

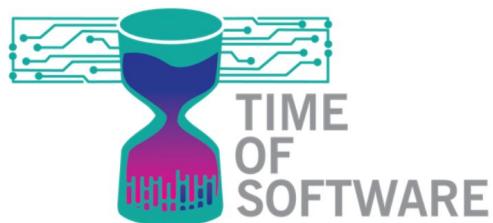
APRENDE JAVASCRIPT

EN UN FIN DE SEMANA



Alfredo Moreno Muñoz
Sheila Córcoles Córcoles

El contenido de la obra ha sido desarrollado exclusivamente por los miembros del equipo de **Time of Software**.



Reservados todos los derechos. Queda rigurosamente prohibida, sin la autorización escrita de Time of Software, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de esta obra por cualquier medio o procedimiento, incluidos la reprografía y

el tratamiento informático, así como la distribución de ejemplares mediante alquiler o préstamo público.

Primera edición.

Para más información visita:

www.timeofsoftware.com

www.aprendeenunfindesemana.com

TABLE OF CONTENTS

INTRODUCCIÓN

¿QUÉ NECESITO PARA EMPEZAR?

PROCESO DE APRENDIZAJE

Organización

Distribución del fin de semana

JAVASCRIPT

Historia

Node.js

npm

Descarga

Instalación en macOS

Instalación en Microsoft Windows

ENTORNO DE DESARROLLO

¿Qué es un entorno de desarrollo?

Microsoft Visual Studio Code

Instalación en macOS

Instalación en Microsoft Windows

Guía básica de uso

MI PRIMER JAVASCRIPT

OBJETIVO 1: VARIABLES Y CONSTANTES

Conceptos teóricos

Variables

Constantes

Ahora eres capaz de...

OBJETIVO 2: MOSTRANDO INFORMACIÓN

FASE 1: Mostrando información

Ahora eres capaz de...

OBJETIVO 3: ENTRADA DE INFORMACIÓN

FASE 1: Instalación de paquetes de NodeJS

FASE 2: Lectura de información

Ahora eres capaz de...

OBJETIVO 4: NÚMEROS

Conceptos teóricos

Números enteros y decimales

[Operadores aritméticos](#)

[FASE 1: Números enteros y operaciones aritméticas](#)

[FASE 2: Uso de paréntesis](#)

[FASE 3: Operadores asignación compuestos](#)

[FASE 4: Número decimales](#)

[Ahora eres capaz de...](#)

OBJETIVO 5: CADENAS DE TEXTO

[FASE 1: Manejo básico de cadenas](#)

[FASE 2: Métodos propios de las cadenas](#)

[Ahora eres capaz de...](#)

OBJETIVO 6: FECHAS

[FASE 1: Fechas](#)

[Ahora eres capaz de...](#)

OBJETIVO 7: BOOLEANOS

[Conceptos teóricos](#)

[Booleanos](#)

[Operadores relacionales](#)

[Operadores lógicos](#)

[FASE 1: Booleanos](#)

[FASE 2: Operadores relacionales](#)

[FASE 3: Operadores lógicos](#)

[Ahora eres capaz de...](#)

OBJETIVO 8: CONTROL DE FLUJO

[Conceptos teóricos](#)

[Bloques de instrucciones](#)

[Sentencia IF](#)

[Sentencia SWITCH](#)

[FASE 1: Sentencia IF](#)

[FASE 2: Sentencia Switch](#)

[Ahora eres capaz de...](#)

OBJETIVO 9: BUCLES

[Conceptos teóricos](#)

[Bucle](#)

[WHILE](#)

[FOR](#)

[DO](#)

[FASE 1: Bucle WHILE](#)

[FASE 2: Bucle FOR](#)

[FASE 3: Bucles DO](#)

[FASE 4: Bucles anidados](#)

[Ahora eres capaz de...](#)

[PROYECTO INTERMEDIO 1](#)

[Código fuente y ejecución](#)

[OBJETIVO 10: FUNCIONES](#)

[Conceptos teóricos](#)

[Funciones](#)

[FASE 1: Uso de una función](#)

[FASE 2: Funciones anidadas](#)

[Ahora eres capaz de...](#)

[PROYECTO INTERMEDIO 2](#)

[Código fuente y ejecución](#)

[Ahora eres capaz de...](#)

[OBJETIVO 11: ARRAYS](#)

[FASE 1: Arrays básico](#)

[FASE 2: Métodos propios](#)

[FASE 3: Iterando arrays](#)

[Ahora eres capaz de...](#)

[OBJETIVO 12: PROGRAMACIÓN ORIENTADA A OBJETOS](#)

[Conceptos teóricos](#)

[Cambio de paradigma](#)

[Concepto de clase y objeto](#)

[Encapsulación](#)

[Composición](#)

[Herencia](#)

[Sintaxis en JavaScript](#)

[FASE 1: Clases simples](#)

[FASE 2: Composición](#)

[FASE 3: Herencia](#)

[Ahora eres capaz de...](#)

[OBJETIVO 13: CONTROL DE EXCEPCIONES](#)

[Conceptos teóricos](#)

[Excepciones](#)

[FASE 1: Controlando excepciones](#)

[Ahora eres capaz de...](#)

[OBJETIVO 14: MANIPULACIÓN DE FICHEROS](#)

[FASE 1: Lectura de ficheros de texto](#)

[FASE 2: Escritura en ficheros de texto](#)

[Ahora eres capaz de...](#)

PROYECTO FINAL

[Código fuente](#)

[Clase Persona](#)

[Clase Dirección](#)

[Clase Teléfono](#)

[Clase Contacto](#)

[Clase Agenda](#)

[Programa](#)

[Ejecución](#)

[Ahora eres capaz de...](#)

¡CONSEGUIDO!

ANEXOS

[Comentarios de código](#)

[SOBRE LOS AUTORES Y AGRADECIMIENTOS](#)

[MATERIAL DESCARGABLE](#)

[OTROS LIBROS DE LOS AUTORES](#)

INTRODUCCIÓN

¡Bienvenid@ al maravilloso mundo de la programación con JavaScript!

Has llegado hasta aquí... ¡eso es porque tienes ganas de aprender a programar con JavaScript! Y lo mejor de todo, es que has decidido hacerlo con nosotros, ¡muchas gracias!

El objetivo del libro consiste en construir una base sólida de JavaScript para que puedas desenvolverte ante cualquier situación. Para ello, hemos diseñado un método de aprendizaje basado completamente en prácticas progresivas junto con nociones básicas teóricas, estructurado de tal forma que te permitirá aprenderlo en un fin de semana.

Una vez hayas acabado el libro, siguiendo el modo de aprendizaje que te proponemos, podemos garantizarte que vas a ser capaz de tener la autonomía suficiente para llevar a cabo tus propios proyectos de programación, o al menos lanzarte a que lo intentes.

Estamos seguros de que, si nos acompañas hasta el final del libro, se te van a ocurrir una gran cantidad de ideas de proyectos de programación, ya que cuantos más conocimientos vas aprendiendo, más curiosidad desarrollarás y por tanto más ideas te irán surgiendo.

Te animamos a que comiences a adentrarte en este mundo y disfrutes con cada proyecto. No desesperes si no lo consigues a la primera, ya que seguro que de cada error aprendes algo que te sirve para seguir avanzando. Esto es solo el comienzo.

¿Empezamos?

¿QUÉ NECESITO PARA EMPEZAR?

Para aprender JavaScript en un fin de semana, tal y como te proponemos en el libro, necesitarás lo siguiente:

- Un **ordenador**, con total independencia del sistema operativo que tenga instalado. Si no dispones de conexión a internet deberás de descargar desde cualquier ordenador conectado a internet el software que te explicamos en los apartados “*Node.js*” y “*Entorno de desarrollo*” e instalarlo en el ordenador que vayas a utilizar durante todo el aprendizaje. En los apartados siguientes te explicaremos los pasos a seguir para instalar el entorno de desarrollo en cada uno de los sistemas operativos soportados por la plataforma que vamos a utilizar.

Y por supuesto... ¡un fin de semana!

Al final del libro encontrarás la URL desde dónde puedes descargar el código fuente de todos los ejercicios del libro.

PROCESO DE APRENDIZAJE

El libro está escrito para ayudarte a aprender JavaScript de forma rápida, sencilla y con un enfoque práctico. Si eres nuev@ en programación, en el libro vamos a explicarte de forma sencilla todos los conceptos que necesitas saber para poder aprender a programar utilizando JavaScript. Si ya sabes programar, en el libro vas a encontrar todo lo que necesitas saber para tener una base sólida que te permita profundizar más.

Los temas tratados en el libro están seleccionados de forma cuidadosa y ordenados de tal forma que se facilita el aprendizaje progresivo de todos los conceptos que se explican.

El libro tiene un claro enfoque práctico, con multitud de ejemplos que te permitirán afianzar todos los conocimientos teóricos que te explicamos.

Veamos cómo está organizado el libro.

Organización

El aprendizaje está dividido en dos partes claramente diferenciadas:

- Bloque teórico sobre JavaScript y puesta en marcha de la plataforma de desarrollo que se va a utilizar.
- Teoría sobre programación y Práctica.

La primera parte del aprendizaje incluye una explicación teórica sobre el lenguaje y todo lo necesario para que seas capaz de poner en marcha el entorno de desarrollo que utilizarás para aprender.

El aprendizaje práctico está dividido en **14 Objetivos** diferentes, **2 Proyectos intermedios** y **1 Proyecto final**.

Los **Objetivos** tienen dificultad incremental. A medida que se va avanzando se van adquiriendo nuevos conocimientos de mayor complejidad que los anteriores. Los Objetivos están compuestos por diferentes ejercicios que llamaremos Fases. En cada Objetivo, antes de empezar, se explican todos los conceptos teóricos que se utilizarán en todas las Fases que lo componen.

Una **Fase** es un conjunto de ejercicios que profundizan en un área de conocimiento dentro del Objetivo. En cada Fase se especifica el código fuente junto con su explicación, además, se incluye un ejemplo de ejecución del código fuente.

Un **Proyecto** es un ejercicio de dificultad media/alta que tiene como objetivo afianzar los conocimientos aprendidos en los objetivos anteriores. El Proyecto es un ejercicio extenso y totalmente diferente a los ejercicios realizados en las fases de los objetivos.

Distribución del fin de semana

El método de aprendizaje ha sido diseñado y optimizado para que seas capaz de aprender JavaScript en un fin de semana. Obviamente, el tiempo de aprendizaje puede verse modificado ligeramente por los conocimientos previos que tengas.

La secuencia de aprendizaje recomendada que debes seguir para alcanzar el objetivo es la siguiente:



JAVASCRIPT

JavaScript es, probablemente, dentro de todos los lenguajes de programación que existen, el lenguaje que ha tenido un comienzo más interesante y el que mayor evolución ha tenido.

JavaScript al principio fue concebido como uno de los tres lenguajes más importantes dentro del desarrollo web. El ecosistema de desarrollo web estaba compuesto por estos lenguajes:

- **HTML**: lenguaje que definía el contenido de las páginas web.
- **CSS**: lenguaje mediante el cual se podía especificar el diseño de las páginas web.
- **JavaScript**: lenguaje para programar el comportamiento de las páginas web.

Básicamente, JavaScript era considerado un lenguaje que permitía realizar validaciones de los datos de los formularios, tales como mostrar una alerta al usuario, un teléfono mal introducido, etc. Al tratarse siempre de las mismas validaciones, los desarrolladores se limitaban a copiar y pegar validaciones de un proyecto a otro. Las operaciones típicas eran las siguientes:

- Cambiar contenido en los HTML.
- Cambiar el valor de atributos HTML.
- Mostrar y ocultar elementos HTML.
- Cambiar estilos CSS.

