en forma de lista.

10.7.2 ENVÍO DE DATOS DE FORMULARIO

10.7.2.1 CODIFICACIÓN ESTILO URL

En el caso de POST, los datos no se envían en la URL sino en el cuerpo del propio paquete http. En el caso clásico, los datos POST proceden de datos de formulario e incluso aunque no procedan, debemos marcar el paquete para indicar que los datos se envían al estilo de los formularios. Esa forma usa un formato similar al del método GET. En las peticiones AJAX se debe asociar esos valores a la propiedad **body.** Por ejemplo:

```
fetch("https://Jorgesanchez.net/servicios/validarNIF.php",{
        method:"post",
        headers: {'Content-Type':'application/x-www-form-urlencoded'},
        body:"n if=12345678A&t i po=dn i"
})
```

En el código anterior, lo interesante es la cabecera:

```
'Content-Type':'application/x-www-form-urlencoded'
```

Esta cabecera es la que permite indicar que los datos usan la forma: **parámetro=valor** y que se encadenan los valores mediante el carácter ampersand: &. De esa forma, se reconocerán los parámetros **nif** y **tipo** de forma apropiada en el servicio final. Suponiendo, eso sí, que este servicio efectivamente esté esperando los datos en ese formato.

La forma en la que se envían los datos es la que aparece en el código anterior:

```
body:"nif=12345678A&tipo=dni"
```

10.7.2.2 USO DE OBJETOS FORMDATA

Hay otra forma de enviar datos de formulario que es el que se asocia al tipo MIME: **application/multipart/form-data.** Es un estilo de envío de datos por pares pensado para transmitir datos de gran tamaño o de tipo binario. Si queremos enviar, por ejemplo, un archivo, este es el formato apropiado. No obstante, se puede utilizar para enviar datos más sencillos.

JavaScript proporciona un objeto llamado **FormData** que facilita la preparación de los datos en este formato. La creación de un objeto de este tipo se hace de esta forma:

```
var form=new FormData();
```

Añadir un nuevo par parámetro/clave se hace con el método **append:**

```
form.append("nif","12345678A");
form.append("tipo","dni");
```

Si quisiéramos borrar un parámetro:

```
form.delete("tipo");
```

Si quiséramos modificar:

```
form.set("ni f","98765432B");
```

Saber si hemos grabado un valor lo hace el método has, que devuelve **true** o **false.**

```
consolé.log(form.has("nif"); //Escribe true
```

Obtener el valor ya grabado de un parámetro se hace con **get** (si el parámetro no existe, devuelve **nuil)**

```
consolé.log(form.get("ni f");          //Escribe 98765432
```

Finalmente, podemos recorrer todos los parámetros del objeto, por ejemplo, con el bucle **for.. of:**

```
<script>
let form=new FormData();
form.append("nombre","Jorge");
form.append("edad","47");
form.append("profesión","profesor");
for(let [dato,valor] of form){
      consolé.log('${dato}=${valor}');
}
</script>
```

En definitiva, es un objeto con una interfaz muy parecida a otros objetos que representan conjuntos clave/valor como el ya comentado objeto **Headers.**

El servicio https://jorgesanchez.net/servicios/validarNlF.php permite enviar, vía POST, un N1F y entonces devuelve, en formato JSON, un objeto con una única propiedad llamada ***error.*** A su vez, error tiene dos propiedades: ***codigo*** que devuelve el código de error (vale cero si no hay error) y ***mensaje*** que contiene el mensaje de error. Por lo tanto, el código para validar el N1F (incorrecto) ***123456789A*** usando ese servicio es el siguiente:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="UTF-     8" >
      <title>Validar >NIF</title>
</head>
<body>
<main>
      <P> </p>
</main>
<script>
      let p=document.querySelector("p");
      let form=new FormData();
      form.append("ni f","12345678A");
      fetch("https://jorgesanchez.net/servicios/validarNIF.php",{
            method:"post",
            body:form
      })
      .then((resultado)=>resultado.json())
```

```
    .then(datos=>{
r               p.textContent=datos.error.mensaje;
        })
        .catch((error)=>{
                document.querySelector("main").textContent=
                        "Error: "+error;
        });
</script>
</body>
</html>
```

Los objetos FormData se pueden construir a partir de los controles de un formulario. De esta forma se pueden recoger de golpe todos los controles del formulario. Basta con usar esta sintaxis:

```
new FormData(elementoForm);
```

El siguiente código crea una aplicación web en la que podremos escribir un N1F en un control de texto y, al pulsar un botón ***(Comprobar)*** se comprueba si es válido o no:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8”>
        <title>Validación de NIF</title>
        <style>
                body{
                        text-align: center;
                }
        </style>
</head>
<body>
        <form action="https://jorgesanchez.net/servicios/validarNIF.php"
                method="POST">
        <label for="nif">Escriba el NIF que desea val idar</label><br>
        <input type="text" id="nif" name="nif"> <br>
        <button>Comprobar</button>
</form>
<div id="resultado">
</div>
<script>
        var form=document.querySelector("form");
        var capaRes=document.getElementById("resultado");
        var tNIF=document.getElementById("nif");
        form.addEventListener("submit",(ev)=>{
                ev.preventDefault();
                let data=new FormData(form);
                fetch(form.getAttribute("action"),{
```

```
                method:form.getAttribute("method"),
                body:data
            })
            .then(respuesta=>respuesta.json())
            .then(datos=>{
                capaRes.textContent=datos.error.mensaje;
            })
            .catch(err=>{
                capaRes.textContent="Error: "+err;
            })
    });
    tNIF.addEventListener("focus",ev=>{
                capaRes.textContent="";
                tNIF.selectionStart=0;
                tNIF.selectionEnd=tNIF.valué.length;
            });
        </script>
</body>
</html>
```

En este código, el método fetch usa los propios atributos del formulario para determinar el destino de la petición y el método de paso. Eso permite que sea la etiqueta form la que controle los detalles del destino.

El evento submit se ha capturado obligando a que no se envíen directamente los datos al destino (mediante el método de evento **preventDefault)** y haciendo la petición fetch que recoge los datos JSON del destino y les muestra en una capa **(capaRes).**

Para que el manejo del usuario de esta página sea más cómoda y permita validar muchos más números, cuando el usuario se disponga a escribir un nuevo N1F, se borra el mensaje anterior y se selecciona el texto del cuadro (métodos **setSelectionStart** y **setSelectionEnd)** para facilitar su borrado.

10.7.3 ENVÍO DE PARÁMETROS EN FORMATO JSON

No siempre el servicio destino de nuestra petición usa los datos en el formato clásico de los formularios. Hay servicios que aceptan los datos en formato JSON. Para estos servicios, lo que tenemos que hacer es preparar los datos a enviar en un objeto y después convertirlo a formato de texto JSON mediante el método **JSON.stringify.**

Para probar este tipo de envío podemos usar un servicio disponible en la URL:

https://jsonplaceholder.typicode.com/

Esta dirección permite, precisamente hacer pruebas de peticiones simulando un sistema de mensajes ***(posts)*** creados por usuarios. Podemos hacer peticiones para obtener datos, añadir datos, modificar, etc. En la URL anterior vienen las instrucciones de uso.

Por ejemplo, mediante una petición de tipo GET podemos obtener la lista de todos los usuarios, utilizando https://isonplaceholder.typicode.com/posts

Si añadimos el número de post: https://jsonplaceholder.typicode.eom/posts/5 nos devuelve el post número 5 (hay 100 para probar).

Mediante peticiones de tipo POST podemos añadir nuevos datos. En realidad no se añaden (es un servicio par practicar, no es real), pero parece como si sí lo hicieran. Por ejemplo, este código hace como si el usuario número 5 añadiera un nuevo post con el título ***"Mi mensaje"*** y el texto ***"Este es un mensaje de prueba"***

```
let data={
        title:"Mi mensaje",
        body:"Mensaje de prueba",
        userld:5
}

fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        body: JSON.stringi fy(data),
        headers: {
                "Content-type": "application/json;   charset=UTF- 8"
        }
    })
    .then(resp => resp.json());
    .then(json          =>          {consolé.log(json)});
    .catch(err => {consolé.log(err)});
```

El resultado de la petición es:

```
{
   id: 101,
   title: 'foo',
   body: 'bar',
   userld: 1
}
```

Realmente no hay un post con el número 101, se nos muestran esos datos simulando la que daría un servidor real.

Lo interesante es que este servidor ha recibido los datos en formato JSON mediante la función stringify. Para saber si hay que enviarlos de esta forma o en la forma de los formularios ***(FormData)*** necesitamos conocer el funcionamiento del servidor destino de nuestras peticiones.

10.8 USO DE AWAIT/ASYNC CON FETCH

Puesto que la API Fetch es compatible con promesas, es posible manejarla mediante la notación **await/async** del estándar ES2017. Esta notación, para muchos desarrolladores, es más legi-

ble y facilita mucho el mantenimiento del código. El ejemplo visto en el apartado anterior para enviar datos de un supuesto POST al servicio de pruebas tripicode, sería, en esta notación, de esta forma:

```
let data={
        title:"Mi mensaje",
        body:"Mensaje de prueba",
        userld:5
}
async      function      peticion(){
    try{
        const resp=await
                                    fetch('https://jsonplaceholder.typicode.com/posts',   {
                                    method: 'POST',
                                    body: JSON.stringify(data),
                                    headers: {
                                        "Content-type":  "application/json; charset=UTF-8"
                                    }
                                });
        const json=await resp.json();
        consolé.log(json)
    }
    catch(err){
        consolé.log(err);
    }
}
peticion();
```

Si comparamos este código con el resto de conexiones realizadas en esta unidad, lo que antes suponía usar el método **then,** ahora son sentencias **await.** La captura de errores no requiere del método **catch,** si no de la estructura **try..catch.**

Puesto que esta notación está ya muy implantada, es cuestión de gustos cuál de las dos notaciones utilizar para nuestras peticiones

10.9 PRÁCTICAS SOLUCIONADAS

Práctica 10.1: Validación de NIF

- Crea un formulario que permita escribir un nombre, apellidos, email y un NIF, con una estética similar a la de la siguiente imagen:



Figura 10.3: Panel inicial de registro del usuario

- Inicialmente los cuadros de entrada de texto poseerán un borde rojo (para indica que el contenido aún no es válido).
- El nombre y apellidos se consideran válidos si al menos hay dos letras.
- El email si hay un símbolo de arroba (@) y a su izquierda y derecha, al menos una letra.
- Se hacen las validaciones a medida que vamos escribiendo, con cada tecla pulsada.
- La validación del NIF utiliza el servicio https://jorgesanchez.net/servicios/validarNIF.php el cual recibe usando el comando http POST, un parámetro llamado nif (en minúsculas) con el NIF a validar. Responde en forma JSON con un objeto llamado error, el cual indica en su propiedad **codigo** (será cero si no hay error), el código del error y en la propiedad **mensaje** el mensaje de error.
- El botón de **Comprobar** permanece desactivado hasta que todos los cuadros de texto sean válidos. Cuando lo sean, el botón se activa.
- Al hacer clic en el botón **Comprobar,** simplemente aparecerá un mensaje indicando que todo es correcto.
- Al hacer clic en el mensaje, comenzamos de nuevo.

SOLUCIÓN: PRÁCTICA 10.1

El código de la solución es extenso. Vamos a explicarlo archivo a archivo.

En primer lugar, el código HTML (sería el archivo **index.html)** simplemente tiene los controles de formulario, carga el CSS (en el directorio **css,** el archivo **estilos.css)** y carga el JavaScript (en el directorio **js,** el archivo **accion.js).** Además, contiene los elementos de formulario y una capa para mostrar los errores. Para mostrar el mensaje final de datos correctos, tenemos una capa llamada **telón** que, a través de CSS, tapará toda la página con color negro al 50% de opacidad. Encima de esta capa se muestra la capa **mensaje.** Estas dos capas inicialmente están ocultas. El código de **index.html** es:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-      8n >
        <meta name="viewport" content="width=device-width,
initial-scale=l,0">
        <meta http-equiv="X-UA-Compatible" contenta"ie=edge">
        <title>Registrar usuario</title>
        <1ink rel="stylesheet" href="css/estilos.css">
</head>
<body>
        <main>
                <hl>Registrar usuario</hl>
                <form action="#" autocomplete="off">
                        <label for="nombre">Nombre</labe1><br>
                        < input type="text" id="nombre" name="nombre"> <br>
                        <label for="apel1 idos">Ape11 idos</label> <br>
                        < input type="text" id="apel1 idos" name="apel1 idos"> <br>
                        <label for="emai1">Erna i 1</label><br>
                        <input type="text" id="email" name="emai1"><br>
                        <label for="NIF">NIF</label><br>
                        <input type="text" id="NIF" name="NIF"><br>
                        <button id="boton" disabled>Enviar</button>
                </form>
        </main>
        <div id="error">

        </div>
        <div id="telon"></div>
        <div id="mensaje">
                <p>Los datos se han enviado correctamente</p>
        </div>
        <script src="js/accion.js"></script>
</body>
</html>
```

El archivo **estilos.css** contiene este código:

```
main{
        width:100%;
        max-width:400px;
        margin:auto;
}
hl{
        text-align: center;
        font-family: sans-serif;
}
label{
        text-align: left;
        font-family: sans-serif;
        color:gray;
}
input{
        width:97%;
        font-size:1.2em;
        padding:.2em;
        marg i n-bottom: .5em;
        border-rad i us:5px;
        border:lpx solid red;
        outline:none;
}
buttonf
        width:100%;
        font-size:1.5em;
        border:none;
        background-color: gray;
        padding:.4em;
        color:white;
        border-radius: 10px;
}
button:disabled{
        background-color: 1ightgray;
        color:white;
}

/* Capa que pone borde verde a los controles
cuando su contenido es válido */
 .verde{
        border-color: green;
}
```

```
/* Capa que muestra los errores */
«error{
        color:red;
        font-family: sans-serif;
        margin-top:3em;
}
/* Capa inicialmente oculta
Cuando se muestra desvanece el formulario*/
#telon{
        position: fixed;
        left:0;
        top:0;
        width:100%;
        height:100%;
        z-index:100;
        background-color: black;
        opacity: .5;
        display:none;
}
/* Capa por encima de la anterior
inicialmente oculta con el mensaje
de envío de datos correctos*/
#mensaje{
        position: fixed;
        left:50%;
        margin-left:-150px;
        margin-top:-100px;
        top:50%;
        width:300px;
        height:200px;
        z-index:200;
        background-color:white;
        display:none;
}
«mensaje p{
        font-weight: bold;
        font-size:2em;
        text-align: center;
        font-family: sans-serif;
}
```

Podemos observar que, además del CSS de maquetación de los elementos del formulario, disponemos de una clase llamada **verde** para dar borde verde a los controles cuando son válidos. Las capas de mensaje y telón se posicionan por encima del formulario dejando la apariencia inicial tapada por el fondo negro de la capa de telón.

El código JavaScript está dividido en funciones. La petición AJAX la gestiona la función **validarNIF.** El resto de validaciones tienen también funciones dedicadas. Muchas de ellas podrían ser validadas directamente por HTML, ya que la propiedad **pattem** admite una expresión regular y la propiedad **type** con valor **"email"** ya valida el email. Pero JavaScript admite una interactividad extra porque el usuario puede ver en caliente, a medida que escribe, si lo que hace es válido o no.

El código del archivo accion.js es este:

```
/**
 *  Valida que al menos haya dos caracteres
 *  alfabéticos en el elemento
 *  (usa caracteres de lenguas españolas)
 *  @param {Element} elemento
 */
 function  validarNombre(elemento){
       //La  expresión        Unicode:       /^\[p{Letter}]{2}/u
        //es  mejor para validar, pero aun no es
      //compatible con todos los navegadores
      if(/^[A-Za-zñÑgQáéíóúüÁÉÍÓÚÜüÁÉÍÓÜáéióú]{2}/
                                          .test(elemento.valué)){
            elemento.classList.add("verde");
      }
      else{
            elemento.classList.remove("verde");
      }
      comprobarTodoValido();
} **
 *  Valida que el Elemento contiene como
 *  valor un Email válido
 *  Solo mira que haya alguna letra, una símbolo @
 *  y otra letra
 *  @param {Element} elemento
 */
function validarEmai1(elemento){
       if(/.*[A-Za-z].*[A-Za-z].*/.test(elemento.valué)){
            elemento.classList.add("verde");
       }
       else{
            elemento.classList.remove("verde");
       }
       comprobarTodoValido();
}
```

```
/**
 *  Acude al servicio de internet de validación
 *  de NIFs para validar la corrección del mismo
 *  ©param {Element} elemento
 */
function validarNIF(elemento){
        //Preparación del parámetro
        let formData=new FormData();
        formData.append("ni f",elemento.valué);

        fetch('https://jorgesanchez.net/servicios/validarNIF.php',{
                method:"POST",
                body:formData