Pero la concepción de JavaScript y su uso cambió radicalmente con el paso de los años, y hoy en día, podemos decir que JavaScript es una bestia y el lenguaje más usado mundialmente. Ahora mismo con JavaScript puedes crear literalmente documentos HTML completos, revisar el estilo CSS sobre la marcha y transportar e interpretar de forma segura datos de varias fuentes remotas.

Mientras que, en el pasado, HTML era la tecnología principal en la web, en la era actual, **JavaScript es el rey**.

Historia

Para aprender JavaScript es necesario que conozcas los detalles más importantes de su historia.

Para entender la historia de JavaScript tenemos que remontarnos al comienzo de la década de los 90 y comprender cómo era la web en esa época. La historia de JavaScript comienza con Netscape y su navegador Netscape Navigator y la necesidad de desarrollar páginas web para ese navegador.

A mediados de la década de los 90, Netscape publicó un lenguaje de programación para su navegador llamado LiveScript (previamente se llamó Mocha). Fue desarrollado en 10 días y estaba principalmente basado en Java.

Todo parecía tranquilo en lo que a desarrollo web concernía, pero, más o menos al mismo tiempo que Netscape creara LiveScript, Microsoft lanzó su propio navegador, llamado Internet Explorer. En este momento empezó una guerra entre navegadores que todavía sigue en nuestros días.

Netscape y Sun Microsystems (empresa creadora de Java) se asociaron por intereses propios. Sun Microsystems tenía como objetivo que el próximo paso para su lenguaje de programación fuera la web y Netscape necesitaba un aliado para luchar contra Microsoft. El nombre de LiveScript fue cambiado a JavaScript. El objetivo que tenían ambas empresas era que mediante Java los desarrolladores profesionales crearan el contenido interactivo para las webs y mediante JavaScript se hiciera lo mismo, pero en ámbitos no profesionales.

La idea que tenían Netscape y Sun Microsystems y lo que realmente sucedió fue bastante diferente. JavaScript se integró más

rápidamente que Java en los navegadores web y hoy únicamente utilizamos JavaScript.

Actualmente, la asociación Ecma International estandarizó la especificación del lenguaje ECMAScript, cuya implementación es JavaScript. Ecma International es la organización que desarrolla y publica la especificación de ECMAScript y todas las nuevas versiones de esta, que eventualmente influyen en el lenguaje JavaScript.

NODE.JS

Node.js es una plataforma de desarrollo basada en JavaScript que nació hace unos 10 años y que desde sus inicios ha ido creciendo rápidamente y extendiendo su uso convirtiéndose en unas de las plataformas de desarrollo más importantes en la actualidad. Actualmente, Node.js sigue creciendo a una velocidad muy alta y se presenta como alternativa a las plataformas tradicionales de desarrollo. Empresas grandes y pequeñas utilizan Node.js para proyectos de todo tipo.

Una de las características principales de Node.js es su versatilidad, se puede utilizar para desarrollar aplicaciones web, aplicaciones de servidor, aplicaciones de red o cualquier tipo de programación de propósito general. Veamos algunos ejemplos de aplicaciones:

- Aplicaciones web.
- Aplicaciones que manipulen ficheros y carpetas.
- Aplicaciones que utilicen bases de datos.
- Aplicaciones de servidor.

Básicamente, con Node.js, puedes hacer de todo.

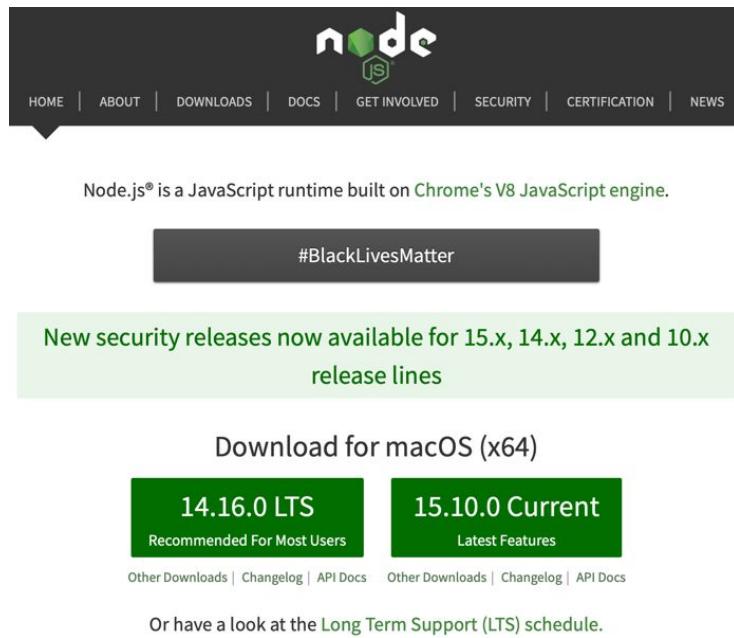
Node.js está basado en el motor JavaScript V8 de Google, cuya tarea es compilar y ejecutar código JavaScript. Dicho motor es el que utiliza el navegador Google Chrome para ejecutar JavaScript, por tanto, todas las mejoras de rendimiento y funcionalidades implementadas en Chrome estarán rápidamente disponibles en Node.js. Entrando un poco más en detalle en el motor V8, es un motor de JavaScript que se puede extender y que lo hace adecuado para la programación de propósito general y el desarrollo de aplicaciones de servidor con la misma potencia y posibilidades que se hace con otros lenguajes, tales como C#, Python o Java.

npm

La plataforma de desarrollo Node.js trae consigo un gestor de paquetes que te permitirá instalar paquetes en los proyectos que desarrolles. También podrás utilizarlo para instalar paquetes para el sistema operativo. Las siglas *npm* provienen de “*Node Package Manager*”, en castellano “*Gestor de paquetes de Node*”. Mediante *npm* vas a ser capaz de instalar paquetes con una sola línea de comandos.

Descarga

La descarga de Node.js la tienes que realizar desde esta dirección: <https://nodejs.org/>. Al entrar en la página web verás lo siguiente:

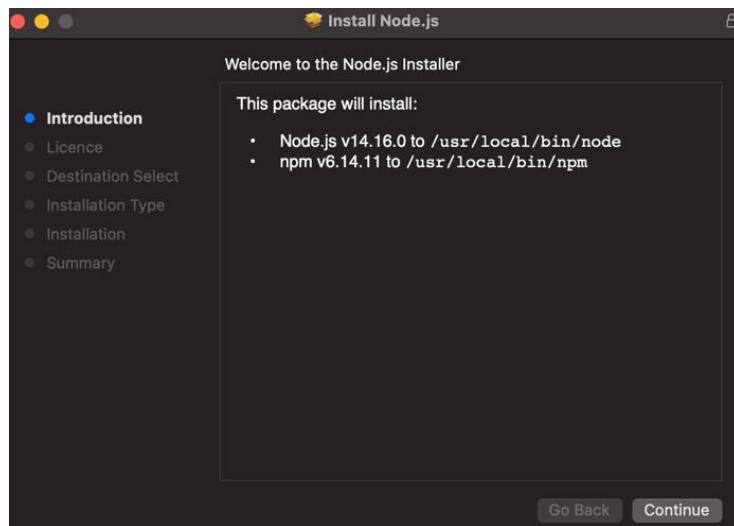


The screenshot shows the official Node.js website. At the top, there's a dark header with the Node.js logo and navigation links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the header, a message states "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine." A "#BlackLivesMatter" button is present. A green banner at the top of the main content area says "New security releases now available for 15.x, 14.x, 12.x and 10.x release lines". Below this, a section for "macOS (x64)" offers two download buttons: "14.16.0 LTS" (Recommended For Most Users) and "15.10.0 Current" (Latest Features). At the bottom of this section, there are links for "Other Downloads | Changelog | API Docs" and "Other Downloads | Changelog | API Docs". A note below the buttons says "Or have a look at the Long Term Support (LTS) schedule."

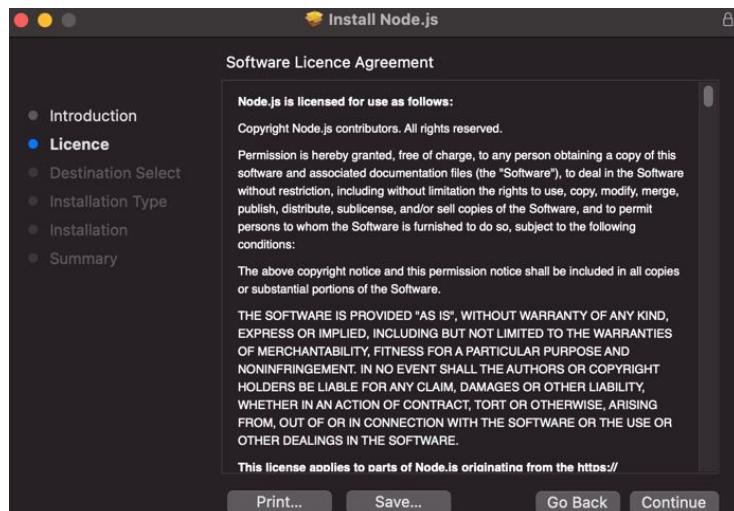
Tienes que darle a descargar a la versión que pone LTS (versión con soporte amplio). Una vez descargada estaremos preparados para proceder con la instalación.

Instalación en macOS

Para instalar Node.js en sistemas operativos macOS tienes que ejecutar el fichero que acabas de descargar. La primera pantalla que verás es la que te explica lo que se instalará en tu ordenador, presiona el botón continuar (*Continue*):

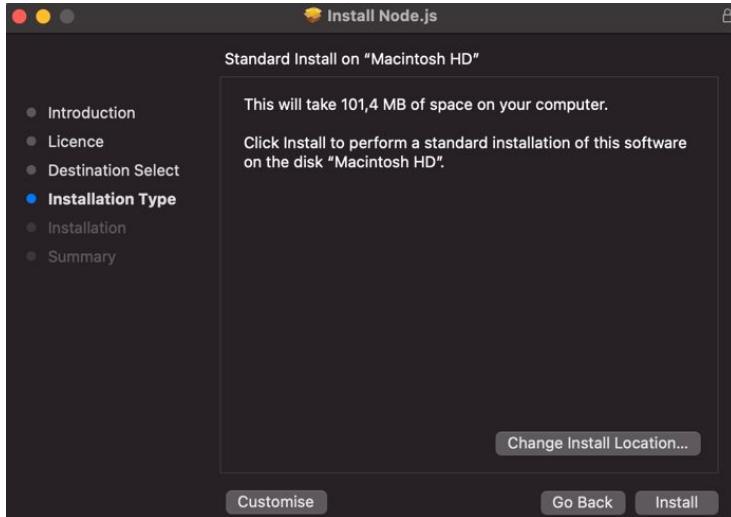


La siguiente pantalla que se muestra es el acuerdo de licencia de utilización, presiona el botón continuar (*Continue*):

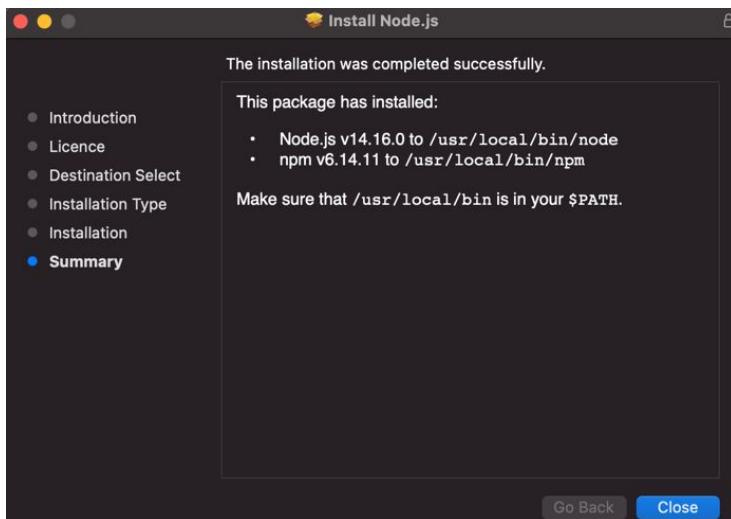


La siguiente pantalla que se muestra es en la que te dice el instalador dónde instalará Node.js. Si quisieras instalarlo en otra

carpeta que no sea la carpeta por defecto puedes hacerlo mediante el botón “Change Install Location...”, aunque nosotros te recomendamos no cambiarlo. Presiona el botón instalar (*Install*).



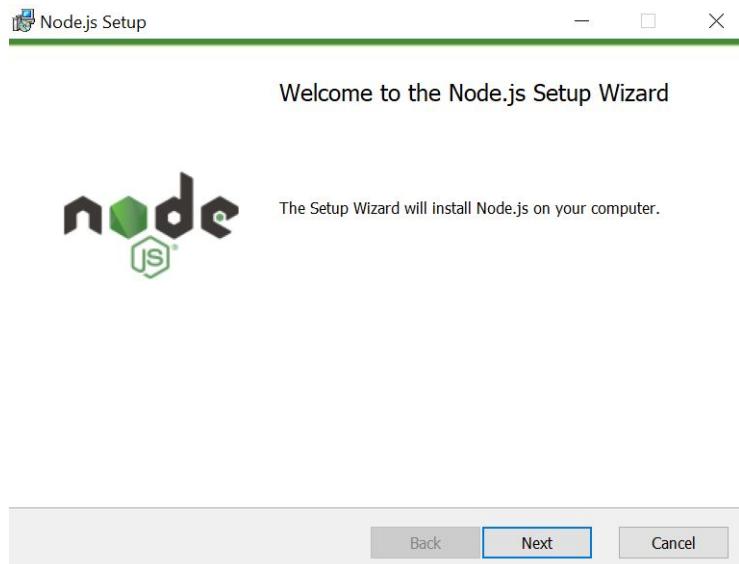
Una vez el proceso de instalación finalice te mostrará un resumen con lo que ha instalado en tu ordenador, presiona el botón cerrar (*Close*).



En este momento ya tendrás instalado Node.js en tu ordenador.

Instalación en Microsoft Windows

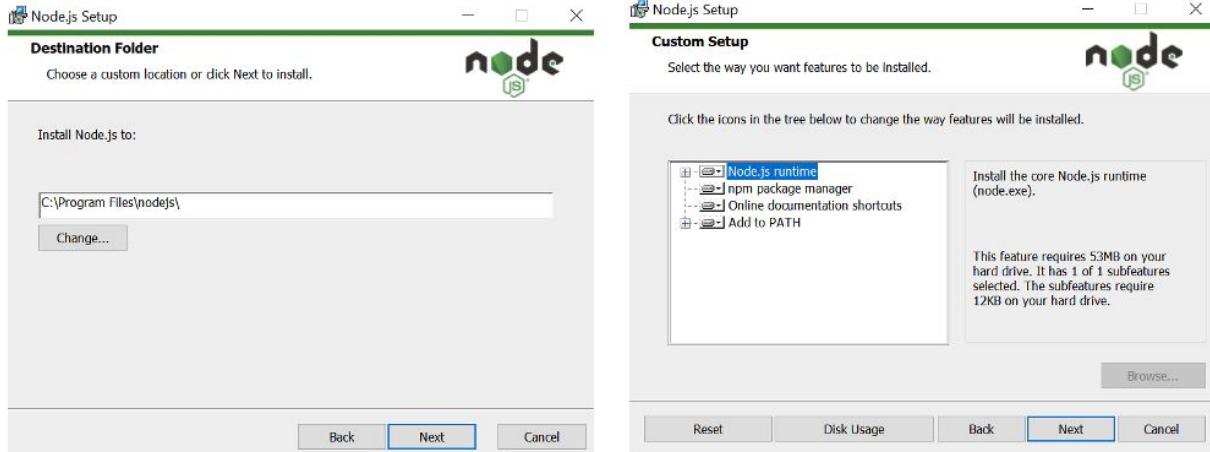
Para instalar Node.js en sistemas operativos Microsoft Windows lo primero que tienes que ejecutar es el fichero que acabas de descargar. Una vez lo ejecutes verás la siguiente pantalla de bienvenida, presiona el botón siguiente (*Next*):



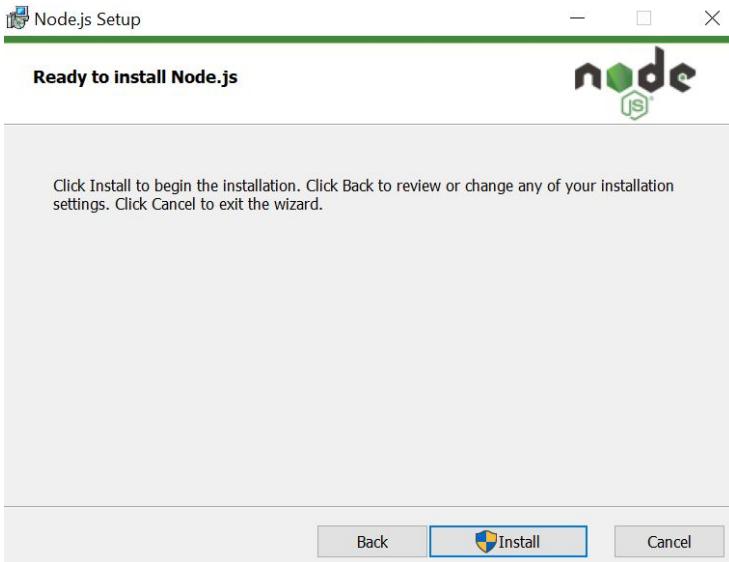
La siguiente pantalla que se muestra es el acuerdo de licencia de uso, presiona el botón siguiente (*Next*):



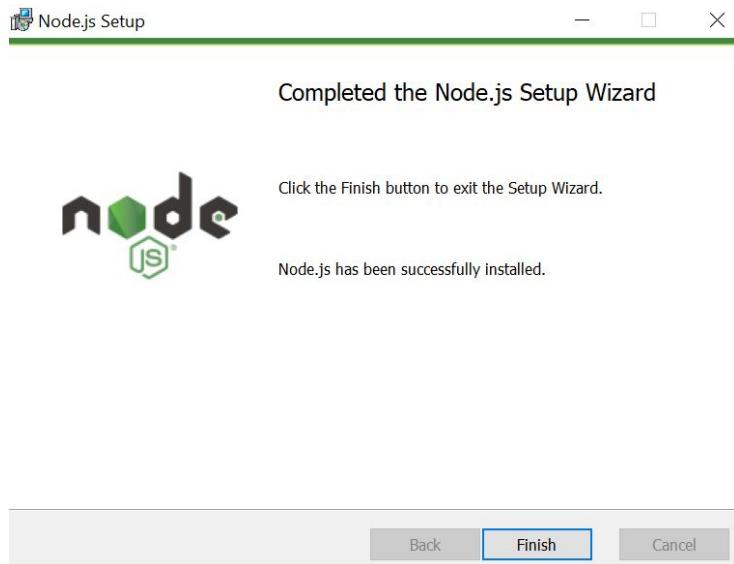
Las siguientes pantallas que se muestran son para cambiar la carpeta de instalación y la configuración de la instalación. Puedes cambiarlas si quieres, aunque nosotros te recomendamos que no cambies nada. En ambos casos presiona el botón siguiente (*Next*):



La siguiente pantalla que aparecerá será la de empezar la instalación, presiona el botón instalar (*Install*):



Una vez el proceso de instalación finalice te aparecerá la siguiente pantalla, presiona el botón finalizar (*Finish*):



En este momento ya tendrás instalado Node.js en tu ordenador.

ENTORNO DE DESARROLLO

En este capítulo vamos a explicarte qué es un entorno de desarrollo, cómo instalar el entorno de desarrollo que utilizaremos para el aprendizaje, y por último, una pequeña guía rápida de uso del entorno de desarrollo.

¿Qué es un entorno de desarrollo?

Un entorno de desarrollo es un programa informático que contiene integradas todas las herramientas, utilidades y funcionalidades necesarias para facilitar la tarea de desarrollo de software. Los entornos de desarrollo se conocen con las siglas IDE, “Integrated Development Environment” o “Entorno de Desarrollo Integrado”

En los entornos de desarrollo puedes realizar las siguientes operaciones:

- Crear/Editar proyectos de desarrollo.
- Compilar código fuente.
- Ejecución directa del código fuente.
- Depurar el código fuente.

Los IDE están compuestos por diferentes componentes, aunque existe un conjunto de ellos que se consideran necesarios para considerarlos un entorno de desarrollo integrado:

- **Editor de texto:** componente con el que escribiremos el código fuente.
- **Compilador:** componente que traducirá el código fuente a lenguaje máquina.
- **Depurador:** componente que permitirá la realización de pruebas del código fuente y eliminación de errores.
- **Editor gráfico:** componente que facilitará la creación y diseño de interfaces gráficas.

Los IDE tienen las siguientes características:

- **Multiplataforma:** pueden ser instalados en todos los sistemas operativos.
- **Múltiples idiomas:** están traducidos a una gran cantidad de idiomas.

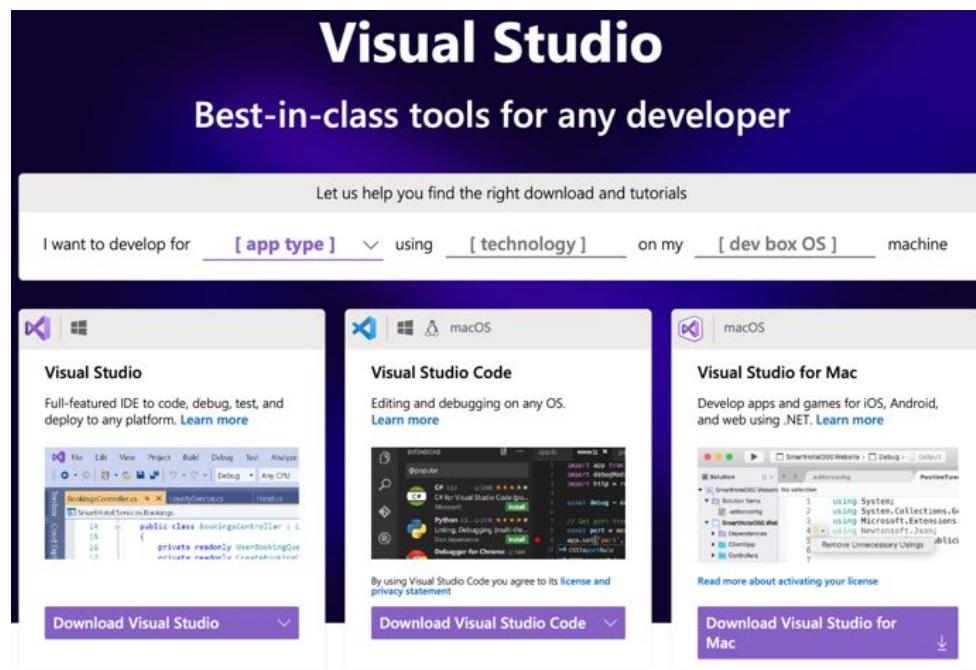
- **Reconocimiento de Sintaxis:** facilitan al desarrollador la escritura de código fuente al ser capaces de reconocer las sintaxis propias del lenguaje de programación con el que están desarrollando.
- **Extensiones y Componentes para el IDE:** dan la posibilidad de añadir extensiones y componentes al IDE para incrementar las funcionalidades de este.
- **Integración con Sistemas de Control de Versiones:** permiten la utilización de sistemas de control de versiones para el código fuente / proyectos / soluciones que tengamos.
- **Depurador:** permiten depurar el código fuente utilizando herramientas disponibles para ello.
- **Importar y Exportar proyectos:** permiten trabajar con proyectos que han sido creados con el mismo IDE pero en instalaciones diferentes.
- Incluyen Manual de Usuarios y Ayuda.