        })
        .then(resp=>{
                return resp.json();
        })
        .then(datos=>{
                if(datos.error.codigo==0){
                        elemento.classList.add("verde");
                }
                else{
                        elemento.classList.remove("verde");
                }
                comprobarTodoValido();

        })
        .catch(error=>{
                document.getElementById("error").textContent=error;
                comprobarTodoValido();
        })
}

/ **
 *  Comprueba si todo el formulario es válido
 *  para activar el botón
 */
function comprobarTodoValido(){
        if(
                nombre.classList.contains("verde") &&
                apel1 idos.classList.contains("verde")          &&
                email.classList.contains("verde") &&
                NIF.classList.contains("verde")
        ){
```

```
            //todo es correcto
            boton.disabled=false;
        }
        else{
            boton.disabled=true;
        }
}

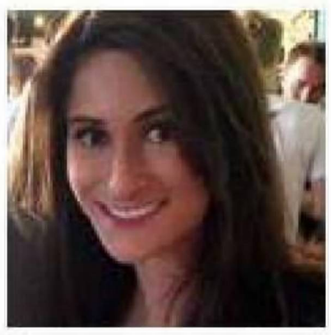
window.addEventListener("load",(ev)=>{
        var nombre=document.getElementById("nombre");
        var apel1idos=document.getElementById("apel1 idos");
        var email=document.getElementById("email");
        var NIF=document.getElementById("NIF");
        var boton=document.getElementById("boton");
        var telon=document.getElementById("telón");
        var mensaje=document.getElementById("mensaje");

        nombre.addEventListener("keyup",(ev)=>{
            val idarNombre(nombre);
        });
        apel1 idos.addEventListener("keyup",(ev)=>{
            val idarNombre(apel1 idos);
        });
        emai1.addEventListener("keyup",(ev)=>{
            val idarEmai1(emai1);
        });
        NIF.addEventListener("keyup", (ev)=>{
            validarNIF(NIF);
        });

        boton.addEventListener("click",(ev)=>{
            ev.preventDefault();
            //Damos por hecho que si el botón está
            //activo, los datos son válidos
            telón.style.display="block";
            mensaje.style.display="block";
        });
        //Con el clic en el telón o en el mensaje,
        //empezamos de nuevo
        telón.addEventListener("elick",(ev)=>{
            location="Índex.html";
        });
        mensaje.addEventListener(("elick",(ev)=>{
            location="index.html";
        });
});
```

Práctica 10.2: Mostrar un usuario aleatorio

- La página https://randomuser.me/ permite obtener datos aleatorios de personas pensando en que los desarrolladores y otros profesionales puedan utilizarlos en sus pruebas y test.
- Las instrucciones de la API de este servicio gratuito están en la URL: https://randomuser.me/documentation
- En todo caso la idea es hacer peticiones vía GET a la URL: ht tp s://r ando mus e r. me/api/
- Se pueden pasar parámetros para indicar cuántos usuarios aleatorios deseamos, el sexo, política de contraseñas, páginas, formato de respuesta, etc.
- En la página de documentación viene un ejemplo de la estructura]SON de las respuestas. Como resumen indicamos que es un objeto formado por dos propiedades: ***results*** e ***info.*** La primera es un array donde cada elemento lo forma un objeto con los datos del usuario aleatorio. La propiedad ***info*** contiene otros detalles entre los que destaca una semilla que permite repetir una petición con los mismos datos y datos de paginación (cuando deseamos dividir en páginas los resultados se debe usar la misma semilla).
- La aplicación mostrará la foto, nombre, apellido, email, dirección y estado al que pertenece el usuario. Cada vez que actualicemos la página, se pedirá otro usuario. Ejemplo de resultado:



laerke sorensen
Email: lserke.s0rensen@example.com
201 msllevaenget
nykobing sj. (MIDTJYLLAND)

Figura 10.4: Ejemplo de usuaria aleatoria

SOLUCIÓN: PRÁCTICA 10.2

La API **Randomuser** lo hace casi todo. Lo único que debemos programar es cómo recoger los datos que deseamos y cómo mostrarlos. El código HTML podría ser este:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Usuario aleatorio</title>
        <1ink rel="stylesheet" href="css/estilos.css">
```

```
</head>
<body>
<main> </main>
<script src="js/accion.js"></script>
</body>
</html>
```

Simplemente, cargamos el JavaScript y el CSS y dejamos un elemento **main** para que sea allí donde dejamos los datos.

Hoja de estilos **(css/estilos.css):**

```
*{
        font-family: sans-seri f;
}
main{
        text-align: center;
}
p{
        margin:0;
}
.error{
        background-color:red;
        color:white;
        font-weight: bold;
}
```

Finalmente, el código JavaScript que únicamente invoca al método **fetch** para realizar la petición, convierte la respuesta a JSON y luego localiza y coloca los datosdeseados:

```
var main=document.querySelector("main");

const url = 'https://randomuser.me/api/';
fetch(url)
    .then((resp) => resp.json())
    .then(data=>{
        let datos=data.results[0];
        let foto=datos.picture.large;
        let mail=datos.emai1;
        let nombre=datos.ñame.first;
        let apel1ido=datos.ñame.last;
        let calle=datos.location.street;
        let ciudad=datos.location.city;
        let estado=datos.location.state;

        main.innerHTML=
                ' <figure> ' +
                    '<img src=’${foto}’ alt='foto’>'+
```

```
            '</figure>' +
            '<p>${nombre} ${apel1 ido}</p>' +
            '<p>Email: ${mail}</p>' +
            '<p>${calle}</p>* +
            '${ciudad} (${estado. tollpperCase()} ) </p>';
})
 .catch(error=>{
    main.innerHTML='<p class='error'> ${ error}</p>';
});
```

Práctica 10.3: Mostrar 10 usuarios que cambian

- Usaremos el mismo servicio de usuarios aleatorios para esta práctica.
- Si pasamos un parámetro llamado **results** con valor **10,** conseguiremos obtener datos de 10 usuarios aleatorios.
- Mostraremos esos datos en dos filas. Distinguiremos el fondo de los pares e impares y, además, en cada usuario colocaremos un botón con el texto ***' Cambiar '.***
- Cuando hagamos clic en el botón ***Cambiar,*** el usuario en el que está situado el botón, se cambiará.
- Mostraremos el mensaje ***Cargando...*** al principio, mientras llegan los datos de los 10 usuarios (cuando lleguen, se quita el mensaje). También cuando pulsemos ***Cambiar*** en el usuario a modificar, escribiremos ***Esperado usuario nuevo...*** en la celda de ese usuario (habremos, antes, quitado los datos del usuario) hasta que lleguen los nuevos datos.

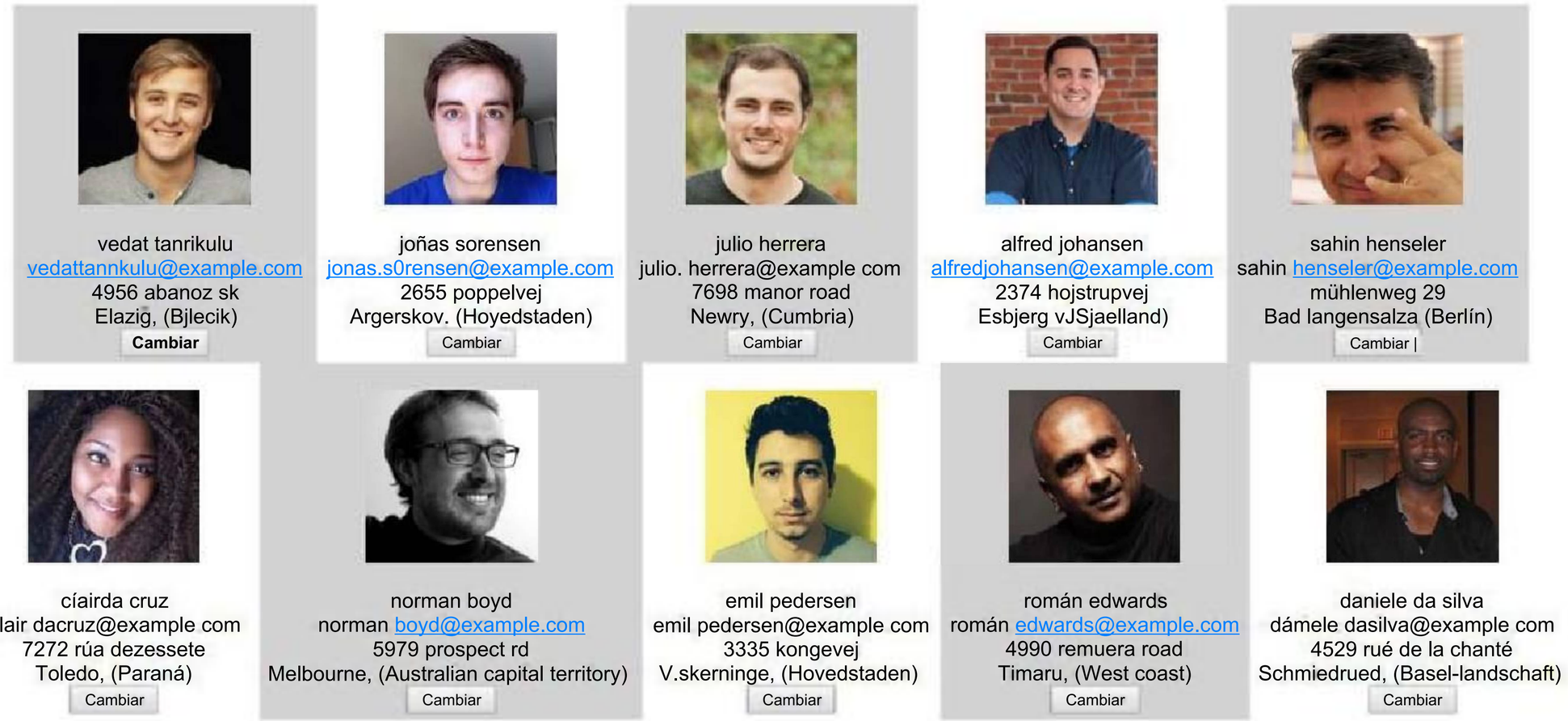


Figura 10.5: Ejemplo de página con los 10 usuarios aleatorios

SOLUCIÓN: PRÁCTICA 10.3

Nuevamente, el código HTML que necesitamos es muy simple:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>10 Usuarios aleatorios</title>
        <1ink rel="stylesheet" href=”css/estilos.css">
</head>
<body>
<main> </main>
<script src="js/accion.js"></script>
</body>
</html>
```

El CSS prepara la maquetación de los paneles en los que irán los usuarios. Al arrimar el ratón, sombreamos en azul el panel de usuario sobre el que está el ratón,

```
*{
        font-family: sans-serif;
}
main{
        text-align: center;
}
main div{
        display:inline-block;
        padding:.3em;
}
main div:nth-of-type(odd){
        background-color: 1ightgray;
}
main div:hover{
        background-color:rgb(88, 230, 230);
}
p{
        margin:0;
}
.error{
        background-color:red;
        color:white;
        font-weight: bold;
}
```

El código JavaScript utiliza varias funciones con la finalidad de conseguir un código más modular y mantenible. Así hay:

- **primeraMayus.** Función que simplemente recibe un texto y le devuelve con la primera letra mayúscula y el resto minúsculas.
- **rellenarüsuario.** Encargada de colocar los datos en la capa correspondiente con el código HTM1 apropiado.
- **generarEventoBotón.** Cada capa de usuario necesita que el botón pueda modificar el contenido para leer otro usuario aleatorio. Habrá una petición http para ello, en la que solo requeriremos un usuario. La petición se asocia al evento ***click*** del nuevo botón. Aunque este código es mejorable (no tiene sentido crear de nuevo el botón de cambio de usuario), le hemos dejado así para recordar la recursividad: tras programar el botón, hacer la petición y grabar los datos de esa petición (con ayuda de la función **rellenarU-suario),** volvemos a invocar a esta función para que genere el evento de nuevo.

La zona principal del código se basa en hacer la petición de los 10 primeros usuarios y colocar-los en las capas correspondientes. Código:

```
/**
 *   Retorn un texto con la primera letra en mayúscula
 *   @param {string} texto
 */
funetion primeraMayus(texto){
   return texto [0] . tollpperCase() +
                texto. si ice(l). tol_owerCase();
}

/ **
 *   Función que formnatea los datos del usuario aleatorio
 *   en un elemento div
 *   ©param {Element} capaUsuario Elemento div en el
 *   que se graba el usuario
 *   @param {JSON} usuario El usuario que
 *   contiene los datos obtenidos de randomuser.me
 */
funetion reí lenallsuario(capaUsuario, usuario) {
   let foto=usuario.picture.large;
   let mail=usuario.emai1;
   let nombre=usuario.ñame.first;
   let apel1ido=usuario.ñame.last;
   let calle=usuario.location.street;
   let ciudad=usuario.location.city;
   let estado=usuario.location.state;

   capaUsuario.innerHTML=
   '<figure>' +
      '<img src='${foto}' alt='foto’>' +
   '</figure>' +
```

```
    '<p>${nombre} ${apel1 ido}</p>' +
    ' <p>${mai1}</p>* +
    '<p>${calle}</p> ' +
    ' ${primeraMayus(ciudad)}, (${primeraMayus(estado)})</p>' +
    '<p> <button>Cambiar</button> </p> ";
}

/ **
  *   Captura el clic de botón en una capa de usuario
  *   @param {Element} capallsuario Elemento div que contiene
  *    los datos a cambiar del usuario
  */
function generarEventoBoton(capallsuario) {
    //Asignamos evento al botón
    let boton=capallsuar i o. querySelector("button" );
    boton.addEventListener("elick",(ev)=>{
        //Mensaje de espera que desaparece cuando se cargan los datos
        capallsuario.textContent="Esperando usuario nuevo...";
        //Hacemos petición de 1 usuario aleatorio nuevo
        fetch('https://randomuser.me/api')
            .then(resp=>resp.json())
            .then(data=>{
                let usuario=data.results[0];
                //Modificamos los datos de la capa actual
                reí lenallsuario(capallsuario, usuario);
                //programamos de nuevo el elick del botón que cambia el usuario
                generarEventoBoton(capallsuario);
            })
            .catch(error=>{
                capallsuario. innerHTML=' <p class='error' >${error} </p>' +
                                        '<p><button>Intentar de nuevo</button></p>';
            });
    })
}**
  * Zona principal del código
  */
var main=document.querySelector("main");
main.textContent="Esperando usuarios...";
//Petición de los 10 usuarios
fetch('https://randomuser.me/api/?results=10')
    .then((resp) => resp.json())
    .then(data=>{
        let datos=data.results;
```