Microsoft Visual Studio Code

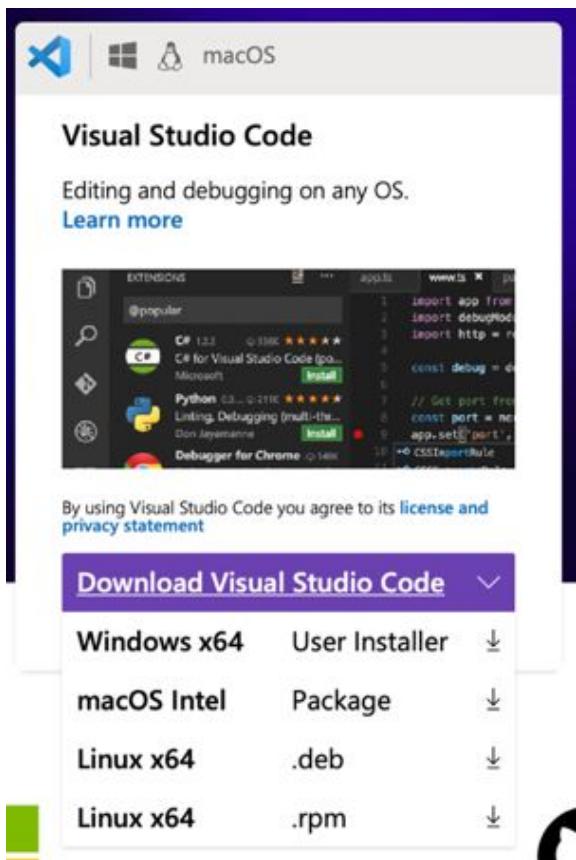
El entorno de desarrollo que utilizaremos durante el aprendizaje es **Microsoft Visual Studio Code**. El entorno te va a permitir escribir todos los ejercicios que hagamos en las fases y ejecutarlos.

Microsoft Visual Studio Code fue lanzado a finales del año 2015 por Microsoft y supuso una revolución en el desarrollo de software por ser un entorno ligero, extensible y con soporte para múltiples lenguajes.

La descarga de Visual Studio Code la tienes que realizar desde esta dirección: <https://visualstudio.microsoft.com>. Al entrar en la página web verás lo siguiente:



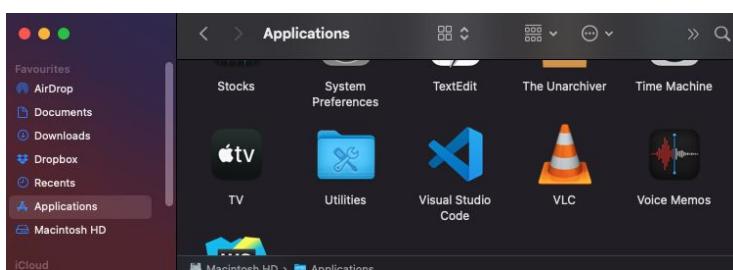
La opción que descargaremos de las 3 es la de Visual Studio Code, al presionar en el desplegable tendrás que elegir la versión del sistema operativo que tienes para descargar la versión para el mismo:



Una vez descargada estaremos preparados para proceder con la instalación.

Instalación en macOS

La instalación en sistemas operativos macOS no es necesario realizarla ya que la descarga es el propio entorno de desarrollo, no es un paquete de instalación. Lo único que deberás de hacer es copiar el fichero que se ha descargado en la carpeta de aplicaciones:

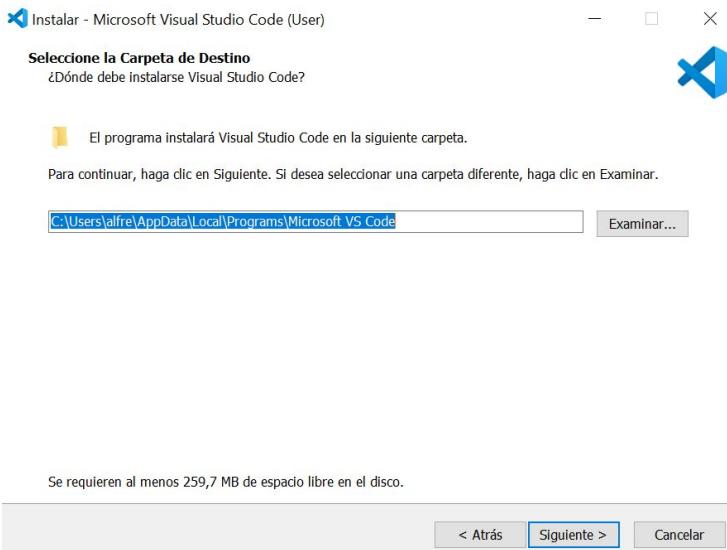


Instalación en Microsoft Windows

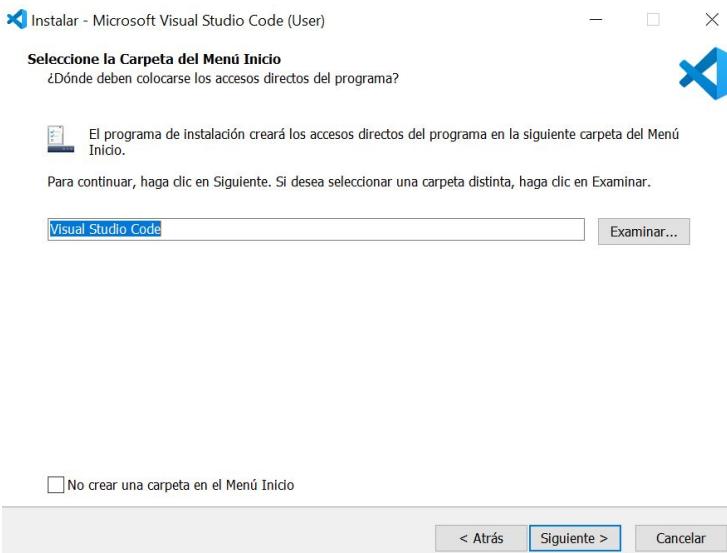
Para instalar Visual Studio Code en sistemas operativos Microsoft Windows lo primero que tienes que ejecutar es el fichero que acabas de descargar. Una vez lo ejecutes verás la siguiente pantalla con la licencia de uso del software. Selecciona “Acepto el acuerdo” y presiona el botón siguiente:



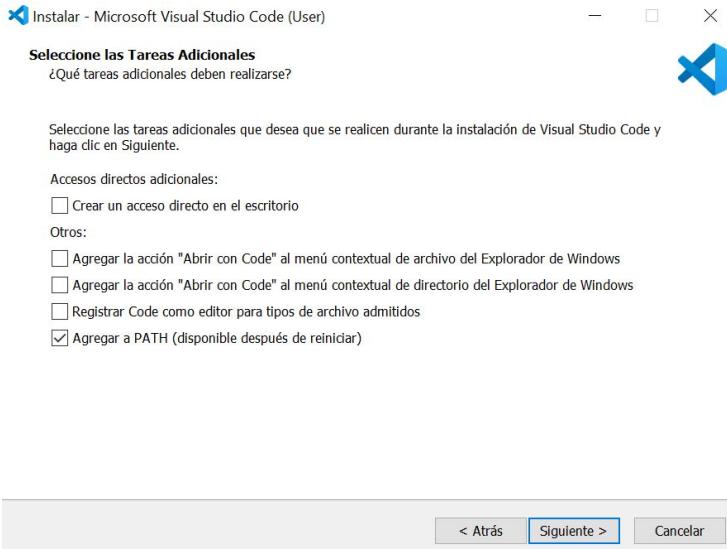
La siguiente pantalla que verás es la de seleccionar la carpeta donde se instalará Visual Studio Code, deja la que viene por defecto o selecciona otra si así lo deseas. Presiona el botón siguiente:



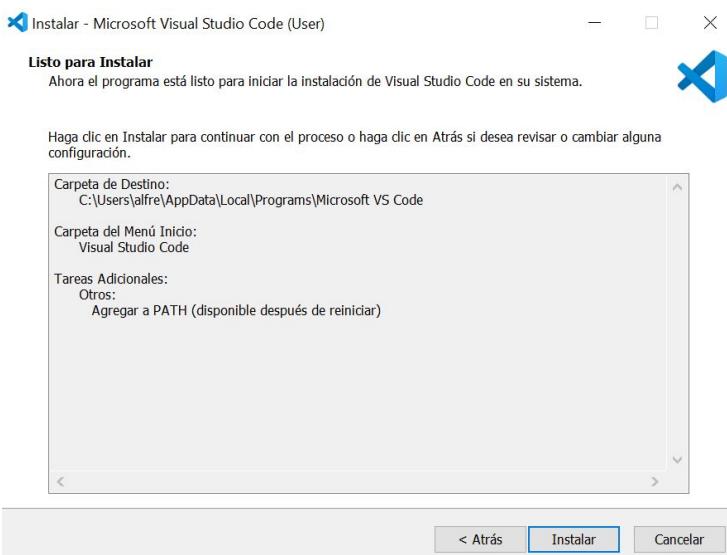
La siguiente pantalla que aparecerá es la de seleccionar si quieras añadir un elemento en el menú de inicio para abrir Visual Studio Code desde ahí. Ponle el nombre que quieras y si quieras añadirlo o no y a continuación presiona el botón siguiente:



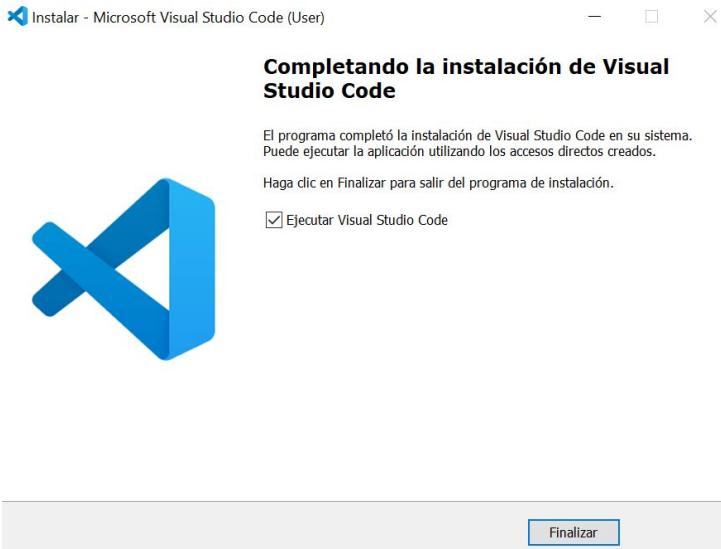
La siguiente pantalla muestra opciones extra de instalación, te recomendamos seleccionar la de crear un acceso directo en el escritorio. Presiona el botón siguiente:



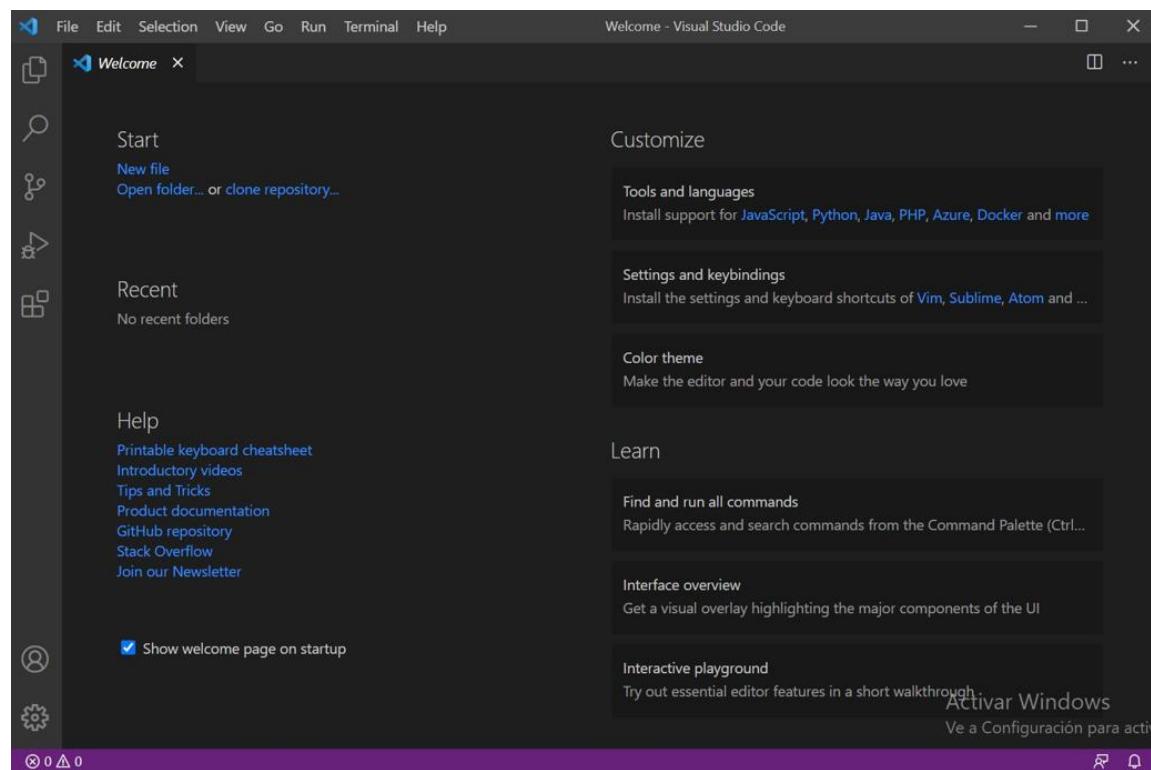
La siguiente pantalla que verás es la de iniciar el proceso de instalación, presiona el botón instalar:



Por último, una vez acabe la instalación, verás la siguiente pantalla. Presiona el botón finalizar:



Una vez instalado Visual Studio Code puedes abrirlo y verás la siguiente pantalla:

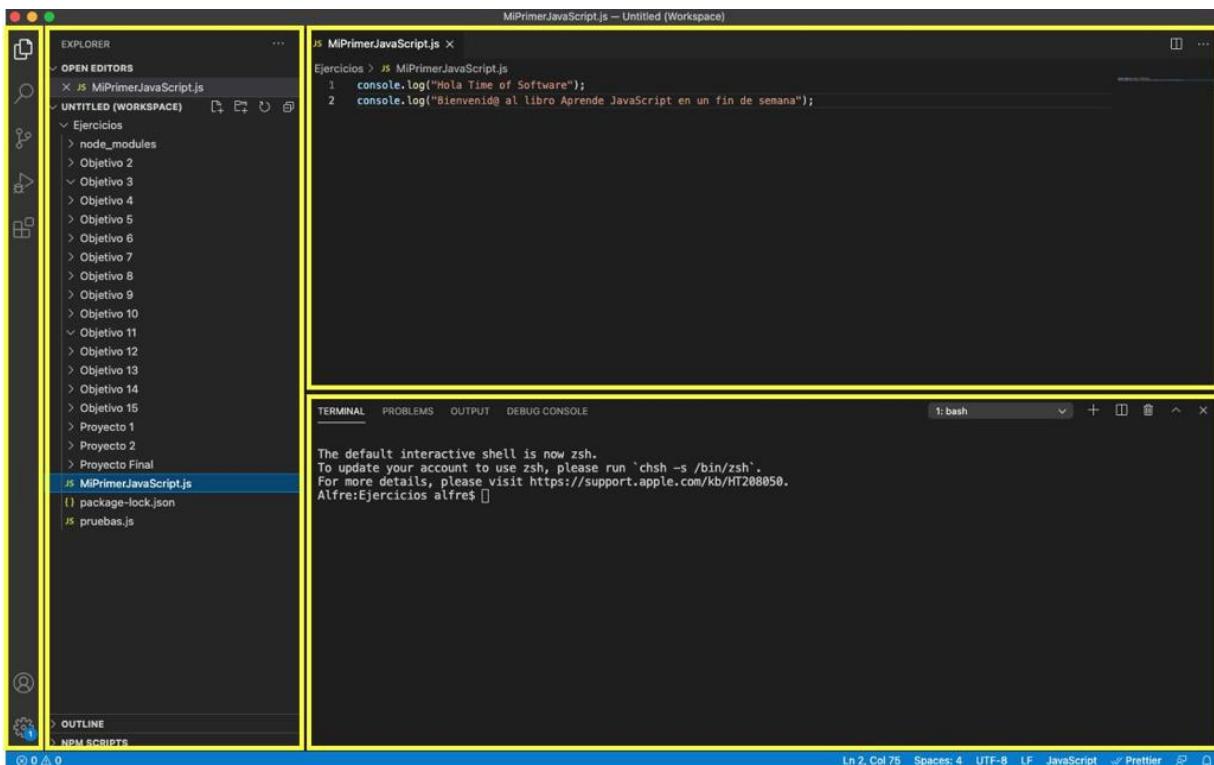


Guía básica de uso

Visual Studio Code está compuesto principalmente por cuatro paneles diferentes:

- Utilidades. (Panel izquierdo).
- Explorador de ficheros. (Panel a la derecha del anterior).
- Editor de código. (Panel derecho superior).
- Terminal / Salida / Errores / Consola. (Panel derecho inferior).

Los paneles los tienes resaltados en la siguiente imagen:



Veamos cada panel en detalle.

Panel utilidades

El panel de utilidades es el panel que nos permite acceder a las diferentes opciones de uso que tiene Visual Studio Code. La selección de una utilidad u otra hará que cambie el panel de “Explorador de ficheros” al correspondiente a esa utilidad. Aunque únicamente utilizaremos la utilidad de explorador de ficheros, vamos a explicarte las diferentes utilidades que incluye Visual Studio Code por defecto.

- **Explorador**: utilidad que permite trabajar con ficheros y carpetas.
- **Búsqueda**: utilidad que permite realizar búsquedas en los ficheros y carpetas que tenemos cargados en Visual Studio Code.
- **Control de código fuente**: utilidad que permite integrar nuestro código fuente con un gestor de código fuente.
- **Ejecución y Debug**: utilidad que añade funcionalidades extra a la ejecución y depuración de código fuente.
- **Extensiones**: utilidad que permite añadir extensiones a Visual Studio Code. Algunas extensiones añadirán un nuevo ícono a este panel, incrementando las utilidades disponibles de Visual Studio Code.

Explorador de ficheros

En el explorador de ficheros se muestran las carpetas y ficheros que tienes cargados en Visual Studio Code. El explorador tiene dos secciones principales, en la sección superior podrás acceder rápidamente a los ficheros que tienes abiertos en el editor de código y en la inferior tienes todas las carpetas y ficheros cargados. Para abrir ficheros en el editor de código simplemente tienes que hacer clic los ficheros y se abrirán en el editor; es posible tener más de un fichero abierto a la vez en el editor de código.

Editor de código

El editor de código te va a permitir editar los ficheros de código fuente que abras.

Terminal / Salida / Errores / Consola

El panel está compuesto por diversos subpaneles:

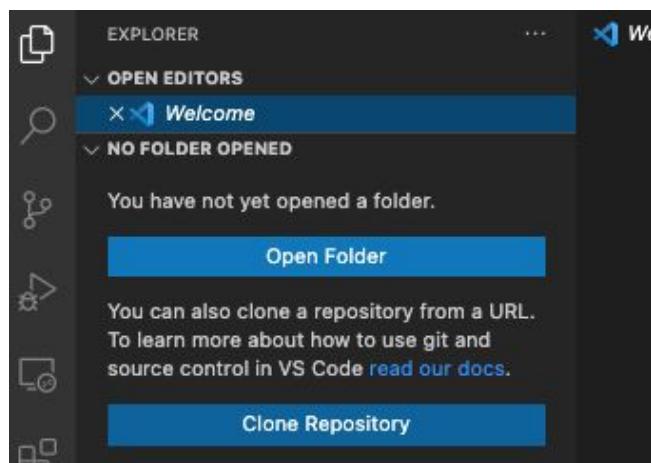
- **Terminal**: es un terminal del sistema operativo integrado en Visual Studio Code y que se abre por defecto en la carpeta que tienes cargada en el explorador de ficheros.
- **Salida**: muestra el resultado de la ejecución de los programas.
- **Errores**: muestra los errores durante la ejecución de los programas.
- **Consola debug**: muestra los mensajes durante la ejecución en modo depuración de los programas. Este panel no lo utilizaremos en el libro.

MI PRIMER JAVASCRIPT

En este capítulo vamos a explicarte cómo realizar tu primer programa con JavaScript paso a paso. En todos los lenguajes de programación normalmente el primer programa que se realiza es el famoso “*Hola Mundo*”. Se trata de un programa muy básico mediante el cual aprendes el proceso de creación y ejecución de programas.

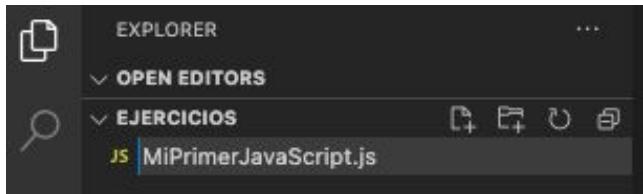
Antes de empezar debes de crear una carpeta que será la carpeta que utilices para ir guardando todos los ejercicios que vayas realizando durante todo el libro. A la carpeta puedes llamarla *Ejercicios*.

Al abrir Visual Studio Code lo primero que te aparece es esto:



Presiona el botón “Open Folder” y selecciona la carpeta que acabas de crear.

El siguiente paso es crear el fichero en el que escribirás tu primer programa en JavaScript. Para crear el fichero presiona el botón de la izquierda del conjunto de cuatro botones que aparecen. El botón creará un nuevo fichero dentro de la carpeta, renómbralo a “*MiPrimerJavaScript.cs*”. La siguiente imagen muestra lo que verás en Visual Studio Code:



Para abrir el fichero en el editor de código fuente tienes que hacer clic o doble clic sobre este. Ahora llega el momento de escribir el código fuente de tu primer ejercicio. Seguramente tengas dudas de lo que estás haciendo, o no entiendas lo que escribes, pero confía en nosotros, simplemente hazlo, durante el libro lo entenderás. Escribe el siguiente código fuente en el editor:

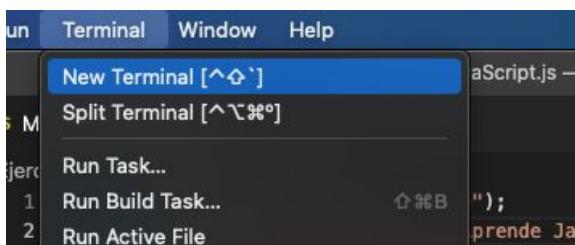
```
console.log("Hola Time of Software");
console.log("Bienvenid@ al libro Aprende JavaScript en un fin de semana");
```

Una vez has escrito el código, guarda el fichero.