```
    main.textContent="";
    for(let usuario of datos){
        //Generamos el HTML de cada capa con datos de usuario
        let divUsuario=document.createElement("div");
        reí lenallsuar io( di vUsuar io, usuario);
        //Programamos el click del botón que cambia el usuario
        generarEventoBoton(divUsuario);
        //Añadimos la nueva capa a main
        main.appendChild(divUsuario);
    }
})
 .catch(error=>{
    main.innerHTML='<p class='error'>${error}</p> ˣ;
});
```

Práctica 10.4: Mapa meteorológico de la Aemet

- La agencia meteorológica española **AEMET** dispone de una API para datos abiertos en la que proporciona numerosas posibilidades de obtener información meteorológica.
- Para poder obtener los datos de esta agencia, necesitamos darnos de alta como desarrolladores en la dirección:
  https://opendata.aemet.es/centrodedescargas/altaUsuario
- Las instrucciones de funcionamiento de la API de AEMET están en:
  https://opendata.aemet.es/dist/index.html?
- En el alta necesitamos indicar un email que tendremos que confirmar para que se nos conceda una API Key, una clave única que se otorga a cada desarrollador. Esa API la deberemos enviar cada vez que hagamos peticiones a la AEMET.
- Nuestra aplicación, inicialmente, muestra un cuadro de texto en el que introducir nuestra API y un botón de cargar mapa



Figura 10.6: Aspecto inicial de la aplicación del mapa meteorológico.

- Tras pegar nuestra API y pulsar el botón ***Cargar mapa,*** solicitaremos el mapa meteorológico del día en la dirección:

  https://opendata.aemet.es/opendata/api/mapasygraficos/analisis

- Se quitarán los controles de formulario y se mostrará (en horizontal) el mapa.
- Hay que recordar que debemos enviar vía GET un parámetro llamado apikey, con nuestra clave (la cual se obtiene del formulario).
- Los datos que llegan, nos ofrecen un objeto JSON donde la propiedad llamada **datos,** es una URL a la imagen del mapa. La petición http nos proporciona ese enlace, pero no el mapa en sí.
- Por ello, hay una segunda petición (a esa URL devuelta por la propiedad datos) que es la que nos proporciona el mapa

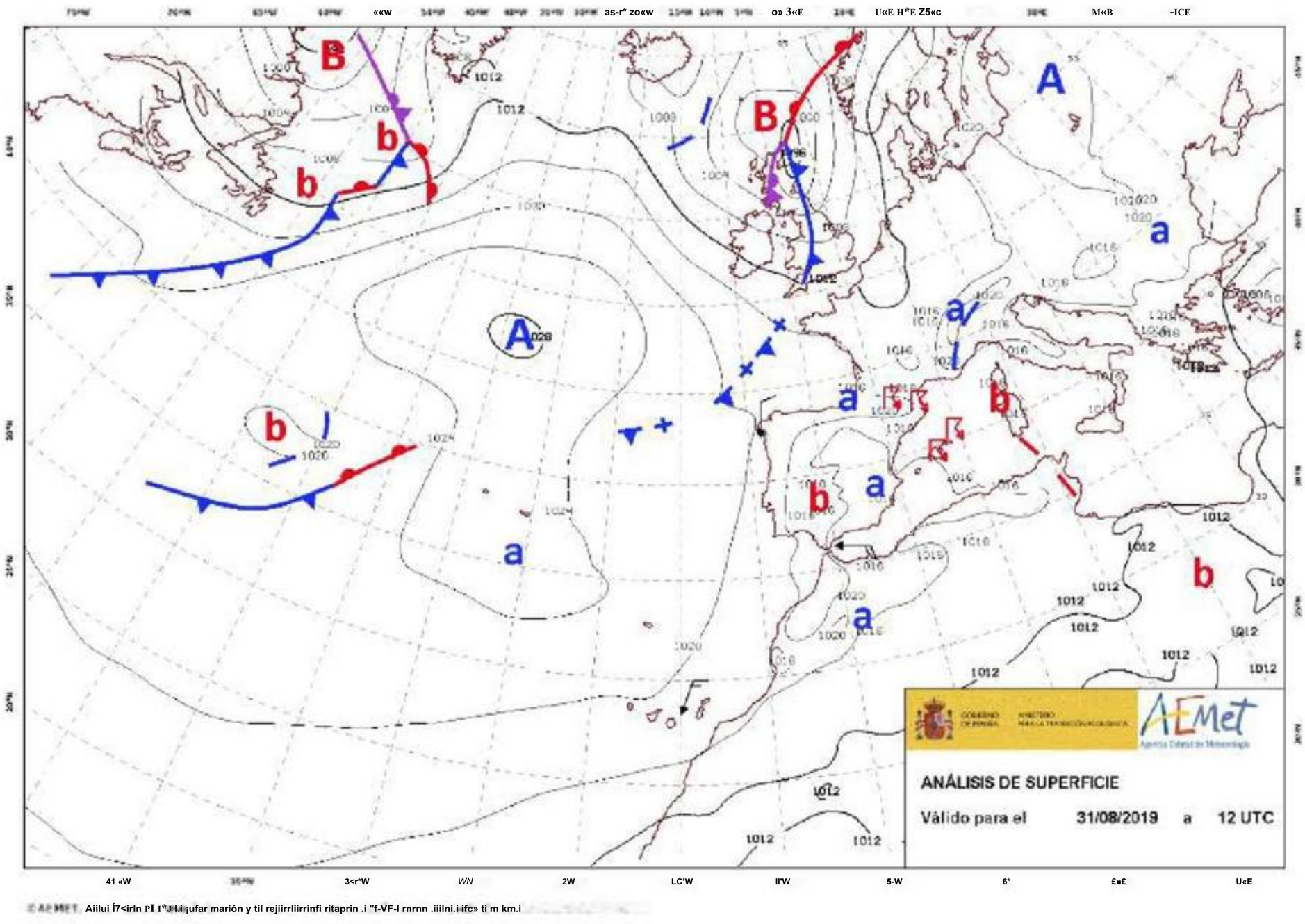


Figura 10.7: Mapa meteorológico obtenido de la API de datos abiertos de la AEMET.

SOLUCIÓN: PRÁCTICA 10.4

Código HTM1 (index.html), simplemente se encarga de mostrar la portada dentro de un contenedor main, además de cargar el código CSS y el JavaScript:

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```
        <meta charset="UTF-8">
        <title>Mapa meteorológico del día</title>
        <link rel="stylesheet" href="estilos.css">
</head>

<body>
<main>
        <hl>Imagen del día de la AEMET</hl>
        <form action="#">
                <label for="apikey">Pega tu API Key</label><br>
                <textarea name="apikey" id="apikey" cols="30"                    rows="10">
                </textarea> <br>
                <button>Cargar mapa</button>
        </form>
</main>
<script src="acción.js"></script>
</body>
</html>
```

El CSS es muy simple, solo se encarga de rotar el mapa y redimensionarlo:

```
main{
        text-align: center;
}
main img{
        transform:rotate(90deg);
        width:70%;
}
```

En el código JavaScript, la función **procesarFetch** se encarga de todo el trabajo. En este caso hemos decidido colocar el método **fetch** en una función **async** a fin de mostrar cómo utilizarla con la instrucción **await.** Es un buen ejemplo para usar la API Fetch de este modo porque hay dos peticiones http encadenadas: una para el enlace al mapa (en formato JSON) y otra para obtener el mapa en sí (formato binario).

Los parámetros de la petición realizados en el objeto de la petición, se han hecho siguiendo las especificaciones de AEMET. Hay que tener en cuenta que la documentación no se basa en el API de Fetch sino en JavaScript usando el objeto **XMLHttpRequest** y también se explica cómo hacer la petición usando la librería **jQuery.** No es difícil adaptar ese código a Fetch (de hecho, es más fácil esta API).

La primera petición obtiene os datos en JSON, se convierten usando el método json(). La segunda petición usa el método blob. Cada proceso y petición se realiza esperando la finalización de las promesas anteriores logrando un código muy sencillo de entender.

```
/**
 *  Función que realiza la petición fetch y coloca el mapa en el documento
 *  en el elemento indicado
```

```
 *   @param {string} apikey
 *   @param {Element} elemento Contenedor de la imagen y el formulario
 * */
async function procesarFetch(apikey,elemento){
        var form=elemento.querySelector("form");
        var headers = new Headers({
                "cache-control": "no-cache"
        });
        var conf={
                method:"GET",
                mode:"cors",
                headers:headers,
        }
        try{
                const respl=await        fetch("https://opendata.aemet.es/opendata/api/
mapasygraficos/analisis?api_key="+
                                                apikey,conf) ;
                const datal=await respl.json();
                const resp2=await fetch(datal.datos,conf);
                const mapa=await resp2.blob();
                const img=document.createElement("img");
                img.setAttribute("src",URL.createObjectURL(mapa));
                //Añadimos la imagen y quitamos el formulario
                elemento.removeChild(form);
                elemento.appendChild(img);
        }
        catch(error){
                let p=document.createElement("p");
                p.innerHTML="<strong>Error al cargar el mapa:</strong>                    "+error;
                elemento.appendChild(p);
        }
}

// Código principal dentro del evento load
// para asegurar la carga previa de los componentes
window.addEventListener("load",(ev)=>{
        let main=document.querySelector("main");
        let apiKey=document.querySelector("textarea");
        let boton=document.querySelector("button");

        boton.addEventListener("elick",(ev)=>{
                ev.preventDefault();
                procesarFetch(apiKey.valué,main);
        });
});
```

10.10 PRÁCTICAS RECOMENDADAS

Práctica 10.5: Cambio de moneda

- La URL https://exchangeratesapi.io/ proporciona una API para obtener cambios de moneda está documentado en la propia página. En la propia página aparece un ejemplo de devolución de datos:


```
{
  "base": "EUR",
  "date": "2018-04-08",
  "rates": {
    "CAD": 1.565,
    "CHF": 1.1798,
    "GBP": 0.87295,
    "SEK": 10.2983,
    "EUR": 1.092,
    "USD": 1.2234,
  ■ ■ ■
  }
}
```


- La propiedad base hace referencia a la moneda base del cambio, en el ejemplo anterior se tomará 1 euro como base. Así, por ejemplo podremos obtener el cambio de 1 EUR=1.565 GBP (Libras)



Figura 10.8: Ejemplo de conversión de monedas

- Para obtener la última fecha de cambio, se usa: https.7/api.exchangeratesapi.io/latest
- Podemos enviar el parámetro **symbols** para que nos devuelva solo los cambios a monedas concretas. Por ejemplo: https://api.exchangeratesapi.io/latest?svmbols=USD.GBP nos consigue los cambios de Euro a Dólares de EEUU y a Libras Esterlinas.

- También podemos cambiar la base de la moneda con el parámetro **base** al que podemos indicar una moneda diferente del Euro.
- Gracias a esta API, creamos el formulario de la Figura 10.8. Este formulario deberá permitir realizar cualquier cambio de moneda. Para rellenar los cuadros de lista, necesitaremos hacer una primera petición y mostrar todas las monedas.

Práctica 10.6: Imagen de la NASA

- La NASA tiene un servicio que permite mostrar lo que llaman la "Imagen del día". Ese servicio se puede invocar desde javaScript usando la URL: https://api.nasa.gov/planetary/apod
- A esa dirección (mediante GET) le podemos pasar un parámetro con la fecha llamado **date** en este formato ***"YYYY-MM-DD"*** por ejemplo ***"2019-05-01".*** Además se pasa otro parámetro llamado ***api_key*** que es una clave que nos pasa la NASA para poder usar su API.
- Si no se pasa el parámetro ***date,*** se nos da la imagen del día actual.
- La api key (la clave) la podemos obtener si rellenamos nuestros datos en la dirección:

  https://api.nasa.gOv/index.html#apply-for-an-api-key

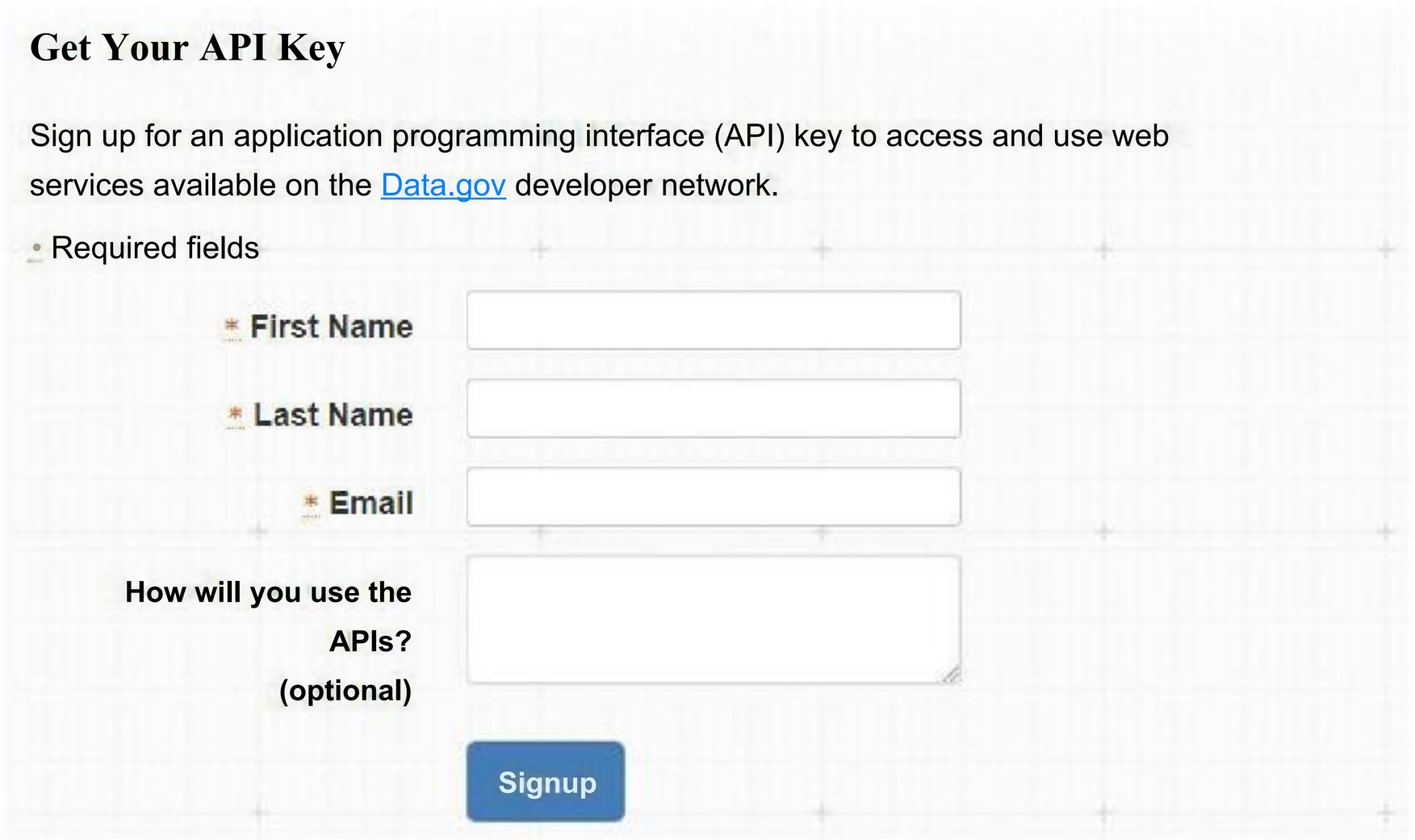


Figura 10.9: Formulario de alta en la página de la NASA para obtener nuestro API Key

- Al rellenar nuestro nombre, apellido y email nos darán la clave (un código numérico muy grande).
- Sabiendo esos datos prueba una petición a ver si llega el archivo JSON y observa el formato. A partir de ahí, crea una página que haga lo siguiente:

- Que inicialmente muestre la imagen del día y un control por debajo permita cambiar la fecha.
- Al cambiar la fecha se muestra la imagen de ese día.
- El color de fondo será negro.

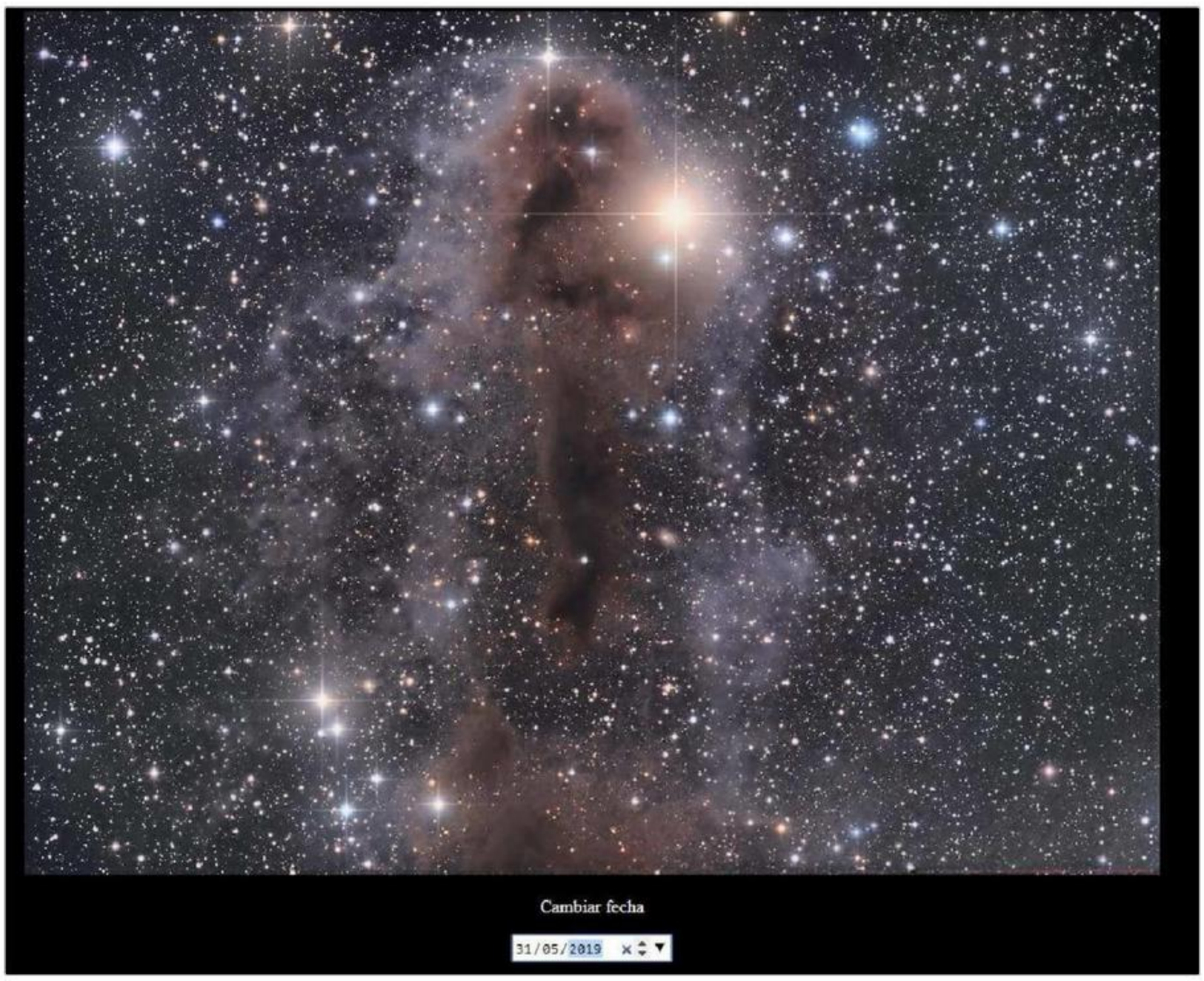


Figura 10.10: Imagen del día de la NASA para la fecha 31/05/2019

Práctica 10.7: Imágenes desde la nube

- El servicio picsum permite utilizar imágenes libremente. Posee una API al respecto. En este ejercicio podemos usar esta sintaxis:

  https://picsum.photos/300/300?image=1

- Este enlace obtiene la imagen número 1 a una resolución de 300 x 300 píxeles.
- Si añadimos el parámetro blur (no hace falta indicar valor en él), tendremos:

  https://picsum.photos/300/300?image=1?blur

  La imagen sale desenfocada.