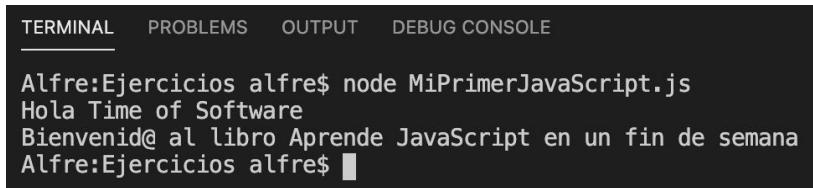
El siguiente paso es ejecutar el programa que acabas de escribir. Para ello, existen dos opciones:

1. Utilizar el terminal integrado en Visual Studio Code.
2. Ejecutarlo utilizando las funcionalidades que ofrece Visual Studio Code.

La primera forma de ejecución es la más sencilla. En el menú principal de Visual Studio Code hay una opción que te permite abrir el terminal integrado:



Una vez abierto el terminal tienes que escribir el siguiente comando: “`node MiPrimerJavaScript.js`” y presionar *Enter*. El programa se ejecutará:

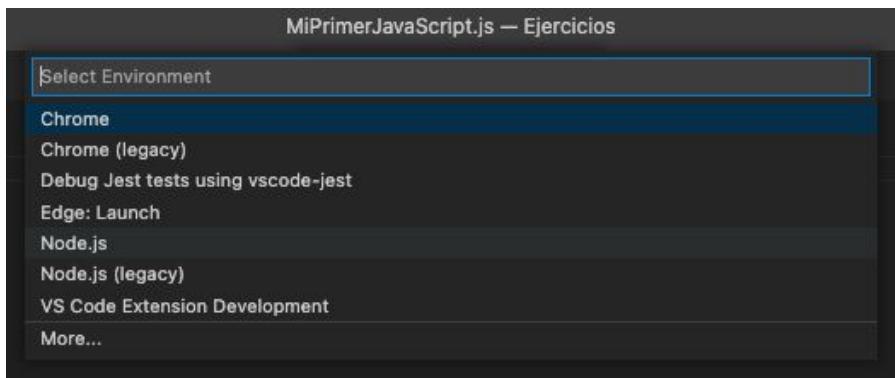


```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
Alfre:Ejercicios alfre$ node MiPrimerJavaScript.js
Hola Time of Software
Bienvenid@ al libro Aprende JavaScript en un fin de semana
Alfre:Ejercicios alfre$
```

La segunda forma de ejecución no es que sea más difícil, es que requiere de un paso extra. En el menú principal de Visual Studio Code hay una opción que se llama “*Run*” o “*Ejecutar*”, elige el submenú “*Run without debugging*” o “*Ejecutar sin depurar*”:



Acto seguido Visual Studio Code te preguntará qué tiene que utilizar para ejecutar el programa, selecciona la opción *Node.js*.



En la siguiente imagen puedes ver el resultado de la ejecución del programa utilizando la opción 2:

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
/usr/local/bin/node ./MiPrimerJavaScript.js
Hola Time of Software
Bienvenid@ al libro Aprende JavaScript en un fin de semana
```

OBJETIVO 1: VARIABLES Y CONSTANTES

En este primer objetivo vamos a explicarte qué son las variables y las constantes, cómo ponerles nombre y cómo declararlas en el código fuente. Es un objetivo teórico pero esencial a la hora de programar y que utilizaremos en el resto de los objetivos.

Conceptos teóricos

En este apartado vamos a explicarte todo lo que necesitas saber para aprender a utilizar variables y constantes.

Variables

Las variables son datos que necesitas almacenar y utilizar en los programas y que residen en la memoria del ordenador. Tienen las siguientes características:

- **Nombre:** identificador dentro del código fuente que utilizamos para usarlas.
- **Valor:** valor que almacena la variable.

Un ejemplo de uso de variables puede ser la necesidad de almacenar en tu programa la edad del usuario. Para ello, crearías una variable con un nombre concreto para que almacene el valor de la edad.

En JavaScript los nombres de las variables empiezan siempre por una letra (mayúscula o minúscula) y pueden contener tantas letras (mayúsculas o minúsculas) y números como necesites. A diferencia con otros lenguajes, tienes que tener en cuenta que las letras minúsculas y las mayúsculas son letras diferentes, una variable con el nombre *edad* será diferente a otra llamada *Edad*.

La declaración de variables en JavaScript se realiza utilizando la palabra *let*. Veamos unos ejemplos de declaración de variables:

- *let edad;*
- *let nombre, apellidos;*

La primera sentencia declara una variable cuyo nombre es *edad* y la segunda declara dos variables cuyos nombres son *nombre* y *apellidos*.

Operador asignación

El operador de asignación ‘=’ sirve para asignar un valor a una variable o constante, lo que está en la parte derecha del operador será asignado (almacenado) en la variable de la parte izquierda. Veamos unos ejemplos:

- *edadUsuario = 36;*
- *nombreUsuario = “Alfredo”;*

En el primer ejemplo se está asignando el valor 36 a la variable *edadUsuario* y en el segundo ejemplo se está asignando la cadena de texto “Alfredo” a la variable *nombreUsuario*.

La declaración de variables puede contener una iniciación a un valor de la variable, para ello se utiliza el operador asignación de la siguiente manera:

- *let edad = 36;*
- *let nombre = “Alfredo”;*

Constantes

Las constantes son como las variables pero con una gran diferencia, que es que la constante no puede cambiar de valor durante la ejecución del programa.

La declaración de constantes en JavaScript se realiza utilizando la palabra **const** en vez de **let**. A diferencia de las variables, las

constantes **siempre** tienen que ser inicializadas con un valor.
Veamos unos ejemplos de declaración de constantes:

- `const pi = 3.1416;`
- `const velocidadLuz = 300000;`

Ahora eres capaz de...

En este primer objetivo has aprendido los siguientes conocimientos:

- Definición de variables.
- Definición de constantes.
- Utilización del operador asignación.

OBJETIVO 2: MOSTRANDO INFORMACIÓN

El segundo objetivo del libro consiste en el aprendizaje y uso de la salida de información a través de la pantalla.

Mostrar información por pantalla y leer información de los usuarios (lo veremos en el siguiente objetivo) son operaciones necesarias para conseguir una interactividad alta en las aplicaciones que desarrolles por parte de los usuarios de éstas.

El objetivo está compuesto por una única fase en la que aprenderás diferentes formas de mostrar información a través de la pantalla.

FASE 1: Mostrando información

En la primera fase aprenderás a mostrar información a los usuarios, para ello utilizarás la siguiente sentencia:

console.log(TextoAMostrar);

El primer ejercicio de la fase consiste en aprender a mostrar mensajes de texto directamente en la consola. El código fuente del ejercicio es el siguiente:

```
console.log("Línea 1 del mensaje que se muestra por la consola");
console.log("Línea 2 del mensaje que se muestra por la consola");
```

Tal y como puedes observar, el texto a mostrar se escribe entre comillas en la sentencia.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_1.js
Línea 1 del mensaje que se muestra por la consola
Línea 2 del mensaje que se muestra por la consola
Alfre:Objetivo 2 alfre$ []
```

El segundo ejercicio de la fase consiste en aprender a mostrar texto almacenado en una variable. El código fuente es el siguiente:

```
let texto = "Primer texto de la variable";
console.log(texto);
texto = "Segundo texto de la variable";
console.log(texto);
```

Al igual que en el ejercicio anterior, el texto a mostrar se asigna a la variable escribiéndolo entre comillas.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_2
Primer texto de la variable
Segundo texto de la variable
Alfre:Objetivo 2 alfre$ []
```

El tercer ejercicio de la fase consiste en mostrar texto y variables en una misma sentencia. Para ello, añadirás a *console.log* el texto y la variable separadas por coma. El código fuente es el siguiente:

```
let texto = "Alfredo";
console.log("Mi nombre es", texto);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_3
Mi nombre es Alfredo
Alfre:Objetivo 2 alfre$ []
```

El cuarto ejercicio de la fase consiste en aprender a mostrar más de una variable a la vez en el mensaje. Para ello, añadirás a *console.log* las variables separadas por coma. El código fuente es el siguiente:

```
let texto1 = "Texto 1";
let texto2 = "Texto 2";
console.log(texto1, texto2);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_4
Texto 1 Texto 2
Alfre:Objetivo 2 alfre$ []
```

El quinto ejercicio de la fase consiste en aprender a incluir texto en mitad de otro texto para que su valor se muestre en esa posición. En la cadena de texto hay que escribir los caracteres **%s** en la posición en la que quieres que aparezca el texto que quieres incluir y añadir ambos textos a *console.log* tal y como hemos hecho anteriormente. El código fuente es el siguiente:

```
let texto = "36";
console.log("Mi edad es %s años",texto);
```

El valor de la variable *texto* será mostrado en el lugar de la cadena indicado por `%s`.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_5
Mi edad es 36 años
Alfre:Objetivo 2 alfre$
```

El sexto ejercicio de la fase consiste en extender el ejercicio anterior añadiendo más de una inclusión de texto en una misma cadena. Para ello vas a añadir tantos `%s` en la cadena como valores quieras introducir en la misma y lo añadirás todo a la sentencia `console.log`. El primer `%s` será sustituido por la primera cadena de texto en los parámetros, el segundo `%s` por la segunda y así sucesivamente. El código fuente es el siguiente:

```
let texto = "36";
console.log("Mi edad es %s años y vivo en %s",texto,"Madrid");
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 2 alfre$ node Ej2_6
Mi edad es 36 años y vivo en Madrid
Alfre:Objetivo 2 alfre$
```

Ahora eres capaz de...

En este segundo objetivo has adquirido los siguientes conocimientos:

- Mostrar información por pantalla.
- Utilización de variables.

OBJETIVO 3: ENTRADA DE INFORMACIÓN

El tercer objetivo del libro consiste en el aprendizaje y uso de la entrada de información por parte de los usuarios de las aplicaciones.

El objetivo está compuesto por dos fases. En la primera aprenderás a instalar paquetes de Node.js y en la segunda aprenderás a leer información introducida por los usuarios.

FASE 1: Instalación de paquetes de NodeJS

La primera fase del objetivo consiste en aprender a instalar paquetes de NodeJS en tu proyecto.

La instalación de paquetes se realiza mediante un comando en el terminal de Visual Studio Code. El comando es el siguiente:

npm install NombrePaquete

Para poder leer información por parte de los usuarios instalaremos el paquete *prompt-sync* utilizando el siguiente comando:

npm install prompt-sync

La siguiente imagen muestra la ejecución del comando en el terminal de Visual Studio Code:

```
Alfre:Ejercicios alfre$ npm install prompt-sync
npm WARN saveError ENOENT: no such file or directory, open '/Users/alfre/Dropbox/Libro JavaScript/Ejercicios/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/Users/alfre/Dropbox/Libro JavaScript/Ejercicios/package.json'
npm WARN Ejercicios No description
npm WARN Ejercicios No repository field.
npm WARN Ejercicios No README data
npm WARN Ejercicios No license field.

+ prompt-sync@4.2.0
added 3 packages from 3 contributors and audited 3 packages in 2.068s
found 0 vulnerabilities
```

FASE 2: Lectura de información

La segunda fase del objetivo consiste en aprender a leer información utilizando el paquete que has instalado en la fase anterior.

El primer y único ejercicio de la fase consiste en leer una cadena de texto que introduzca el usuario y mostrarla por pantalla.

Lo primero que tienes que hacer en el código fuente para poder utilizar el paquete que instalamos en la fase anterior es incluirlo con la sentencia siguiente:

```
const prompt = require('prompt-sync')();
```

La sentencia te permitirá utilizar el paquete a través de la constante *prompt*. La lectura de la información se realiza de la siguiente forma:

```
Variable = prompt(TextoAMostrar);
```

Veámoslo en detalle:

- **Variable**: almacenará el texto leído que ha introducido el usuario.
- **prompt**: es la constante que nos permite utilizar el paquete de Node.js.
- **TextoAMostrar**: texto que se mostrará al usuario junto con la acción de leer información.