- Sabiendo esto, genera una página que muestre 200 imágenes a 300 por 300 de resolución.
- Solo deben caber imágenes 4 en horizontal. Si la pantalla es pequeña, entonces, las imágenes se muestran más verticales. No debe haber scroll horizontal. Las imágenes deben quedar centradas en la ventana.

- Si se hace clic en una imagen, esta se desenfoca
- Además aparece desenfocada durante una semana.
- Si se hace clic en otra, entonces esa será la desenfocada la semana siguiente.
- Si se hace clic en la que ya está desenfocada, entonces quedará sin desenfocar.
- Usa las peticiones AJAX para enfocar y desenfocar.

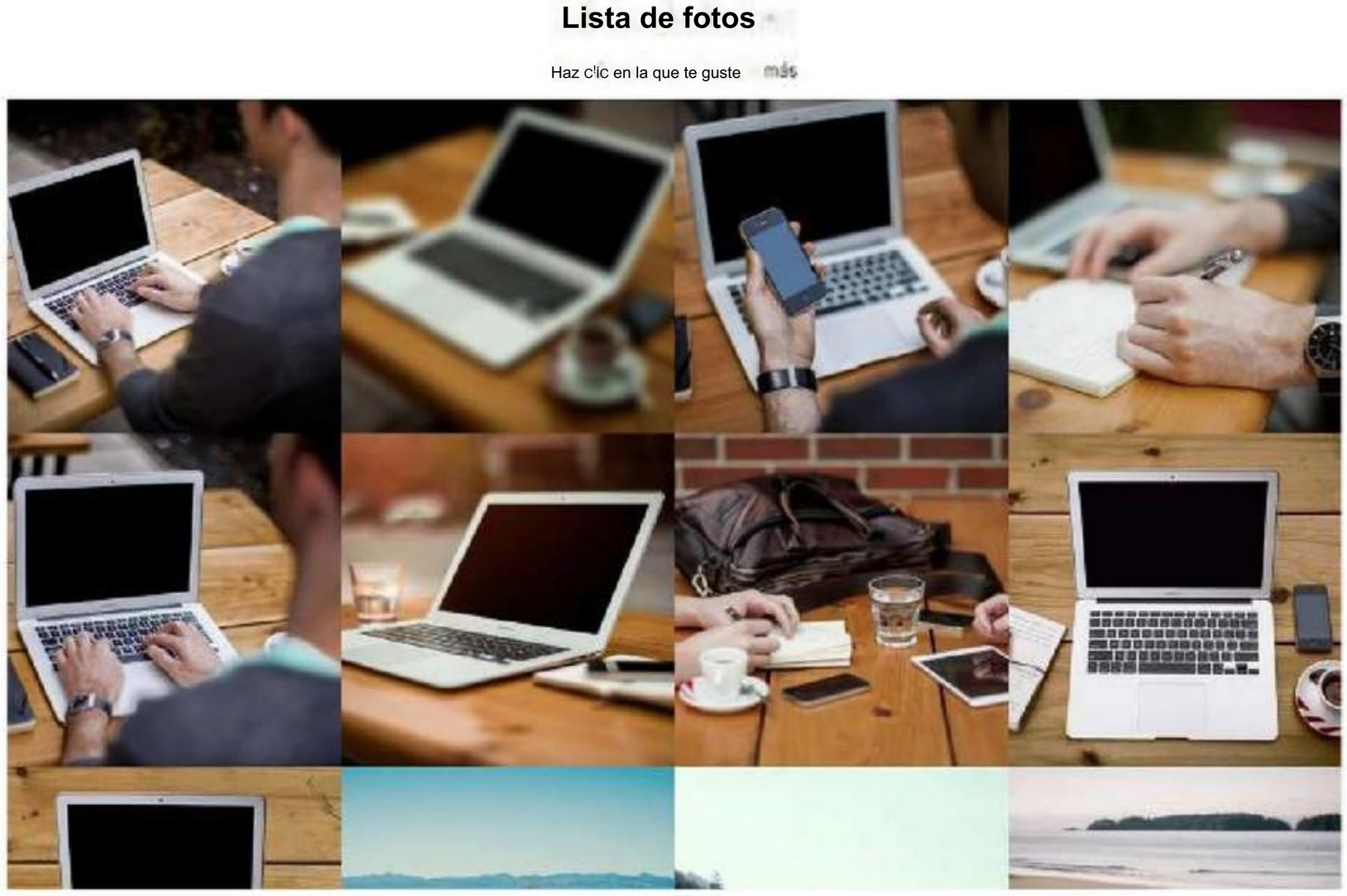


Figura 10.11: Aplicación con la lista de fotos de **picsum,** habiendo marcado la segunda para desenfocar.

Práctica 10.8: Animación vistosa

- Basándote en la API de **randomuser.me** (véase Práctica 10.2 y Práctica 10.3), consigue crear una página web que muestre 50 caras aleatorias consiguiendo que cambien 100 veces, de forma aleatoria, cada dos décimas de segundo.
- Es una animación muy llamativa cuyo efecto se basa en que no haya un exceso de retraso al conseguir las fotos. Lo ideal es cargar las fotos por adelantado.
- Lo lógico es hacer una petición a esa API que devuelva muchas personas (por ejemplo 1000) y almacenarlas en un array. Luego el array se puede desordenar de forma aleatoria cada dos o tres décimas.
- El control de las décimas lo llevan los intervalos.

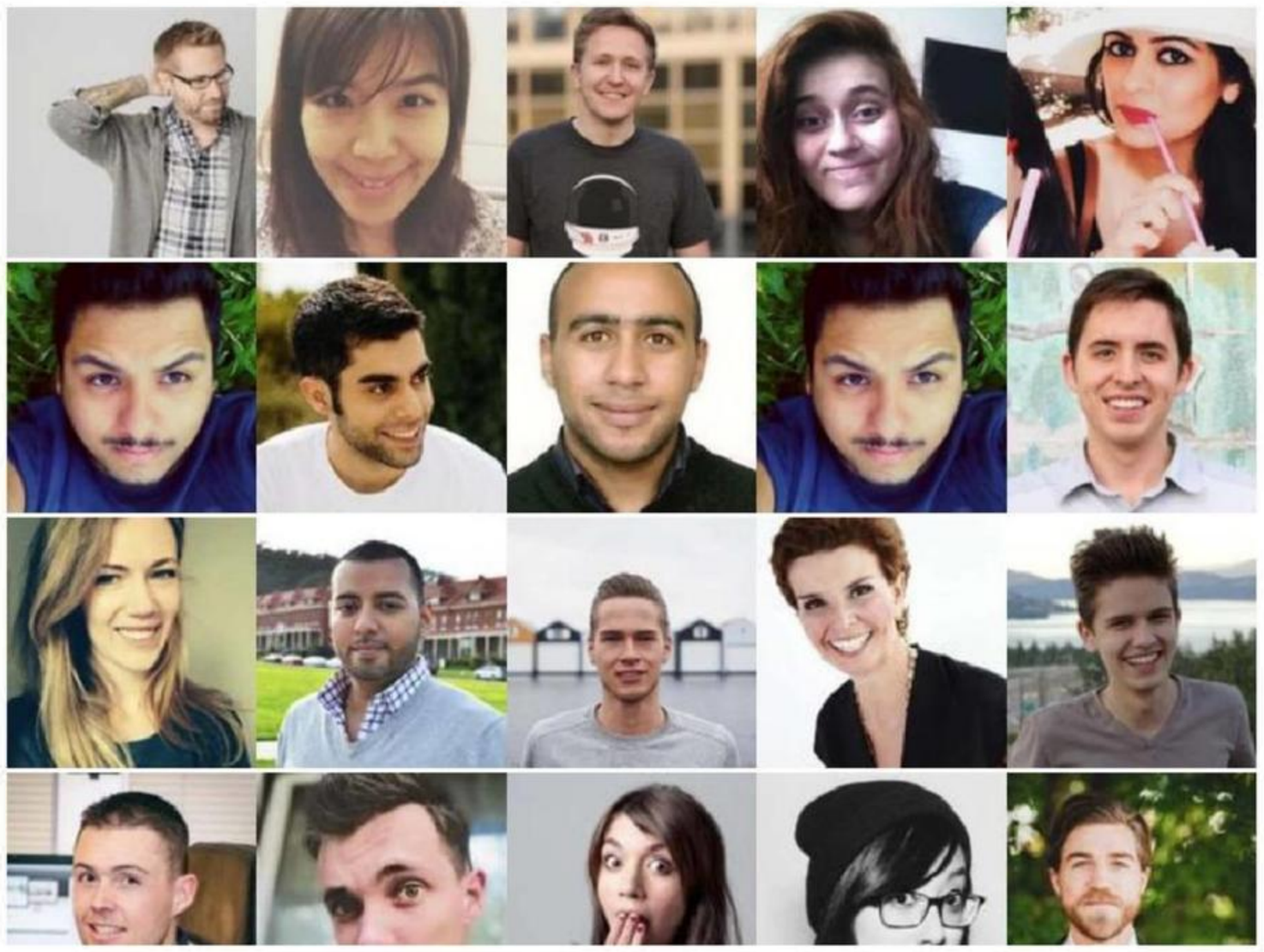


Figura 10.12: Una muestra de la página de la animación vistosa. Las fotos deben ir cambiando continuamente 100 veces. Luego se detiene la animación

10.11 RESUMEN DE LA UNIDAD

- AJAX es una tecnología que permite la comunicación cliente/servidor de forma dinámica mediante JavaScript a través de peticiones y respuestas http. Permite la carga de datos asincrona hacia un determinado componente de la aplicación.
- Los datos recibidos por las peticiones AJAX pueden ser de todo tipo: texto, binario, XML, etc. Actualmente la preferencia es JSON.
- Los servicios de Internet pueden aplicar políticas CORS para protegerse de peticiones procedentes de otros dominios. Solo podemos recoger datos de peticiones a otros dominios si su política de CORS lo acepta.
- AJAX facilita la creación de aplicaciones enriquecidas y de nuevos modelos de interfaz de usuario muy dinámicos y compatibles con todo tipo de tecnologías back-end. Hay algunas desventajas relativas a la dificultad de los navegadores para indexar páginas con contenidos obtenidos mediante peticiones AJAX o el no poder crear marcadores con el estado actual de una página.
- El objeto **XMLHttpRequest** es el que tradicionalmente se usa para gestionar las peticiones y respuestas AJAX. Actualmente, es más recomendable el uso de la API Fetch, basada en el método global **fetch.**
- El método fetch es el encargado de realizar las peticiones utilizando la URL destino y un objeto (llamado **request** u objeto de petición) que contiene los parámetros concretos de http para la petición.
- El resultado del método fetch es una promesa que, si se resuelve correctamente, retorna un objeto resultado de la petición **(response)** que contiene las cabeceras http y los datos de la respuesta.
- El objeto de respuesta posee métodos para procesar los datos que contiene, en función de su tipo (texcto, binario, JSON, etc.), lo cual genera una nueva promesa cuyo resultado positivo son los datos correctamente convertidos al formato deseado.
- Las cabeceras http se gestionan con un tipo de objeto llamado **Headers** que permite crear y modificarlas.
- El envío de datos permite utilizar todos los verbos http (GET, POST, PUT, etc.) y enviar los datos en los formatos habituales: codificados por URL o tipo ***form data.*** Para los datos de tipo form data existe un objeto llamado **FormData** que permite gestionarlos y crearlos de forma cómoda.

10.12 TEST DE REPASO

1.- ¿Qué es CORS?

a) Un grupo de música irlandesa.
b) Un comando http como GET o POST.
c) Un mecanismo que permite dar o no permiso a peticiones http de otros dominios.
d) Un tipo de cifrado http que facilita la autentificación del cliente hacia el servidor y viceversa.

2.- **¿Qué objeto clásico en JavaScript está relacionado con el manejo de AJAX?**

a) AJAXObject.
b) XMLHtmlRequest.
c) XMLAllowRequest.
d) XMLHttpRequest.

3.- **¿Qué objeto clásico en JavaScript está relacionado con el manejo de AJAX?**

a) AJAXObject.
b) XMLHtmlRequest.
c) XMLAllowRequest.
d) XMLHttpRequest.

4.- **¿Qué parámetros utiliza el método fetch para realizar peticiones?**

a) Una URL y el comando http de envío de petición
b) Una URL y un objeto de tipo Headers
c) Una URL y un objeto de tipo Response
d) Una URL y un objeto de tipo Request

5.- **¿Cómo podemos saber que el resultado de una petición AJAX tiende código "ok"?**

a) Mediante la propiedad **status** del objeto de respuesta.
b) Mediante la propiedad **statusText** del objeto de respuesta.
c) Mediante la propiedad **ok** del objeto de respuesta.
d) Las dos primeras respuestas son correctas.
e) Las tres primeras respuestas son correctas.

6.- **¿Qué resulta de invocar al método fetch?**

a) Un objeto Promise
b) Un objeto Response
c) Un objeto Request
d) Un objeto Headers

7.- **¿Cómo se recoge el objeto de respuesta (Response) de las peticiones fetch?**

a) Mediante la función callback del método then
b) Mediante el resultado del operador await aplicado a fetch
c) Las dos respuestas anteriores son correctas

8.- **¿Cómo se recoge el objeto de respuesta (Response) de las peticiones fetch?**

a) Mediante la función callback del método then.
b) Mediante el resultado del operador await aplicado a fetch.
c) Las dos respuestas anteriores son correctas.
d) Ninguna es correcta, el objeto Response se usa como segundo parámetro del método fetch.

9.- **¿Qué propiedad hay que modificar para indicar que nuestra petición usando el comando http PUT?**

a) Propiedad method del objeto Request
b) Propiedad type del objeto Request
c) Propiedad method del objeto Response
d) Propiedad type del objeto Response

10.- **¿Qué tipo de objeto resulta de usar el método json() de los objetos de respuesta?**

a) JSON
b) String
c) Object
d) Promise

11.- **¿Cómo podemos saber si los datos recibidos son de tipo JSON?**

a) Analizando la cabecera Contení-Type de la propiedad headers del objeto de respuesta.

b) Analizando la cabecera MIME-Type de la propiedad headers del objeto de respuesta.

c) Analizando la cabecera Content-Type de la propiedad headers del objeto de petición.

d) Analizando la cabecera MIME-Type de la propiedad headers del objeto de petición.

e) Analizando la propiedad type del objeto de respuesta.

12.- **¿Qué método del objeto de respuesta se usa para procesar datos binarios?**

a) binary()

b) bin()

c) data()

d) blob()

13.- **¿Qué método del objeto de petición se usa para indicar que nuestra petición es de tipo CORS ?**

a) type

b) method

c) mode

d) cors

14.- **¿Qué método permite añadir una nueva cabecera a un objeto de tipo Headers ?**

a) append

b) add

c) header

d) new

15.- **¿Qué método permite añadir una nueva cabecera a un objeto de tipo FormData ?**

a) append

b) add

c) header

d) new

16.- **¿Qué método permite obtener el valor de una cabecera concreta de un objeto Headers?**

a) has

b) get

c) is

d) obtain

17.- **¿Qué método permite obtener el valor de una cabecera concreta de un objeto FormData?**

a) has

b) get

c) is

d) obtain

18.- **¿En qué propiedad del objeto de petición se colocan los datos enviados por una petición GET?**

a) method

b) body

c) headers

d) En ninguna, se añaden a la URL

19.- **¿En qué propiedad del objeto de petición se colocan los datos enviados por una petición POST?**

a) method

b) body

c) headers

d) En ninguna, se añaden a la URL

20.- **Queremos enviar con nuestra petición AJAX, de tipo POST, los datos procedentes de un formulario asociado a un objeto llamado *form*. ¿Cómo podemos crear un objeto FormData, llamado Jd, con esos datos?**

a) let fd=FormData(form);

b) let fd=new FormData(form);

c) let fd=form.FormData();

d) let fd=form.getFormData();

UNIDAD 11

USO DE APIS DE JAVASCRIPT

OBJETIVOS

- Reconocer las ventajas de usar APIs de JavaScript en nuestras aplicaciones web
- Identificar las principales APIs
- Reconocer el funcionamiento básico de la API para el elemento canvas
- Almacenar datos en el navegador del usuario mediante la API Webstorage
- Obtener la localización del usuario mediante la API Geolocation
- Manejar elementos multimedia desde JavaScript
- Crear notificaciones desde JavaScript

CONTENIDOS

11.1 ¿QUÉ SON LAS APIS?