El código fuente del ejercicio es el siguiente:

```
const prompt = require('prompt-sync')();

let informacionLeida = prompt('Escribe algo: ');
console.log("Has escrito:", informacionLeida)
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 3 alfre$ node Ej3_1
Escribe algo: Me llamo Alfredo
Has escrito: Me llamo Alfredo
Alfre:Objetivo 3 alfre$ []
```

Ahora eres capaz de...

En este tercer objetivo has adquirido los siguientes conocimientos:

- Inclusión y utilización de paquetes de Node.js.
- Lectura de información introducida por el usuario.

OBJETIVO 4: NÚMEROS

El cuarto objetivo del libro consiste en el aprendizaje y uso de los números, tanto enteros como decimales, los operadores aritméticos y los operadores de asignación compuestos.

El objetivo está compuesto por cuatro fases. En la primera aprenderás a utilizar los operadores aritméticos con números enteros, en la segunda el uso de paréntesis en las operaciones aritméticas, en la tercera cómo usar los operadores de asignación compuestos y en la última aprenderás a utilizar los números decimales.

Conceptos teóricos

En este apartado vamos a explicarte la teoría de todo lo que necesitas saber para aprender a utilizar los números y los operadores aritméticos.

Números enteros y decimales

En este objetivo vamos a aprender a manejar dos tipos de números, los números enteros y los números decimales, veamos en qué se caracterizan ambos:

- **Número entero:** los números enteros son aquellos números compuestos por los números naturales, sus inversos negativos y el cero. Por ejemplo: -3, -2, -1, 0, 1, 2, 3, etc.
- **Número decimales:** los números decimales son números que tienen dos partes y que estas se encuentran separadas por una coma. La parte a la izquierda de la coma es la parte entera y la parte a la derecha de la coma es la parte decimal.

Operadores aritméticos

Los operadores aritméticos son aquellos operadores que nos van a permitir realizar operaciones aritméticas con los datos. JavaScript permite los siguientes operadores aritméticos:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
**	Exponente
++	Incremento
--	Decremento

Veamos unos ejemplos:

- `resultadomultiplicacion = 8 * 4`
- `coste = (6*11)-4`
- `numerochucheriasporpersona = 50/4`

En el primer ejemplo, el resultado de la multiplicación se almacenará en la variable `resultadomultiplicacion`. En el segundo ejemplo, el resultado de la operación será almacenado en la variable `coste`, y por último, el resultado de la división será almacenado en la variable `numerochucheriasporpersona`.

Una buena práctica a la hora de utilizar operadores aritméticos es la utilización de paréntesis para establecer el orden concreto de resolución de las operaciones. Cada lenguaje de programación establece la resolución de las operaciones aritméticas de forma diferente y puedes encontrar resultados diferentes dependiendo del lenguaje.

Además de los operadores aritméticos mostrados anteriormente, JavaScript posee una serie de operadores de asignación compuestos que son los siguientes:

Operador	Ejemplo	Equivale a
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>
<code>**=</code>	<code>x **= y</code>	<code>x = x ** y</code>

FASE 1: Números enteros y operaciones aritméticas

La primera fase de este objetivo consiste en aprender a usar números enteros y operaciones aritméticas sencillas.

El primer ejercicio de la fase consiste en realizar una suma, una resta, una multiplicación, una división, un módulo y un exponente de dos números. El código fuente es el siguiente:

```
let numero1 = 8;
let numero2 = 3;
let resultadosuma = numero1 + numero2;
let resultadoresta = numero1 - numero2;
let resultadomultiplicacion = numero1 * numero2;
let resultadodivision = numero1 / numero2;
let resultadomodulo = numero1 % numero2;
let resultadoexponente = numero1 ** numero2;
console.log("Número 1:", numero1);
console.log("Número 2:", numero2);
console.log("Resultado suma:",resultadosuma);
console.log("Resultado resta:",resultadoresta);
console.log("Resultado multiplicacion:",resultadomultiplicacion);
console.log("Resultado división:",resultadodivision);
console.log("Resultado módulo:",resultadomodulo);
console.log("Resultado exponente:",resultadoexponente);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_1
Número 1: 8
Número 2: 3
Resultado suma: 11
Resultado resta: 5
Resultado multiplicacion: 24
Resultado división: 2.6666666666666665
Resultado módulo: 2
Resultado exponente: 512
Alfre:Objetivo 4 alfre$
```

El segundo ejercicio de la fase consiste en realizar el mismo ejercicio, pero solicitando al usuario que introduzca los dos números. Para realizar las operaciones aritméticas es necesario

convertir la información que introduce el usuario a entero, para ello utilizarás el comando *parseInt*, que tiene la siguiente sintaxis:

parseInt(CadenaDeTexto)

La cadena de texto será la información que introduzca el usuario a través de la sentencia de lectura de información *prompt*. El resultado tienes que asignárselo a una variable para poder utilizarlo posteriormente.

El código fuente del ejercicio es el siguiente:

```
const prompt = require('prompt-sync')();

let numero1 = parseInt(prompt('Escribe el primer número: '));
let numero2 = parseInt(prompt('Escribe el segundo número: '));
let resultadosuma = numero1 + numero2;
let resultadoresta = numero1 - numero2;
let resultadomultiplicacion = numero1 * numero2;
let resultadodivision = numero1 / numero2;
let resultadomodulo = numero1 % numero2;
let resultadoexponente = numero1 ** numero2;
console.log("Número 1:", numero1);
console.log("Número 2:", numero2);
console.log("Resultado suma:",resultadosuma);
console.log("Resultado resta:",resultadoresta);
console.log("Resultado multiplicacion:",resultadomultiplicacion);
console.log("Resultado división:",resultadodivision);
console.log("Resultado módulo:",resultadomodulo);
console.log("Resultado exponente:",resultadoexponente);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_2
Escribe el primer número: 6
Escribe el segundo número: 3
Número 1: 6
Número 2: 3
Resultado suma: 9
Resultado resta: 3
Resultado multiplicacion: 18
Resultado división: 2
Resultado módulo: 0
Resultado exponente: 216
Alfre:Objetivo 4 alfre$
```

El tercer ejercicio de la fase consiste en aprender a utilizar los operadores incremento (++) y decremento (--). El código fuente es el siguiente:

```
let variable = 7;
console.log("Valor:", variable);
variable++;
console.log("Valor después del incremento:", variable);
variable--;
console.log("Valor después del decremento:", variable);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_3
Valor: 7
Valor después del incremento: 8
Valor después del decremento: 7
Alfre:Objetivo 4 alfre$
```

FASE 2: Uso de paréntesis

La segunda fase del objetivo consiste en aprender a utilizar paréntesis en las operaciones aritméticas. Tal y como te indicamos en la parte teórica del objetivo, te recomendamos utilizar paréntesis a la hora de hacer operaciones aritméticas complejas. De esta forma te asegurarás de que las operaciones se ejecutan en el orden que deseas.

El primer y único ejercicio de la fase consiste en aplicar paréntesis de diferentes formas a una misma expresión aritmética compleja y que compruebes que, dependiendo del orden de ejecución establecido por el uso de paréntesis, el resultado será diferente.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero1 = parseInt(prompt('Escribe el primer número: '));
let numero2 = parseInt(prompt('Escribe el segundo número: '));
let numero3 = parseInt(prompt('Escribe el tercer número: '));
let numero4 = parseInt(prompt('Escribe el cuarto número: '));

let resultado1 = (numero1 * numero2) + numero3 / numero4;
let resultado2 = numero1 * (numero2 + numero3 / numero4);
let resultado3 = numero1 * (numero2 + numero3) / numero4;

console.log("Operación:      (%s      *      %s)      +      %s      /      %s      = %s",numero1,numero2,numero3,numero4,resultado1);
console.log("Operación:      %s      *      (%s      +      %s)      /      %s      = %s",numero1,numero2,numero3,numero4,resultado2);
console.log("Operación:      %s      *      (%s      +      %s)      /      %s      = %s",numero1,numero2,numero3,numero4,resultado3);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_4
Escribe el primer número: 4
Escribe el segundo número: 4
Escribe el tercer número: 4
Escribe el cuarto número: 4
Operación:  $(4 * 4) + 4 / 4 = 17$ 
Operación:  $4 * (4 + 4 / 4) = 20$ 
Operación:  $4 * (4 + 4) / 4 = 8$ 
Alfre:Objetivo 4 alfre$ []
```

FASE 3: Operadores asignación compuestos

La tercera fase del objetivo consiste en aprender a utilizar los operadores de asignación compuestos que explicamos en la parte teórica.

El primer y único ejercicio de la fase consiste en utilizar todos los operadores con dos números que se definen al comienzo del programa. Ten en cuenta que al tratarse de operadores de asignación el resultado de la operación es asignado al primer número, por tanto, deberás de establecer su valor al valor original antes de cada operación si quieras seguir utilizando los valores definidos al principio del programa. El código fuente es el siguiente:

```
let numero1 = 8;
let numero2 = 3;
numero1 += numero2;
console.log("numero1+=numero2:",numero1);
numero1 = 8;
numero1 -= numero2;
console.log("numero1-=numero2:",numero1);
numero1 = 8;
numero1 *= numero2;
console.log("numero1*=numero2:",numero1);
numero1 = 8;
numero1 /= numero2;
console.log("numero1/=numero2:",numero1);
numero1 = 8;
numero1 %= numero2;
console.log("numero1%=numero2:",numero1);
numero1 = 8;
numero1 **= numero2;
console.log("numero1**=numero2:",numero1);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_5
numero1+=numero2: 11
numero1-=numero2: 5
numero1*=numero2: 24
numero1/=numero2: 2.6666666666666665
numero1%=numero2: 2
numero1**=numero2: 512
Alfre:Objetivo 4 alfre$ []
```

FASE 4: Número decimales

La cuarta fase de este objetivo consiste en aprender a usar números decimales del mismo modo que utilizaste los números enteros en las fases anteriores.

El primer ejercicio de la fase consiste en realizar las mismas operaciones aritméticas del primer ejercicio de la primera fase, pero esta vez con números decimales. El código fuente es el siguiente:

```
let numero1 = 9.34;
let numero2 = 3.1;
let resultadosuma = numero1 + numero2;
let resultadoresta = numero1 - numero2;
let resultadomultiplicacion = numero1 * numero2;
let resultadodivision = numero1 / numero2;
let resultadomodulo = numero1 % numero2;
let resultadoexponente = numero1 ** numero2;
console.log("Número 1:", numero1);
console.log("Número 2:", numero2);
console.log("Resultado suma:",resultadosuma);
console.log("Resultado resta:",resultadoresta);
console.log("Resultado multiplicación:",resultadomultiplicacion);
console.log("Resultado división:",resultadodivision);
console.log("Resultado módulo:",resultadomodulo);
console.log("Resultado exponente:",resultadoexponente);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_6
Número 1: 9.34
Número 2: 3.1
Resultado suma: 12.44
Resultado resta: 6.24
Resultado multiplicacion: 28.954
Resultado división: 3.0129032258064514
Resultado módulo: 0.03999999999999959
Resultado exponente: 1018.7680497522394
Alfre:Objetivo 4 alfre$
```

El segundo ejercicio de la fase consiste en realizar el mismo ejercicio, pero solicitando al usuario que introduzca los dos números decimales. Para realizar las operaciones aritméticas es necesario

convertir la información que introduce el usuario a decimal, para ello utilizarás el comando *parseFloat*, que tiene la siguiente sintaxis:

parseFloat(CadenaDeTexto)

La cadena de texto será la información que introduzca el usuario a través de la sentencia de lectura de información *prompt*. El resultado tienes que asignárselo a una variable para poder utilizarlo posteriormente.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero1 = parseFloat(prompt('Escribe el primer número con decimales: '));
let numero2 = parseFloat(prompt('Escribe el segundo número con decimales: '));
let resultadosuma = numero1 + numero2;
let resultadoresta = numero1 - numero2;
let resultadomultiplicacion = numero1 * numero2;
let resultadodivision = numero1 / numero2;
let resultadomodulo = numero1 % numero2;
let resultadoexponente = numero1 ** numero2;
console.log("Número 1:", numero1);
console.log("Número 2:", numero2);
console.log("Resultado suma:",resultadosuma);
console.log("Resultado resta:",resultadoresta);
console.log("Resultado multiplicacion:",resultadomultiplicacion);
console.log("Resultado división:",resultadodivision);
console.log("Resultado módulo:",resultadomodulo);
console.log("Resultado exponente:",resultadoexponente);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 4 alfre$ node Ej4_7
Escribe el primer número con decimales: 6.79
Escribe el segundo número con decimales: 2.5
Número 1: 6.79
Número 2: 2.5
Resultado suma: 9.29
Resultado resta: 4.29
Resultado multiplicacion: 16.975
Resultado división: 2.716
Resultado módulo: 1.79
Resultado exponente: 120.13635074339449
Alfre:Objetivo 4 alfre$ []
```


Ahora eres capaz de...