11.2 APIS DE HTML5

11.3 API PARA CANVAS

11.3.1 ¿QUÉ ES CANVAS?
11.3.2 EMPEZAR A TRABAJAR CON CANVAS
11.3.3 DIBUJO DE FORMAS
11.3.4 DIBUJO DE TEXTO
11.3.5 DIBUJAR IMÁGENES
11.3.6 BORRAR
11.3.7 CONCLUSIONES SOBRE CANVAS

11.4 WEB STORAGE

11.4.1 INTRODUCCIÓN A WEB STORAGE
11.4.2 GRABAR Y LEER DATOS
11.4.3 BORRAR ITEMS
11.4.4 CONCLUSIÓN SOBRE WEB STORAGE

11.5 GEOLOCATION

11.5.1 OBTENER LA POSICIÓN
11.5.2 OBTENER POSICIÓN CONTINUAMENTE

11.6 API PARA MULTIMEDIA

11.7 NOTIFICACIONES

11.7.1 FUNCIONAMIENTO DE LA API DE NOTIFICACIONES
11.7.2 CONCLUSIONES SOBRE LAS NOTIFICACIONES

11.8 PRÁCTICAS SOLUCIONADAS

11.9 RESUMEN DE LA UNIDAD

11.10 TEST DE REPASO

11.1 ¿QUÉ SON LAS APIS?

En varias ocasiones en este libro, hemos utilizado el término API ***(Application Programming Interface)*** explicando que es el conjunto de funciones, métodos, propiedades, objetos y componentes que permiten manejar un determinado software o componente. Así, por ejemplo el método **fetch,** los objetos **Response, Request** y **Headers,** con sus propiedades y métodos, forman, junto a otros elementos, la API conocida como **Fetch** que facilita la gestión de peticiones y respuesta de tipo AJAX. Incluso hay APIs mucho más extensas, como es el caso de la API DOM de JavaScript que permite manipular los elementos de las aplicaciones web y que está formada por cientos de métodos, propiedades y objetos.

Realmente las bases del lenguaje JavaScript ya están vistas y trabajadas en este libro, salvo algunos detalles avanzados o cuestiones de uso menos habitual. Lo que queremos decir es que lo fundamental para crear aplicaciones web en el lado del cliente es conocer el lenguaje JavaScript al máximo nivel posible, además hay que comprender y manipular el DOM perfectamente, es indispensable conocer cómo funciona AJAX para poder crear aplicaciones que utilizan datos y contenidos de servidores de Internet. A partir de aquí, JavaScript proporciona numerosas APIs más para realizar otro tipo de labores. Pero dependerá de nuestros intereses utilizar unas u otras.

Puede ser que algunas de las APIs que se comentan en esta unidad, no tengamos necesidad de tener que utilizarlas jamás. Hay APIs que son parte del estándar HTML5, otras las desarrollan terceros y no son estándares, aunque algunas de ellas son tan populares como las estándares.

Por ello, esta unidad tiene como principal pretensión entender cómo funcionan algunas de las más importantes, pero siempre con la idea de que si se necesita utilizar otras, el camino no va a ser complicado. La idea es entender cómo funcionan las APIs y como se aplican las más importantes, para así, si tenemos que aprender otras, no nos sea complicado.

Lo único que necesitamos conocer de las APIs son los métodos y elementos que proporcionan. Cuanto mejor las conozcamos, más grato será nuestro trabajo con ellas. En ese sentido es fundamental consultar su documentación.

11.2 APIS DE HTML5

El estándar HTML5 cubre el lenguaje HTML, CSS y el propio JavaScript. A diferencia de la entidad **ECMA** que se encarga de estandarizar el lenguaje JavaScript en su norma **ECMAScript,** las entidades normalizadoras de HTML5 (fundamentalmente **W3C** y **WHATWG)** definen APIs creadas para JavaScript a fin de manipular aspectos de las aplicaciones web.

Ya hemos trabajado con APIs HTML5: la API Fetch y también el propio DOM, que es una API que proporciona objetos **(window, document,** etc.) y los métodos y propiedades para trabajar con ellos y que nos permiten manipular los elementos de un documento HTML y gestionar los eventos.

Ejemplos de ellas son:

- **Web Storage.** Que permite el almacenamiento de datos en el navegador. Su finalidad es no depender de las cookies para realizar esta labor.

- **Web Workers.** API para la creación de tareas pesadas que se ejecutan en segundo plano y que, en primer plano, ralentizarían el trabajo.
- **Geolocation.** API que permite gestionar coordenadas de localización (longitud y latitud) con el fin de utilizarlas en aplicaciones web.
- **Web Sockets.** Permite la creación de sockets de red para la comunicación directa entre el cliente y un servidor.
- **API Canvas.** Para la manipulación del elemento canvas de HTML5 que permite la creación de gráficos en dos dimensiones.
- **Web GL.** API que incorpora las capacidades de **Open GL** para crear gráficos y animaciones en tres dimensiones en los elementos **canvas.**
- **FileSystem API. API** para poder utilizar desde javaScript archivos locales.
- **Application Cache.** Elemento fundamental para la creación de aplicaciones web que permitan el trabajo offline.
- **Media API.** Que permite manipular elementos de vídeo y audio en una aplicación web.
- **Text Track API.** Que permite manipular subtítulos para componentes de vídeo y audio.
- **Drag and Drop.** Para facilitar la programación de componentes arrastrables.
- **FullScreen.** API para poder manipular las aplicaciones web en estado de "pantalla completa".

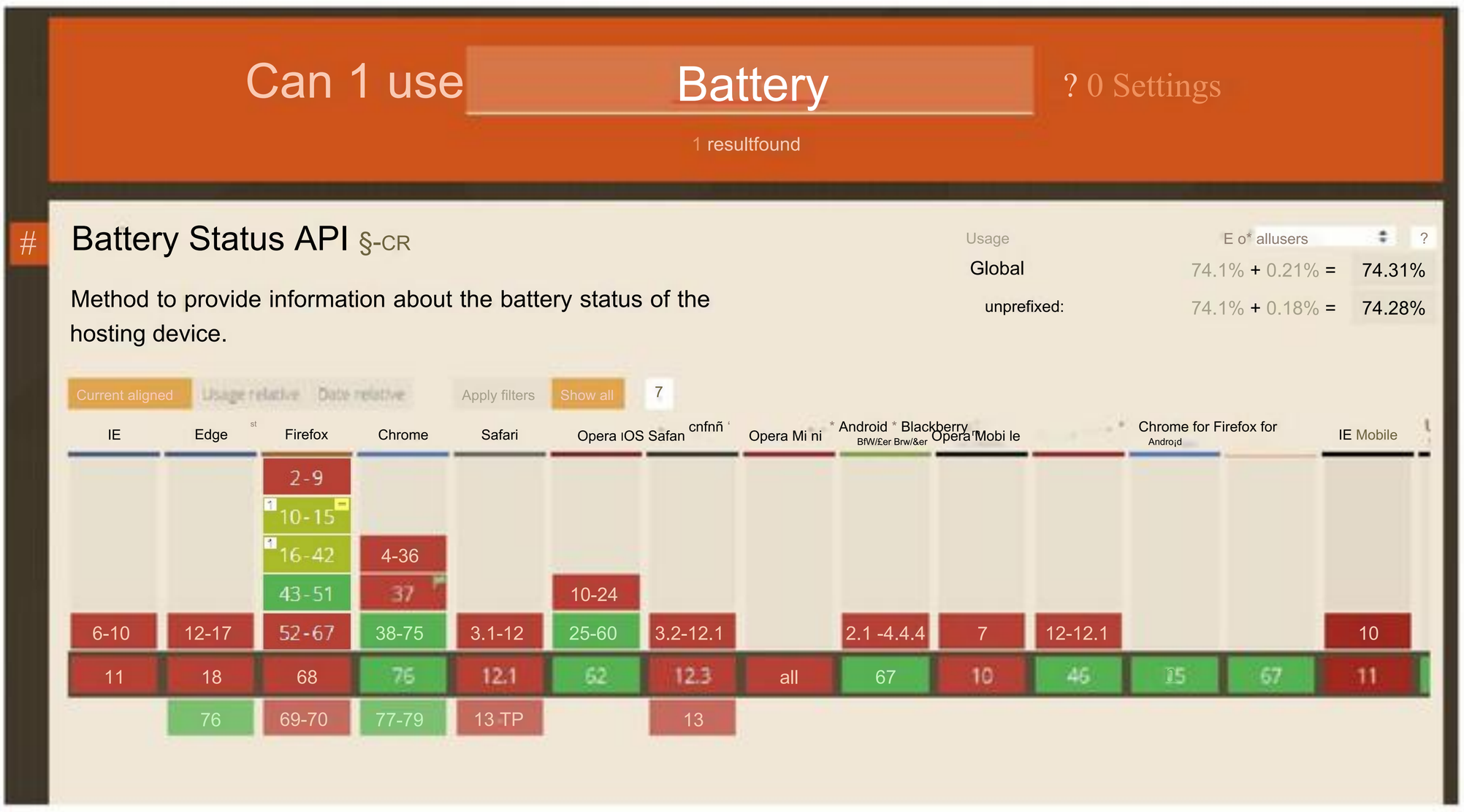


Figura 11.1: Estado de aceptación de la API Battery según la página web https://caniuse.com

- **Battery Status.** API para poder obtener información de la batería del dispositivo.

Hay muchas más. No todas están soportadas por todos los navegadores, pero muchas sí lo están y se pueden utilizar con total tranquilidad. La página https://caniuse.com/ nos permite saber la compatibilidad con los navegadores de las principales APIs de HTML5. De este modo podremos decidir si usarlas o no en función de su grado de implantación real.

Año a año, se van añadiendo nuevas APIs que proporcionan nuevas funcionalidades y, también, se modifican las APIs existentes para adaptarlas a las nuevas capacidades del lenguaje JavaScript o para modernizar alguno de sus aspectos. Conviene documentarse de vez en cuando sobre estos cambios.

11.3 API PARA CANVAS

11.3.1 ¿QUÉ ES CANVAS?

Canvas es un elemento de HTML5 que permite crear gráficos y animaciones utilizando comandos de JavaScript. El elemento canvas de HTML es un marco que se posiciona en el sitio que deseamos y que podemos modificar de tamaño:

```
<canvas id=" 1ienzo" width="500" height="300"> </canvas>
```

Desde JavaScript podemos acceder a este componente y, a través de los métodos que nos proporciona la API de Canvas, realizar creaciones gráficas en el espacio que proporciona el elemento **canvas.**

Este elemento tuvo un éxito muy grande desde su aparición en el estándar. Actualmente se sigue usando mucho, pero tiene bastante competencia con el éxito que también están teniendo los gráficos **SVG.**

11.3.2 EMPEZAR A TRABAJAR CON CANVAS

Como hemos comentado, en primer lugar debemos obtener una referencia al elemento **canvas** desde JavaScript:

```
const 1ienzo=document.getElementById("1ienzo");
```

Después debemos obtener el contexto del elemento.

```
const ctx=lienzo.getContext("2d");
```

El contexto es el objeto que permite dibujar en el lienzo. Proporciona todos los métodos y propiedades de dibujo. El método **getContext,** presente en los elementos de tipo canvas, es el encargado de obtener el contexto. A este método se le pasa un string que puede contener los valores **2d** o **3d.** Normalmente se usa el valor 2d porque es el que usa para gráficos en dos dimensiones, que son los habituales. El valor 3d permite el dibujo de gráficos 3d con ayuda de la librería **WebGL.**

A partir de este momento, se pueden utilizar los métodos que permiten elegir las propiedades de trazo y relleno de los objetos que dibujemos y los métodos que permiten dibujar dichos objetos.

En muchas ocasiones tenemos que indicar coordenadas y, por ello, hay que entender como funciona el modelo de coordenadas de canvas.

El origen de coordenadas está en la esquina superior izquierda. Se llama eje ***x*** al horizontal e ***y*** al vertical. Cuando se dan posiciones de puntos, se indica siempre primero la coordenada ***x*** y luego la y. La dirección de las coordenadas positivas va hacia abajo (eje x) y a la derecha (eje y).

El tamaño en píxeles de las coordenadas del **canvas** lo ajustan las propiedades width y height de la etiqueta canvas de HTML.

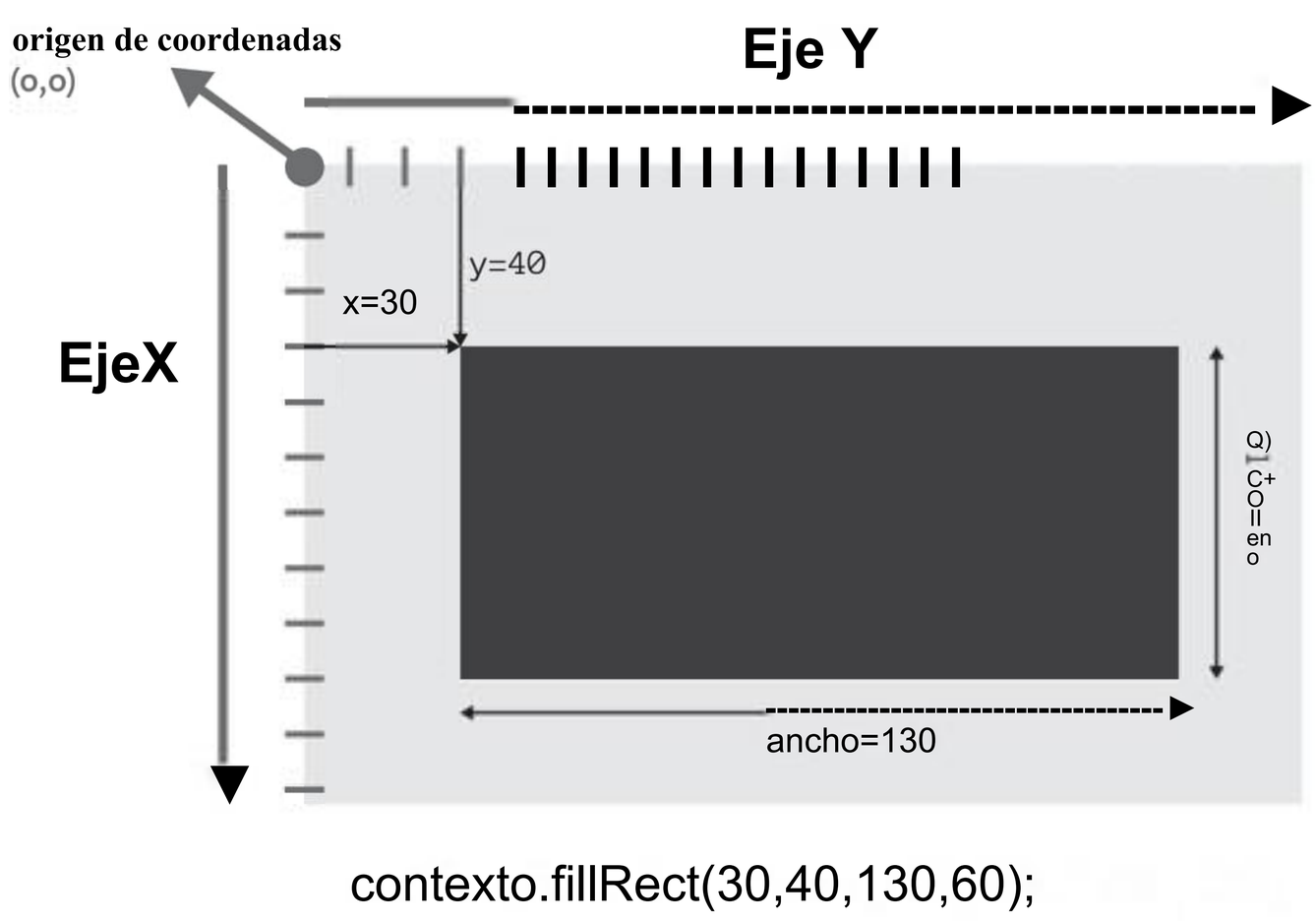


Figura 11.2: Funcionamiento del sistema de coordenadas de canvas

En la imagen anterior se muestra como se coloca un rectángulo en un área **canvas** (señalado en gris claro) mediante el método **fillRect.** Los dos primeros valores indican la posición de la esquina superior izquierda del mismo (coordenadas 30,40) el siguiente la anchura(130) y el último, la altura (60).

11.3.3 DIBUJO DE FORMAS

11.3.3.1 PROPIEDADES DE LAS FORMAS

Una forma es una figura geométrica pintada sobre el lienzo. Las formas tienen un color de fondo o relleno y un borde. El relleno se indica con un color y el borde tiene color y también grosor. Para indicar el color de relleno actual se usa la propiedad **fillStyle.** A esta propiedad se la puede indicar un color al estilo de CSS, por ejemplo ***"blue"*** o ***"#0000FF".*** La propiedad **strokeStyle** es similar y sirve para indicar el color del borde. Hay otra propiedad llamado **lineWidth** que permite indicar la anchura del borde.

Una vez indicadas estas tres propiedades, las figuras dibujadas utilizarán esos colores y grosores de línea. Si no los especificamos, se utilizarán los valores por defecto configurados en el navegador del usuario.

11.3.3.2 RECTÁNGULOS

Canvas solo dispone de una forma básica, el rectángulo. Se dispone de dos métodos para dibujar rectángulos:

- **fillRect.** Dibuja un rectángulo sólido. Para hacerlo se indica la coordenada superior izquierda del rectángulo y, la anchura y la altura del mismo.
- **strokeRect.** Dibuja un rectángulo transparente y pinta el borde.

Ejemplo:

```
<canvas id="lienzo" width="500" height="300"></canvas>
<script>
const 1ienzo=document.getElementById("1ienzo");
const contextor1ienzo.getContext("2d");
contexto. fillStyle=”red";
contexto.1ineWidth=2;
contexto.strokeStyle="gray";
contexto.fillRect(10,10,150,100);
contexto.strokeRect(30,30,150,100);
</script>
```



Figura 11.3: Rectángulos dibujados con los métodos **fillRect** y **strokeRect.**

11.3.3.3 TRAZOS

Un trazo es una serie de dibujos de líneas que forman una figura común. Los trazos deben comenzar invocando el método **beginPatli.** Este método sirve para marcar dónde comienza el trazo. Desde ese momento todos los comandos se entienden que van dirigidos a formar el trazo.

El trazo se finaliza invocando, sin argumentos, al método **closePath,** el cual cierra el trazado. Tras esa instrucción las siguientes acciones se entiende que ya no van dirigidas a ese trazo. El trazo se dibuja realmente invocando al método **stroke** (dibuja el contorno) o a **fill** (pinta el relleno). Tanto **stroke** como **fill** usan la configuración de relleno y trazo actual establecida por los métodos **strokeStyle, fillStyle** y **lineWidth** comentados anteriormente.

Tras la instrucción **beginPath,** los métodos **moveTo** y **LineTo** permiten dibujar trazos poligonales. El primero ***(moveTo)*** permite colocar el **cursor** en una coordenada concreta. El segundo

(lineTo) marcará una línea desde la última posición del cursor a la posición indicada por los parámetros de lineTo. Los parámetros de ambos métodos son una coordenada ***"x"y*** otra ***"y".***

Ejemplo:

```
const                1ienzo=document.getElementById("lienzo");
const ctx=lienzo.getContext("2d");
ctx.beginPath();
ctx.1ineWidth=2;
ctx.strokeStyle="blue";
ctx.moveTo(50,10);
ctx.1ineTo(100,60);
ctx. 1 ineTo(50,110)      ;
ctx.1ineTo(0,60);
ctx.1ineTo(50,10);
ctx.stroke();
ctx.closePath();
```

El resultado sería:

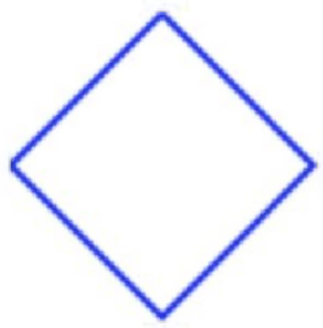


Figura 11.4: Rombo dibujado con los métodos de creación de trazos **lineTo** y **moveTo.**

Hay un método que dibuja trazos rectangulares. Se llama **rect.** Los parámetros son los habituales en el relleno de rectángulos: coordenadas de la esquina superior izquierda (x ey), anchura y altura.

11.3.3.4 ARCOS

El método **are también se usa para crear trazos. De hecho se suele combinar con los anteriores. Tiene como parámetros: las coordenadas (x** e ***y)*** en las que se debe situar el centro del arco, el tamaño del radio, el ángulo inicial y el ángulo final. Los ángulos se miden en radianes, una medida muy matemática que se usa mucho en ciencias matemáticas y en programación. Cabe decir que, en esa unidad, dos veces P1 equivale a 360 grados. 90 grados serían Pl/2. Hay un último parámetro que indica, con un valor booleano, la dirección del arco: el valor **true indica que se ha de dibujar en la misma dirección que las agujas del reloj.**

Ejemplo:

```
const 1ienzo=document.getElementById("1ienzo");
const ctx=lienzo.getContext("2d");
ctx.beginPath();
ctx.1ineWidth=2;
```

```
ctx.strokeStyle="blúe";
ctx.beginPath();
ctx.moveTo(50,10);
ctx.lineTo(100,10);
ctx.1ineTo(100,110);
ctx.lineTo(50,110);
ctx.stroke();
ctx.closePath();
ctx.beginPath();
ctx.are(50,60,50,Math.P1*3/2,Math.PI/2,true);
ctx.stroke();
ctx.closePath();
```

El resultado es:

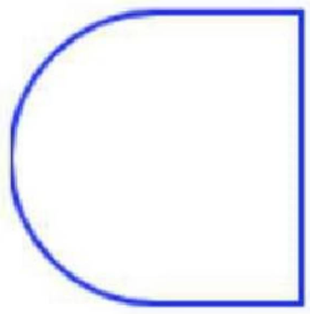


Figura 11.5: Figura conseguida con los métodos **lineTo** y **arc**.

11.3.4 DIBUJO DE TEXTO

La propiedad **font** permite indicar las propiedades del texto que vayamos a dibujar en el canvas. Las propiedades a indicar se separan con espacios (al estilo de la propiedad **font** de CSS) dentro de un string.

El método **fillText** dibuja y rellena un texto cuyo contenido es el primer parámetro. Después se indican dos coordenadas que serán la posición inferior izquierda para la línea base del texto,

```
ctx.1ineWidth=2;
ctx.strokeStyle="black";
ctx.beginPath();
ctx.moveTo(50,50);
ctx.1ineTo(200,50);
ctx.stroke();
ctx.strokeStyle="red";
ctx.font="bold 30px sans-serif";
ctx.fi 1lText("Prueba",50,50);
```



Figura 11.6: Texto y línea dibujados con **lineTo** y **fillText.**

11.3.5 DIBUJAR IMÁGENES

Una de las opciones más potentes de canvas es poder colocar una imagen **(gif, jpg,** etc.) en el lienzo. En este sentido disponemos de un constructor de objeto llamado lmage que crea un objeto de imagen equivalente al elemento img de HTML. Podemos modificar las propiedades habituales de las imágenes, por ejemplo, la propiedad **src.**

Canvas proporciona un método llamado **drawlmage** que, como primer parámetro, recibe un objeto de imagen y después dos coordenadas que marcarán dónde quedará la esquina superior izquierda de la imagen. Además, admite dos parámetros más con los que podemos modificar la anchura y altura de la imagen.

```
const img=new Image();
img.src="https://img.icons          8 .com/material-outlined/96/000000/dog.png";
img.addEventListener("load          " ,(ev)->{
        ctx.1ineWidth=2;
        ctx.strokeStyle="black      " ;
        //cuadrado contenedor
        ctx.strokeRect(200,150      , 96,96);
        //dibujo de la imagen
        ctx.drawImage(img,200,150);
});
```

En el código, las instrucciones de dibujo se colocan en la función callback del evento load de la imagen porque hay que asegurar que el código se ejecuta tras la carga de dicha imagen.



Figura 11.7: Imagen colocada con **drawlmage.** El recuadro lo consigue **strokeRect.**

11.3.6 BORRAR

El método **clearRect** dibuja un área rectangular sobre el canvas, con la intención de eliminar todo lo que hay dentro. Es decir, borra esa área y la deja con color transparente. Los parámetros son los habituales para indicar rectángulos:

```
const 1ienzo=document.getElementById("1ienzo");
const contexto=lienzo.getContext("2d");
contexto.fi 1lStyle="red";
contexto.1ineWidth=2;
contexto.strokeStyle="gray";
contexto.rect(0,0,200,200);
contexto.fi 11();
```

```
contexto.stroke();
contexto.clearRect(100,100,200,200);
```



Figura 11.8: Forma lograda con el apoyo de clearRect

11.3.7 CONCLUSIONES SOBRE CANVAS

Hay muchas más funciones y posibilidades para crear gráficos en Canvas. Evidentemente, crear un gráfico complicado usando estos métodos es difícil, pero combinado con otras utilidades de JavaScript (temporizadores, eventos, etc.) con formas muy sencillas se logran efectos muy llamativos e interesantes.

Hay programas gráficos profesionales que permiten dibujar en un entorno de trabajo más amigable y después convertir el resultado a formato canvas. Esto es, por ejemplo, lo que permite el software ***Adobe Anímate and Mobile Device Packaging*** (antiguo ***Adobe Flash),*** se crean escenarios utilizando las potentes herramientas de este software y luego se le pide que cree un archivo con las instrucciones JavaScript que lo conseguirían dibujar en un elemento canvas.

Otra cuestión es que existen APIs de terceros para trabajar con gráficos avanzados en las aplicaciones mucho más avanzadas y fáciles que Canvas(Tween, **paper.js, Raphael.js,** etc.). Aunque, como siempre ocurre al usar librerías externas, siempre es más veloz para el navegador usar las APIs nativas.

11.4 WEB STORAGE

11.4.1 INTRODUCCIÓN A WEB STORAGE

Se trata de una API con una finalidad similar a la de las cookies, pero con una forma de trabajar distinta. Al igual que las cookies, sirve para almacenar claves nombre/valor, pero con menos restricciones y más ventajas que las cookies.

Los datos que se almacenan, no se envían al servidor en ningún momento, se quedan en la máquina del cliente. No forman parte de ninguna petición ni respuesta http (como sí ocurre con las cookies). Además, pueden almacenar datos de incluso varios GB, superando con creces la limitación de 4KB de las cookies.

En definitiva, es un mecanismo de almacenamiento de información persistente que graba información en el ordenador del usuario, pero cuyos datos no son accesibles en caso de escucha de la comunicación entre el cliente y el servidor, porque solo se utilizan en la máquina cliente.

Para manejar esta API, el objeto window aporta estas propiedades:

- **localStorage.** Mediante este objeto, los datos se almacenan de forma perenne.
- **sessionStorage.** Los datos están disponibles hasta que se cierra la sesión, lo cual ocurre, normalmente, cuando se cierra el navegador.

El funcionamiento de ambos objetos es idéntico, solo cambia la duración de los datos.

11.4.2 GRABAR Y LEER DATOS

El manejo de esta API es muy sencillo. Para añadir o modificar un dato se usa el método **setItem** que recibe como primer parámetro el nombre o clave que damos al dato y como segundo, el valor:

```
localStorage.setItem("nombre","Jorge");
```

Para recoger un valor almacenado:

```
localStorage.getItem("nombre");
```

O bien:

```
localStorage.nombre
```

Se admite grabar valores de texto. Pero el resto de datos simples también funcionan:

```
localStorage.setItem("casado",true);
localStorage.setItem("salario",2200);
```

Eso sí, la clave siempre es de tipo texto.

Podemos almacenar datos más complejos gracias a la ayuda del objeto JSON. Si tenemos un objeto llamado **datosPersonales** cuyos datos queremos almacenar, valdría:

```
localStorage.setItem("datos",JSON.stringify(datosPersonales));
```

Se usa el método **stringify** para convertir un objeto en su equivalente **JSON.** El proceso contrario sería:

```
const datosPersonales=JSON.parse(localStorage.getItem("datos"));
```

Un detalle importante a tener en cuenta es que solo páginas lanzadas desde un servidor web pueden grabar este tipo de datos. Por lo que, incluso para practicar, debemos alojar y probar nuestro código en un servidor web. Puede valer el plugin **Live Server** de **Visual Studio Code** (para su instalación, véase Práctica 2.3 ***"Configurar Visual Studio Code",*** en la página 41).

11.4.3 BORRAR ITEMS

El borrado de claves almacenadas se realiza con **removeItem:**

```
localStorage.removeItem("nombre");
```

Como a los ítems podemos acceder directamente como propiedades del objeto **localStorage,** podemos borrar el ítem con el operador **delete** (véase 6.2.5 ***"Borrar Propiedades de objetos",*** en la página 210):

```
delete localStorage. nombre;
```

Podemos incluso borrar todas las entradas de golpe mediante el método **clear:**

```
localStorage.clear();
```

11.4.4 CONCLUSIÓN SOBRE WEB STORAGE

Como vemos, el manejo de esta API es muy sencillo y además, abre enormes posibilidades de trabajo que habitualmente se hacían con cookies y que en muchos casos podemos olvidar.

No siempre podemos olvidar las cookies, porque es habitual que en el servidor se requiera de las cookies para algunas labores. Pero, en el caso de datos que guardan detalles de la acción del usuario y que no requieren la comunicación de estos datos en el servidor, indudablemente ofrecen más ventajas los objetos **localStorage** y **sessionStorage** que las cookies.

Aunque todos los ejemplos de este apartado se han hecho con **localStorage,** el código funciona igual para sessionStorage. Podríamos, simplemente, haber cambiado simplemente en el código el objeto **localSgtorage** por **sessionStorage.** La diferencia, como ya se ha comentado, es que los datos de **sessionStorage** se borran al cerrar la sesión, pero en lo demás, son idénticos.

11.5 GEOLOCATION

La Geolocalización es la capacidad de detectar la posición geográfica de una persona o un dispositivo. Es útil en las aplicaciones para aplicar utilidades que pueden, por ejemplo, ayudar a detectar elementos cercanos al usuario que le otorguen determinados servicios. Por ejemplo, podemos crear una aplicación de búsqueda de gasolineras cercanas a la posición del usuario. Otro ejemplo es el de algunas aplicaciones web que permiten escuchar la radio y, mediante la geolocalización, permiten escuchar la emisora más cercana.

Detectar la posición del usuario entra en el terreno de la privacidad de dicho usuario, por lo que requiere que el propio usuario permita el uso de su localización a la aplicación web.

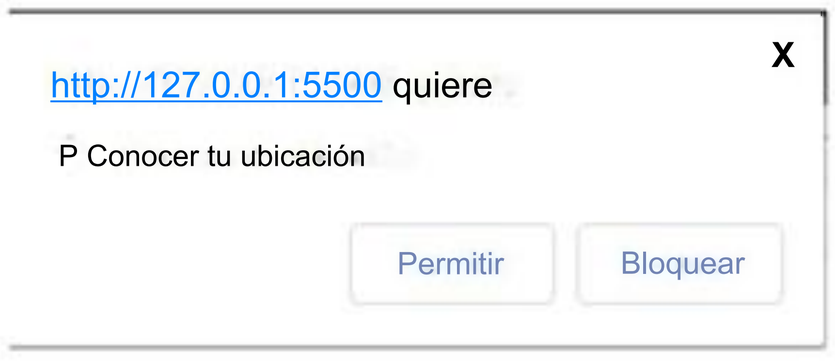


Figura 11.9: Cuadro de permiso de uso de geolocalización en Google Chrome.

El DOM proporciona un objeto llamado **geolocation** como propiedad del objeto **navigator** (véase 7.1.2 ***"Objeto navigator",*** en la página 251).

11.5.1 OBTENER LA POSICIÓN

Para obtener la posición actual del usuario se puede usar el método **getCurrentPosition** que posee, como parámetro, una función **callback** que recibe un objeto de tipo **Position.** Mediante este objeto podemos conseguir las coordenadas GPS de la localización del usuario:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Geolocalización</title>
</head>
<body>
<main> </main>
<script>
        var main=document.querySelector("main");
        navigator.geolocation.getCurrentPosition(pos=>{
                main.textContent='Latitud: ${pos.coords.latitude},
                'Longitud: ${pos.coords.longitude} ';
        })
</script>
</body>
</html>
```

Este código permite acceder a la propiedad **coords** del objeto de posición y así conseguir las coordenadas de longitud **(longitude)** y latitud **(latitude),** las cuales se muestran en el cuerpo del documento.

Aún más interesante es utilizar esas coordenadas para pasarles (vía AJAX o con un enlace) a un servicio de Internet para, por ejemplo, que nos muestre un mapa centrado en esa posición. Es lo que hace el servicio **openstreetmap.org** que proporciona una API gratuita para poder utilizar sus mapas en nuestras páginas.

El objeto de posición proporciona estas propiedades:

PROPIEDAD	USO
speed	Velocidad (en metros por segundo) a la que se desplaza el dispositivo al que le estamos obteniendo la posición.
altitude	Altura sobre el nivel del mar.
accuracy	Precisión (en metros) de la longitud y latitud.
aítitudeAccuracy	Precisión (en metros) de la altitud.
heading	Ángulo en el que el dispositivo se está moviendo.

Además, la función **getCurrentPosition** acepta un segundo parámetro que es otra función callback que se ejecuta en caso de errores al obtener la posición del usuario. Esa función permite recoger el error.

11.5.2 OBTENER POSICIÓN CONTINUAMENTE

Hay otro método llamado **watchPosition** que funciona exactamente igual que **getCurrentPosition.** La diferencia es que watchPosition se invoca continuamente de forma automática para recalcular la posición, lo que le hace ideal como método de seguimiento de dispositivos móviles.

El método devuelve un identificador asociado al examinador de posición que invoca a este método continuamente. Si deseamos que se deje de invocar el método, entonces debemos usar el método **clearWatch,** el cual recibe el identificador obtenido. De esa forma se deja de examinar la posición del dispositivo.

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Geolocalización</title>
</head>
<body>
        <main> </main>
        <script>
        var main = document.querySelector("main");
        var cont=0;
        let id=navigator.geolocation.watchPosition(pos => {
                cont++;
                let nodo=document.createElement("p");
                nodo.textContent=
                        'Latitud: ${pos.coords.latitude}, '                    +
                        'Longitud: ${pos.coords.longitude} ';
                main.appendChild(nodo);
                if (cont==3) navigator.geolocation.clearWatch(id);
        });
        </script>
</body>
</html>
```

El código anterior muestra la posición del dispositivo, pero en lugar de ver el texto una vez, lo veremos tres veces. watchPosition llama a la función callback continuamente para examinar la posición, pero hay un contador ***(cont)*** que cuando vale tres hace que se invoque a **clearWatch** para dejar de examinar la posición.

11.6 API PARA MULTIMEDIA

El vídeo y el audio son cada vez un recurso más habitual en las aplicaciones web. En muchas ocasiones se utilizan vídeos procedentes de servicios de Internet como **Youtube** o **vimeo** que proporcionan un código especial para incrustar el vídeo en nuestras páginas de forma cómoda. El problema de esta forma de colocar vídeo, es que dependemos de la API concreta del servicio y

que, además, estamos sujetos a los intereses de estos servicios. Es decir, no podremos controlar la publicidad, los controles ni tampoco qué enlaces a otros vídeos se van a ofrecer a nuestros usuarios.

Hay otras ventajas que nos ofrece el uso de esos servicios, como el hecho de no ser nosotros los que gestionamos su almacenamiento, que tampoco nos preocuparemos por la velocidad a la que ofrecemos el vídeo (la delegamos a ese servicio) y, la más importante, la visibilidad social que ofrecen.

Pero, cuando queremos mostrar un vídeo propio que queremos controlar al 100%, entonces se utilizan las etiquetas **video** o **audio** para mostrarlo:

```
<video src="video.mp4">
    No se puede mostrar el vídeo en este navegador
</video>
```

Por defecto, no se ofrecen botones de control del vídeo. Si los deseáramos, disponemos de atributos:

```
<video src="video.mp4" controls>
    No se puede mostrar el vídeo en este navegador
</video>
```

Aunque el código HTML es muy cómodo, cada navegador usa un formato de controles propio. En muchas ocasiones desearemos ser nosotros los que controlemos el vídeo.

Si queremos hacer un control más personal de los medios que incrustamos en las páginas, entonces, podemos acudir a la API que nos proporciona javaScript para este fin. Esa API se asocia a los elementos de tipo **video o audio y desde JavaScript podremos acceder a los siguientes métodos y propiedades:**

MÉTODO O PROPIEDAD	USO
playo	Reproduce el medio. Si estaba en pausa, se abandona la pausa.
pause()	Pausa el medio.
load()	Recarga el vídeo y coloca el cursor de posición al inicio.
paused	Propiedad booleana de solo lectura (no se puede modificar) que indica si el vídeo está en pausa.
loop	Valor booleano que, con valor true, indica que el vídeo se reproducirá en un bucle continuo. `video.loop=true;`
muted	Valor booleano que sirve para mostrar si el medio está silenciado o no. Permite cambiar el estado: `video.muted=true;`

MÉTODO O PROPIEDAD	USO
volume	Permite ver o modifica el valor del volumen del vídeo o audio al valor indicado. El valor es un número del 0 al 1: `video.volume=.5; //Volumen a la mitad`
controls	Booleano que permite indicar si deseamos mostrar o no los controles del navegador para pausar, retroceder, etc. `video.controls=false; //No deseamos los controles`
duration	Propiedad de solo lectura, que devuelve el número de segundos que dura el vídeo.
currentTime	Valor decimal que muestra, en segundos, la posición actual en la que se reproduce el vídeo. Permite modificar dicha posición: `video.currentTime=30; //posiciona el video a 30s`
ended	Propiedad booleana de solo lectura, que permite saber si el vídeo ha terminado de reproducirse.
playbackRate	Permite especificar o examinar la velocidad a la que se reproduce el vídeo o audio. Un valor de 1, reproduce el vídeo a velocidad normal. Por debajo de 1 reproduce a velocidad lenta ***y*** por encima a velocidad rápida.

También hay eventos propios relacionados con la reproducción de medios ([[referencia para consultar 8.5.2]]) que permiten realizar acciones avanzadas.

Así, este código nos permite mostrar por debajo del vídeo, el segundo en el que se está reproduciendo el vídeo, gracias a la captura del evento **timeupdate** que se lanza continuamente (cada vez que el vídeo cambia su cursor de posición temporal) cuando el vídeo se reproduce.

```
<video src="video.mp4" controls>
        No se puede mostrar el vídeo en este navegador
</video>
<main> Pruebas</main>
<script>
        let video=document.querySelector("video");
        let main=document.querySelector("main");
        video.addEventListener("timeupdate",(ev)=>{
                                main.textContent="Posición en segundos: "+video.currentTime;
        });
</script>
```



Figura 11.10: Vídeo reproduciéndose e indicando la posición actual en segundos.

11.7 NOTIFICACIONES

11.7.1 FUNCIONAMIENTO DE LA API DE NOTIFICACIONES

En los últimos años ha crecido la necesidad de que las aplicaciones web se comuniquen con el usuario para informar de novedades, cambios o cualquier otra información. Esta necesidad procede, realmente, de lo que las aplicaciones móviles han utilizado desde hace muchos años: las notificaciones al usuario.

Las notificaciones son mensajes que se hacen al usuario y que el sistema gestiona de manera especial. En el caso de las aplicaciones web suelen ser mensajes en una zona especial del navegador. Como ocurría con la API de Geolocalización, el usuario debe aceptar las notificaciones para que estas tengan lugar.

Las notificaciones las controla un objeto llamado **Notification.** Lo primero que necesitamos es solicitar al usuario el permiso para poder utilizar notificaciones. Esto se realiza mediante el método estático **Notification.requestPermission().** Este método proporciona una promesa que se resuelve positivamente cuando el usuario contesta a la petición. El objeto de respuesta guarda el texto **"granted"** si el usuario aceptó el mensaje de permiso o **"denied"** si denegó el permiso:

```
Notification.requestPermission()
.then(resp=>{
        i f(resp=="granted")
                consolé.log('Permiso concedido');
        else
                consolé.log('Permiso denegado');
});
```

El código anterior muestra por consola si se concedió o no el permiso. Si recargamos la página, veremos que ya no nos pregunta más por las notificaciones. La mayoría de navegadores guardan nuestra respuesta durante toda la sesión. Además, suelen proporcionar (normalmente en la barra de direcciones) algún icono o propiedad que permita modificar de nuevo el estado de la notificación.



Figura 11.11: Cuadro de permiso de uso de notificaciones en Google Chrome.

Generar una nueva notificación es crear un objeto de tipo **Notification.** En la construcción del objeto podemos indicar el texto que deseamos mostrar en la notificación.

Así pues, un código JavaScript que pida permiso de notificaciones y, en caso de que el usuario acceda, muestre un mensaje de saludo, sería el siguiente:

```
Notification.requestPermission()
.then(resp=>{
        if(resp=="granted")
                new Noti fication("¡Hola!");

});
```

La notificación, normalmente aparece en un área concreta del sistema. Por ejemplo, en los sistemas Windows suele ser el área inferior derecha donde está el área de notificaciones.

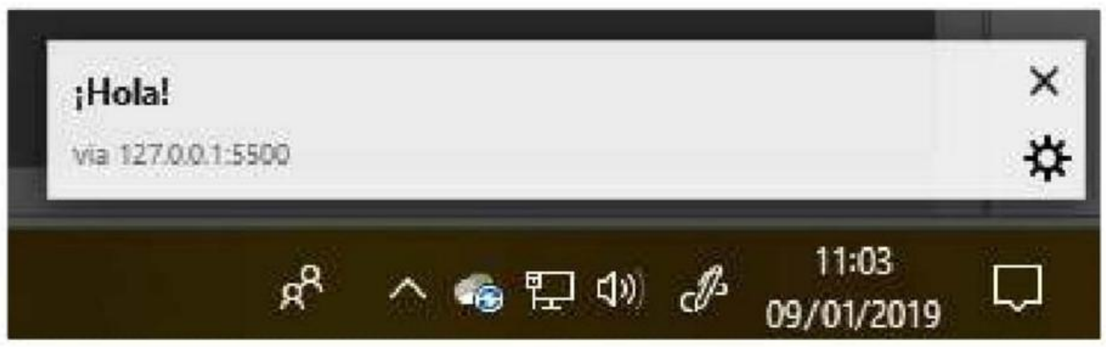


Figura 11.12: Un sistema Windows mostrando una notificación

Para saber si disponemos de permisos de notificación antes de requerirlo al usuario, el objeto **Notification** dispone de una propiedad llamada **permission** la cual puede tomar los valores **denied** o **granted** dependiendo de lo que el usuario haya elegido previamente. Pero si al usuario aún no se le ha pedido permiso, esta propiedad toma el valor **default.** De ese modo solo requeriremos el permiso si es necesario:

```
let permiso= Notification.permission;
if(permiso=="default"){
    Notification.requestPermission()
    .then(resp=>{
```

```
      i f(resp=="granted")
            new Notification(" ¡Hola!");

  });
}
```

El formato de las notificaciones puede ser más avanzado porque el constructor **Notification** admite dos parámetros: el primero es el título y el segundo es un objeto que puede indicar dos propiedades: **body** para el contenido de la notificación e **icón** para colocar un icono en la notificación:

```
new Notification("Bienvenida",{
    body:"Bienvenido/a a esta <strong>web</strong>",
    icón:"webLogo.png"
});
```

Las notificaciones, normalmente, se cierran automáticamente por parte del sistema. Pero el método **close(),** nos permite cerrarlas desde el propio código:

```
let permiso^ Notification.permission;
if(permiso=="default"){
   Notification.requestPermission();
    .then(resp=>{
       if(resp=="granted")
            new Notification(" ¡Hola!");
            setTimeout(()=>n.close(),2000);
   });
}
```

El código anterior cierra la notificación a los 2 segundos.

Las notificaciones, como cualquier elemento del DOM, disponen de eventos. Por ejemplo, podemos hacer que, al hacer clic sobre la notificación, abramos una URL concreta:

```
function mostrarNotificacion(){
      let n=new Notification("Información",{
            body:" ¡Puedes ir a mi página si haces clic!"
      });
      n.addEventListener("elick",(ev)=>{
            window.open("https://jorgesanchez.net");
      })
}
let permiso=Notification.permission;
if (permiso == "default") {
      Notification.requestPermission()
            .then(resp => {
                  if (resp == "granted") {
                        mostrarNoti ficacion();
                  }
```

```
            });
}
else if(permiso=="granted"){
        mostrarNoti ficacion();
}
```

11.7.2 CONCLUSIONES SOBRE LAS NOTIFICACIONES

La API de notificaciones es bastante reciente y además, la configuración del usuario en el sistema operativo permite, fácilmente, su deshabilitación. Es un recurso muy interesante para informaciones anexas a la principal que muestra una aplicación web para avisos que queremos que se vean fuera del navegador.

Un ejemplo de aviso podría ser el que proponemos en la ([[práctica]]), un formulario de suscripción a un servicio en el que hemos detectado que el usuario no ha enviado. Aunque la página web esté en segundo plano, porque el usuario se ha distraído con otro software de su sistema, podemos enviarle una notificación por ejemplo a los 15 segundos, indicando que aun no ha enviado los datos.

En todo caso, no hay que avisar de este recurso. El abuso de las notificaciones genera un gran rechazo por parte de los usuarios.

11.8 PRÁCTICAS SOLUCIONADAS

Práctica 11.1: Icono cargando

- Crea una página que consiga mostrar en una animación una serie de 10 rectángulos de color gris-
- Los rectángulos tendrán un ancho de 5 píxeles y una altura de 50.
- Uno de los rectángulos será de color negro y se irá cambiando a gris a la vez que el siguiente cambia a negro en cada décima de segundo
- Puede ser usada como capa que muestre a animación cuando se cargan elementos en segundo plano. En esta práctica basta con lograr la animación.

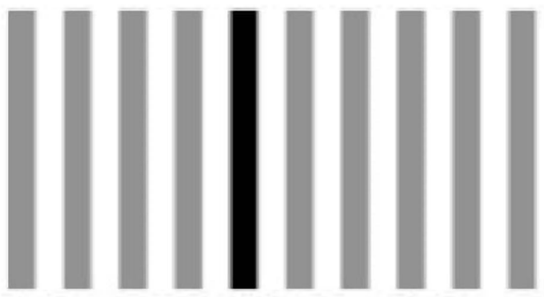


Figura 11.13: Imagen canvas con una animación que cambia el rectángulo negro en cada segundo

SOLUCIÓN: PRÁCTICA 11.1

La solución pasa por crear una función de dibujo en la que indicamos qué rectángulo queremos en negro (basta un número). Esa función se invoca usando un intervalo cada 100 milisegundos a la vez que se incremente un contador. Código:

```
<!DOCTYPE html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Animación cargando...</title>
</head>
<body>
        <canvas id="1ienzo" width="100px" height="50px"> </canvas>
        <script>

                function dibujaRect(ctx,n=l){
                        for(let i=l;i <=10;i++){
                                i f(n==i) ctx.fi 1lStyle="black";
                                else ctx.fi 11Style="gray";
                                ctx.fi 1lRect((i-1)*10,0,5,50)
                        }
                }
```

```
            window.addEventL istener("DOMContentLoaded",(ev)=>{
                let 1ienzo=document.getElementById("lienzo");
                let cont=0;
                ctx=lienzo.getContext("2d");
                setInterval(()=>{
                    cont++;
                    dibujaRect(ctx,cont);
                    i f(cont==10) cont=0;
                },100);
            });
        </script>
</body>
</html>
```

Práctica 11.2: Contador de visitas

- Crea una página que le diga al usuario el número de veces que el usuario ha pasado por la página.
- No se pueden utilizar cookies.

SOLUCIÓN: PRÁCTICA 11.2

Se trata de una página muy sencilla que nos va a permitir practicar con la API WebStorage. Hay que tener en cuenta que el control de visitas solo lo podemos realizar con la 1P concreta del usuario y con su navegador actual, lo que incapacita este método para un control real de visitas.

Código:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-    8 H >
    <title>Control de visitas</title>
</head>
<body>
<hl>Esta es tu visita          nº
    <span id="nVisita"></span>
</hl>
<script>
    let nVisita=localStorage.getItem("visitas");
    let span=document.getElementById("nVisita");
    if(!nVisita) nVisita=I;
    span.textContent=nVisita;
```

```
        //actualizamos el contador
        nVisita++;
        1ocalStorage.setItem("visitas",nVisita);
</script>
</body>
</html>
```

Práctica 11.3: Creador de tareas

- Crea una página que permita crear una lista de tareas.
- Aparecerá un cuadro de texto en el que podremos escribir el nombre de la tarea y un botón de **Añadir.**
- Las tareas añadidas se muestran en una lista que siempre las muestra en orden alfabético.
- Cada tarea va acompañada de un botón con el texto ***Eliminar*** que permitirá eliminar la tarea.
- No se permitirá añadir la tarea.
- La lista solo está vigente para el navegador, pero sin límite de fecha.



Figura 11.14: Aspecto con la aplicación con dos tareas añadidas y a punto de una tercera

SOLUCIÓN: PRÁCTICA 11.3

Se trata de un clásico de aplicación, pero al que hemos añadido ingredientes que nos permitirán repasar las estructuras vistas anteriormente, concretamente usaremos una estructura de tipo **Set** (véase 4.5 ***"Estructuras de tipo Set",*** en la página 141) para evitar las repeticiones.

Las tareas las añadiremos a esa estructura y luego las mostraremos. Cada vez que añadamos una tarea, la colocamos en la lista buscando su posición (en orden alfabético) y actualizaremos el **LocalStorage.** Esta última acción es la menos eficiente, en un sistema real seguramente sea más práctico guardar cada cierto tiempo o añadir un botón de guardar. Pero en este caso, con pocas tareas, el almacenaje en **LocalStorage** es lo suficientemente eficiente.

Código HTML **(index.html):**

```
<!doctype html>
<html lang="es">
<head>
        <meta charset="UTF-8">
        <title>Lista de tareas</title>
        <1ink rel="stylesheet" href="css/estilos.css">
</head>
<body>
<hl>Lista de tareas</hl>
<form>
<input type="text" id="tarea">
<button id="mas">+</button>
</form>
<div>
        <ul id="lista">

        </ul>
</div>
<script src="js/accion.js"></script>
</body>
</html>
```

Los elementos identificados como ***tarea, mas*** y ***lista*** serán los que utilicemos desde JavaScript.

En este caso nos hemos explayado un poco más con el CSS para dar un formato un poco más vistoso:

```
hl {
        font-family: sans-serif;
        font-weight: 700;
        text-align: center;
        color:teal;
        font-size:3em;
}
input{
        width:85%;
        height:56px;
        font-size:1.8em;
        padding-left:.5em;
        font-family: sans-serif;
        border-radius:10px 0 0 10px;
        border:lpx solid brown;
        outline:none;
        color:brown:
        text-align: left;
```

```
        float:left;
}
buttonf
        width:10%;
        height:60px;
        font-size:2em;
        color:white;
        font-family: sans-serif;
        font-weight: 700        ;
        background-color        :   brown;
        border-radius: 0            10px 10px 0;
        border:none;
        outline:none;
}
button:hover{
        background-color        : red;
}
li{
        font-size:2em;
        color:teal;
        font-family:sans-seri f;
        margin-bottom:.5em;
}
li    button{
        border-radius: 20px;
        font-size:.5em;
        width:auto;
        height:2em;
}
```

Finalmente el código JavaScript en el que varias funciones realizan toda la labor:

- **prepararBorrar.** Recibe el elemento **li** de una nueva tarea y genera apropiadamente el evento sobre el botón de eliminar dentro de dicho elemento **li.**

- **prepararLista.** Recibe un array con nombres de tarea y genera una lista de elementos de tipo li.

- **actualizarLista.** Usa la estructura de tipo **Set** con las tareas para que se almacene en **LocalStorage** en forma de array, convertido a texto JSON.

- **anadirTarea.** Función encargada de añadir la nueva tarea. Recibe el texto de la tarea y la coloca tanto en la lista de tareas interna (estructura ***Set)*** como en los elementos de tipo **li.** Usa los métodos anteriores para que se añada correctamente la nueva tarea.

Gran parte del problema es que para almacenar una lista de tareas, necesitamos un array. Pero si no queremos duplicados, nos facilita mucho la programación el hecho de tener una estructura de tipo **Set** que almacene las tareas. La conversión entre arrays y sets es sencilla.

Código JavaScript **(accion.js):**

```
//Variables globales
var 1istaTareas=new Set();
var btMas=document.getElementById("mas");
var tTarea=document.getElementById("tarea");
var uLista=document.getElementById("1ista");
/**
  *   Realiza todas las acciones necesarias para añadir
  *   la tarea y que se vea en la lista
  */
function anadirTarea(nuevaTarea){
        //buscamos si la tarea existe, si no, la añadimos
if(1istaTareas.has(nuevaTarea)==false){
        //almacenamos la nueva tarea en la lista
        1istaTareas.add(nuevaTarea);
        actual izarLista();
        //creamos el nuevo elemento con la nueva tarea
        let 1iNuevo=document.createElement("1 i");
        1 iNuevo.innerHTML='<span>${nuevaTarea}</span> ' +
                                    '<button>Eliminar</button>';
        //preparamos el evento en el botón
        prepararBorrar(1iNuevo)
        //Ahora la colocamos en su posición idónea
        //hay que buscar cuál es
        i f(uLista.children.length>0){

                for(let li of uLista.children){
                        //comprobamos que el elemento actual siga siendo menor
                        //en orden alfabético, de no ser así, añadimos el nuevo
                        //elemento delante del actual,
                        let contenidoLi=li.querySelector("span").textContent;
                        if(contenidoLi.localeCompare(nuevaTarea ,"es")>0){
                                uLista.insertBefore(1iNuevo,1 i);
                                return; //abandonamos la función
                        }
                }

        }
        //tanto en caso de que la lista este vacía
        //como en el de que sea el último elemento
        //hacemos lo mismo
        uLista.appendChild(1iNuevo);
    }
}
```

```
function prepararBorrar(1iTarea){
        let boton=liTarea.querySelector("button");
        boton.addEventListener("elick",(ev)=>{
                let textoLista=ev.target.parentNode.querySelector("span")
                                                                .textContent;
            //borramos el elemento li
            uLista.removeChild(ev.target.parentNode);
            //borramos el elemento de la lista y la actualizamos
            1istaTareas.delete(textoLista)
            actual izarLista();
        })
}

/**
 *  Muestra el contenido de la lista de tareas en forma de lista
 *  ordenada
 */
function prepararLista(arrayTareas){
        //recorremos el set y mostramos la lista
        for(let textoLi of arrayTareas){
                let 1iNuevo=document.createElement("1 i");
                1 i Nuevo.innerHTML= <span>${textoLi}</span>
                                        '<button>El iminar</button>';
                prepararBorrar(1iNuevo)                 ;
                uLista.appendChild(1iNuevo)
        }
>
/ **
 * Pone    el LocalStorage al día
 */
function actualizarLista(){
                localStorage.setItem("1istaTareas",          JSON.stringi          fy(
                        [...1istaTareas]
                        )
);
}

/** Código principal **/
//Lectura del array de tareas en LocalStorage
i f(localStorage.getItem("1istaTareas")){
        //leemos el array de tareas
        let arrayTareas=JSON.parse(localStorage.getItem("1istaTareas"));
        //ordenamos
        arrayTareas.sort((a,b)=>a.localeCompare(b,"es"))
        prepararLista(arrayTareas);
```

```
        //Convertimos el array en Set
        1istaTareas=new Set(arrayTareas);
}

//Evento del botón de añadir tarea
btMas.addEventListener("click",(ev)=>{
        ev.preventDefault();
        anadirTarea(tTarea.valué);
        tTarea.value="";
        tTarea.focus();
});
```

Práctica 11.4: Vídeo gigante

- Mostrar un vídeo que hayas descargado de modo que ocupe toda la ventana (todo el ***body)*** pero sin deformarse.
- No deseamos que aparezcan controles.
- Queremos que al hacer clic directamente en el vídeo, el vídeo se reproduzca o se pare. Cada clic cambia de un estado a otro.
- Inicialmente queremos que el vídeo se reproduzca automáticamente.
- Al pulsar el botón derecho queremos que un texto centrado nos diga el tiempo total del vídeo en minutos y segundos. Si volvemos a pulsar el botón derecho, este panel se oculta de nuevo. Así sucesivamente

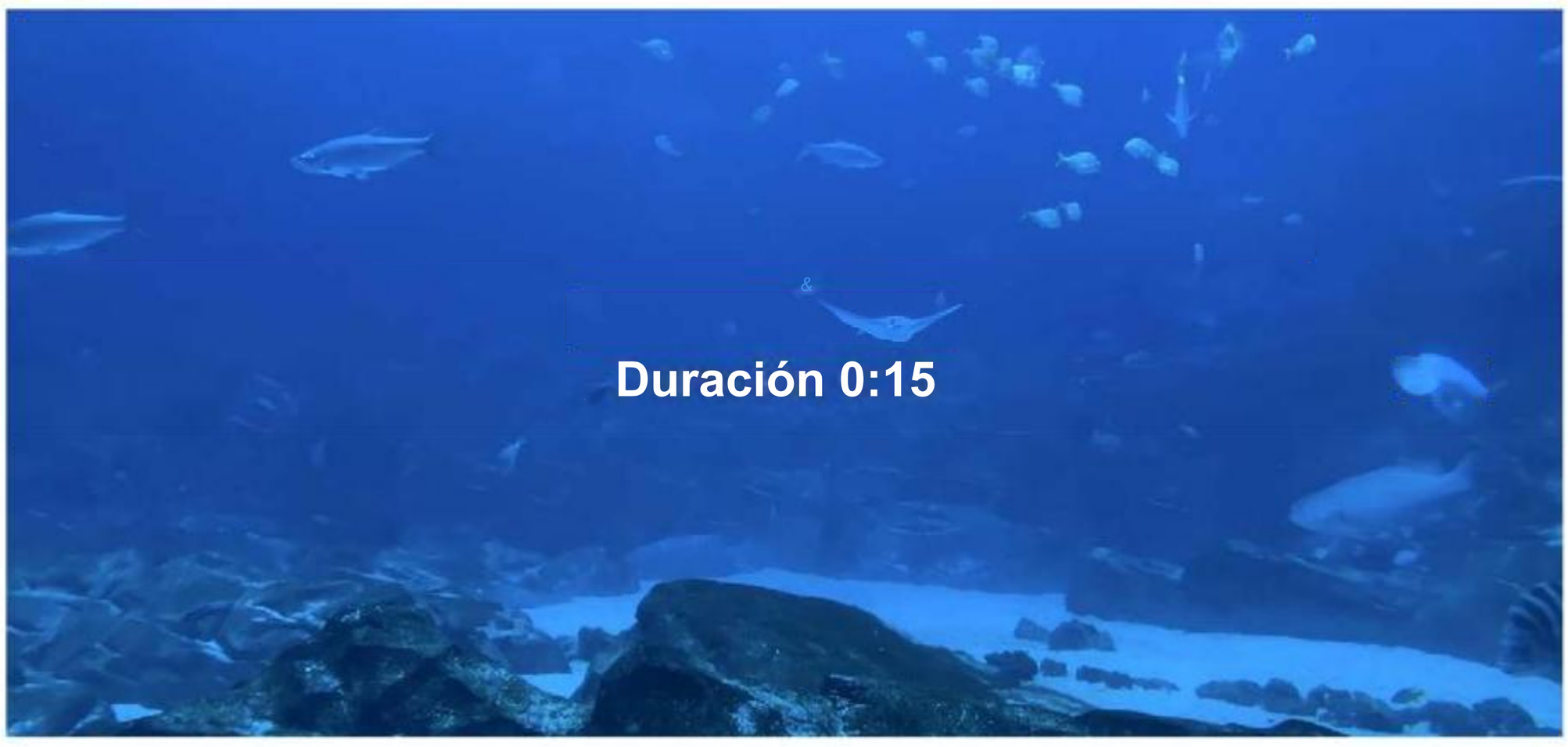


Figura 11.15: Apariencia de un vídeo de 15 segundos tras hacer clic con el botón derecho

SOLUCIÓN: PRÁCTICA 11.4

Para el vídeo a pantalla grande, basta con usar apropiadamente las propiedades CSS, sobre todo **object-fit.**

El resto es sencillo, conociendo la API Multimedia. El HTML es muy simple **(index.html):**

```
<!DOCTYPE html>
chtml lang="es">
<head>
        <meta charset="UTF-8">
        <title>Imagen a tamaño completo</title>
        <1ink rel="stylesheet" href="css/estilos.css">
</head>
<body>
<video src="video.mp4">
        No se puede mostrar el vídeo en este navegador
</video>
<div id="info">

</div>
<script src="js/accion.js"></script>
</body>
</html>
```

Este es el CSS, en el que también hay que destacar la presencia de la clase **mostrar** la cual ponemos y quitamos usando JavaScript sobre el panel ***info*** para que se muestre o no su contenido.

```
video {
        position: fixed;
        top: 0;
        left: 0;
        width: 100%;
        height: 100%;
        object-fit: cover;
}
#info{
        position:fixed;
        left:calc(50% - 200px);
        top:calc(50% - 50px);
        width:400px;
        height:100px;
        color:white;
        font-family: sans-serif;
        font-weight: bold;
        font-size:2em;
        display:none;
        text-align: center
```

```
}
.mostrar{
        display:block !important;
}
```

Finalmente el código javaScript con los dos eventos que usan las propiedades necesarias del elemento de vídeo.

```
let video=document.querySelector("video");
let info=document.getElementById("info");
video.controls=false;
video.play();
video.addEventListener("click",(ev)=>{
        if(video.paused)
                video.play();
        else
                video.pause();
});
video.addEventListener("contextmenu",(ev)=>{
        let minutos=parseInt(video.duration/60);
        let segundos=parseInt(video.duration%60);
        ev.preventDefault();
        info.textContent=`Duración ${miñutos}:${segundos}';
        info.classList.toggle("mostrar");
})
```

11.9 RESUMEN DE LA UNIDAD

- Las APIs de JavaScript, estándares o no, aportan una interfaz de acceso a métodos, propiedades y objetos mediante los cuales se nos puede facilitar el trabajo o podemos acceder a nuevos recursos disponibles en JavaScript.

- El estándar HTML5 aporta numerosas APIs, las cuales crecen y se mejoran constantemente.

- La API para el elemento **canvas** nos permite poder crear gráficos y animaciones utilizando el propio lenguaje JavaScript. Tiene numerosos métodos y capacidades, y se comunica perfectamente con el resto de elementos del documento.

- Con el nombre de **WebStorage** se da nombre a una interfaz que permite grabar información en el navegador del usuario utilizando una estructura clave/valor. Se pueden grabar datos de gran tamaño y tiene claras ventajas sobre las cookies, ya que no requiere que se usen los valores almacenados en las comunicaciones http.

  El objeto **locaíStorage** permite almacenar valores de forma permanente y **sessionStorage** solo para la sesión actual.

- La API de **Geolocalización** permite obtener, con el permiso del usuario o usuaria, las coordenadas de posición del dispositivo del usuario. Gracias a ello podemos mostrar mapas de su localización o dar información sobre servicios cercanos a su posición.

- La API **Multimedia** permite manejar vídeos y audios desde el propio JavaScript con la idea de personalizar su uso, o bien de comunicar el vídeo con las acciones que el usuario realiza en el resto de elementos HTML.

- Las notificaciones permiten dar información al usuario utilizando el sistema de notificaciones del sistema operativo. El objeto **Notification** es el encargado, en primer lugar, de pedir permiso (obligatoriamente) al usuario o usuaria para poder utilizar notificaciones, y también de proporcionar un constructor para crear nuevas notificaciones.

11.10 TEST DE REPASO

1.- **Consultando el valor de la propiedad Notification.permission, obtenemos un valor de *default.* ¿Qué ocurrirá cuando invoquemos al método de petición Notification.requestPermission()**

a) Se solicitará al usuario que indique si desea o no que la aplicación realice notificaciones.

b) Se tomará el valor predeterminado del navegador para las notificaciones.

c) Se tomará el valor predeterminado del sistema para las notificaciones.

d) No se permitirá usar notificaciones.

2.- **¿Qué objeto es el propietario del objeto geolocation?**

a) window

b) document

c) navigator

d) location

3.- **Supongamos que la variable *aud* es una referencia a un elemento de audio de la aplicación. ¿Cómo podemos hacer que el audio se vuelva a reproducir desde el principio?**

a) aud.restart()

b) aud.load()

c) aud.currentTime=0

d) Las dos anteriores son correctas

e) Todas son correctas

4.- **El método *getCurrentPosition* del objeto *gelocation* permite indicar como parámetro un objeto donde se recoge la posición del usuario. Supongamos que dicho objeto se llama *pos.* ¿Cómo podemos obtener la velocidad a la que va el dispositivo el usuario?**

a) pos. s peed

b) pos.coords.speed

c) pos . getSpeed()

d) pos.coords.getSpeed()

5.- **Queremos poner a tope el volumen de un vídeo referenciado por la variable *vid.* ¿Cómo lo podemos lograr?**

a) aud.volume=100;

b) aud.volume=l;

c) aud.volume=true;

d) Las dos anteriores son correctas

e) Todas son correctas

6.- **Queremos grabar el valor *Jorge* en *localStorage* asociado a la clave *nombre.* ¿Qué código lo consigue?**

a) localStorage.nombre="Jorge";

b) localStorage.setItem(
"nombre","Jorge");

c) Las dos anteriores son correctas

d) Ninguna es correcta

7.- **Hemos grabado en *localStorage* en formato JSON, con ayuda del método *JSON. stringtfy* los datos de un objeto asociándolo a la clave *objeto.* Como podemos recuperarlos para que los almacene una variable llamada *obj.***

a) obj=JSON.parse(localStorage.
getItem("objeto"));

b) obj=localStorage.
getItem("objeto");

c) Las dos anteriores son correctas

d Ninguna es correcta

8. **¿Qué devuelve el método Notification.requestPermission() ?**

a) Un valor booleano que si es vale **true** significa que se acepta el cuadro de permiso de notificaciones

b) Un texto que valdrá ***"granted"*** si se acepta el permiso de notificaciones.

c) Una promesa que se cumple si se acepta el mensaje de permiso para usar notificaciones.

d) Una promesa que se cumple si se acepta o cancela el mensaje de permiso para usar notificaciones.