En este cuarto objetivo has adquirido los siguientes conocimientos:

- Utilización de números enteros.
- Utilización de operadores aritméticos.
- Conversión de cadenas de texto en números enteros.
- Utilización de paréntesis en operaciones aritméticas complejas.
- Utilización de operadores de asignación compuestos.
- Utilización de números decimales.

OBJETIVO 5: CADENAS DE TEXTO

El quinto objetivo del libro consiste en el aprendizaje y uso de las cadenas de texto en JavaScript.

El objetivo está compuesto por dos objetivos, en el primero aprenderás operaciones básicas sobre las cadenas de texto, cómo crearlas, manipulaciones sencillas, etc. En el segundo aprenderás operaciones propias de las cadenas de texto.

FASE 1: Manejo básico de cadenas

La primera fase del objetivo consiste en aprender diferentes operaciones básicas sobre la creación y manipulación de las cadenas de texto.

El primer ejercicio de la fase consiste en aprender diferentes formas de mostrar cadenas de texto en JavaScript. El ejercicio está compuesto por cuatro sentencias que muestran un texto por pantalla. En la primera sentencia el texto utiliza comillas dobles en la sentencia y en la segunda utiliza comillas simples. En JavaScript es posible incluir en las cadenas de texto comillas siempre y cuando sean diferentes a las utilizadas para la definición de la cadena, es decir, dentro de una cadena definida con comillas dobles podremos incluir comillas simples y viceversa; la tercera y cuarta instrucción del ejercicio representan esto que acabamos de explicarte.

El código fuente del ejercicio es el siguiente:

```
console.log("Mensaje con comillas dobles");
console.log('Mensaje con comillas simples');
console.log("Mensaje con comillas dobles contenido 'comillas simples'");
console.log('Mensaje con comillas simples contenido "comillas dobles"');
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_1
Mensaje con comillas dobles
Mensaje con comillas simples
Mensaje con comillas dobles contenido 'comillas simples'
Mensaje con comillas simples contenido "comillas dobles"
Alfre:Objetivo 5 alfre$
```

El segundo ejercicio de la fase consiste en aprender el uso del carácter escape en las cadenas de texto. El carácter escape te va a permitir por ejemplo añadir comillas dobles al texto definido con comillas dobles, lo mismo para comillas simples. El carácter escape es el carácter \ y la forma de utilizarlo es incluyéndolo en la cadena

seguido del carácter de las comillas. Veámoslo con un ejemplo, el código fuente es el siguiente:

```
console.log("Mensaje con \"comillas dobles\"");
console.log('Mensaje con \'comillas simples\'');
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_2
Mensaje con "comillas dobles"
Mensaje con 'comillas simples'
Alfre:Objetivo 5 alfre$
```

El tercer ejercicio de la fase consiste en aprender a utilizar el carácter escape para incluir caracteres especiales que tiene JavaScript. Los caracteres especiales que se pueden incluir son los siguientes:

- **Barra inversa:** se representa por el carácter \\
- **Nueva línea:** se representa por el carácter \\n
- **Retorno de carro:** se representa por el carácter \\r
- **Tabulación vertical:** se representa por el carácter \\v
- **Tabulación:** se representa por el carácter \\t
- **Retroceso:** se representa por el carácter \\b
- **Avance de página:** se representa por el carácter \\f

En el ejercicio vas a incluir la barra inversa, el salto de línea, la tabulación y la tabulación vertical. El código fuente del ejercicio es el siguiente:

```
console.log("Esta es una cadena compuesta\npor dos líneas y que incluye \\");
console.log("");
console.log("Cadena con dos líneas\n\ty la segunda tabulada");
console.log("");
console.log("Cadena con dos líneas\n\t\ty la segunda tabulada verticalmente");
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_3
Esta es una cadena compuesta
por dos líneas y que incluye \
Cadena con dos líneas
    y la segunda tabulada
Cadena con dos líneas
    y la segunda tabulada
Alfre:Objetivo 5 alfre$ []
```

El cuarto ejercicio de la fase consiste en aprender a concatenar cadenas de texto para formar cadenas de texto más grandes. La forma de concatenar cadenas es utilizando el operador `+` entre las dos cadenas. Las cadenas de texto pueden ser variables o cadenas de texto directamente. El código fuente del ejercicio es el siguiente:

```
let cadena1 = "Cadena 1" + " Cadena 2";
console.log(cadena1);
let cadena2 = cadena1 + " Cadena 3";
console.log(cadena2);
let cadena3 = cadena1 + " " + cadena2;
console.log(cadena3);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_4
Cadena 1 Cadena 2
Cadena 1 Cadena 2 Cadena 3
Cadena 1 Cadena 2 Cadena 1 Cadena 2 Cadena 3
Alfre:Objetivo 5 alfre$ []
```

El quinto ejercicio de la fase consiste en aprender a definir cadenas que son muy grandes y que en lugar de estar escritas en una única línea (cosa que haría el código ilegible) están escritas en varias líneas para facilitar la lectura del código. Para definir cadenas en diferentes líneas se utiliza el carácter de la barra invertida `\` al final de la línea. El código fuente del ejercicio es el siguiente:

```
let cadena = "Esta es una cadena que la hemos \
dividido en varias líneas porque \
si no lo hiciéramos el código no \
```

```
se podría leer fácilmente";  
console.log(cadena);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_5  
Esta es una cadena que la hemos dividido en varias líneas porque si no lo hiciéramos el código no se podría leer fácilmente  
Alfre:Objetivo 5 alfre$ ]
```

FASE 2: Métodos propios de las cadenas

La segunda fase del objetivo consiste en aprender a utilizar diferentes métodos propios y propiedades que el tipo de dato cadena de texto posee. Mediante una serie de métodos y propiedades vas a poder manipular el contenido de las cadenas de texto y saber características de estas.

Los métodos siguen normalmente el siguiente formato:

CadenaDeTexto.Método(Parámetros);

Veámoslos en detalle:

- **CadenaDeTexto:** cadena de texto sobre la que se ejecutará el método.
- **Método:** método que se ejecutará sobre la cadena de texto.
- **Parámetros:** valores que se utilizarán para ejecutar el método. Los parámetros son opcionales en los métodos y dependerán del método que se quiera ejecutar.

El primer ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **length:** propiedad que devuelve el número de caracteres que tiene la cadena de texto. Fíjate en el código fuente que la forma de usar *length* es diferente a los métodos, no lleva ni los paréntesis ni los parámetros.
- **charAt:** método que devuelve el carácter que ocupa la posición indicada como parámetro en la cadena de texto.
- **charCodeAt:** método que devuelve el código ASCII del carácter que ocupa la posición indicada como parámetro en la cadena de texto.

- []: no es un método propiamente dicho, pero permite acceder a la posición que ocupa el carácter entre los corchetes en la cadena de texto. Es similar al método *charAt*.

El código fuente del ejercicio es el siguiente:

```
let cadena = "En un lugar de la mancha";
console.log("Número de caracteres:", cadena.length)
console.log("Carácter en la posición 4: ", cadena.charAt(4));
console.log("Carácter en la posición 8: ", cadena.charCodeAt(8));
console.log("Carácter en la posición 19: ", cadena[19]);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_6
Número de caracteres: 24
Carácter en la posición 4: n
Carácter en la posición 8: 103
Carácter en la posición 19: a
Alfre:Objetivo 5 alfre$ []
```

El segundo ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **indexOf**: método que devuelve la primera ocurrencia de la cadena pasada como parámetro en la cadena. El método tiene un segundo parámetro opcional que permite indicar la posición de la cadena por la cual empezar a buscar. En caso de no encontrar la cadena devuelve el valor -1.
- **lastIndexOf**: método que devuelve la primera ocurrencia de la cadena pasada como parámetro en la cadena empezando la búsqueda desde el final de la cadena. El método tiene un segundo parámetro opcional que permite indicar la posición de la cadena por la cual empezar a buscar. En caso de no encontrar la cadena devuelve el valor -1.

- **search**: método que hace básicamente lo mismo que *indexOf* pero que no permite incluir un segundo parámetro para indicar la posición para empezar a buscar.

El código fuente del ejercicio es el siguiente:

```
let cadena = "En un lugar de la mancha";
console.log("Cadena:", cadena);
console.log("Posición de la cadena 'lugar' (indexOf):", cadena.indexOf("lugar"));
console.log("Posición de la cadena 'lugar' (search):", cadena.search("lugar"));
cadena = "¿En qué lugar aparece la palabra 'lugar'?";
console.log("Cadena:", cadena);
console.log("indexOf 'lugar' sin segundo parámetro:", cadena.indexOf("lugar"));
console.log("indexOf 'lugar' con segundo parámetro 20:", cadena.indexOf("lugar",20));
console.log("lastIndexOf 'lugar' sin segundo parámetro:", cadena.lastIndexOf("lugar"));
console.log("lastIndexOf 'lugar' con segundo parámetro 20:", cadena.lastIndexOf("lugar",20));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_7
Cadena: En un lugar de la mancha
Posición de la cadena 'lugar' (indexOf): 6
Posición de la cadena 'lugar' (search): 6
Cadena: ¿En qué lugar aparece la palabra 'lugar'?
indexOf 'lugar' sin segundo parámetro: 8
indexOf 'lugar' con segundo parámetro 20: 34
lastIndexOf 'lugar' sin segundo parámetro: 34
lastIndexOf 'lugar' con segundo parámetro 20: 8
Alfre:Objetivo 5 alfre$ ]
```

El tercer ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **slice**: método que devuelve una cadena de texto a partir de la cadena original. El método recibe dos parámetros, el primero es obligatorio e indica la primera posición de la cadena en la cadena original, el segundo es opcional e indica la última posición de la cadena en la cadena original, que será hasta el final de la cadena si no se incluye dicho parámetro. Es posible introducir valores

negativos, esto indicará que se empezará a contar desde el final de la cadena de texto.

- **substring**: método que hace exactamente lo mismo que el método *slice*, pero sin admitir valores negativos en los parámetros.
- **substr**: método que devuelve una cadena de texto a partir de la cadena original, pero en el que los parámetros son el índice en el que empieza la cadena y la longitud de esta. Si no se especifica el segundo parámetro se devolverá hasta el final de la cadena.

El código fuente del ejercicio es el siguiente:

```
let cadena = "Coche, Moto, Avión, Helicóptero";
console.log("Cadena:", cadena);
console.log("slice 7 - 11:", cadena.slice(7,11));
console.log("slice 13 - final:", cadena.slice(13));
console.log("slice (-18) - (-13):", cadena.slice(-18,-13));
console.log("substring 7 - 11:", cadena.substring(7,11));
console.log("substring 13 - final:", cadena.substring(13));
console.log("substr 7 - 4:", cadena.substr(7,4));
console.log("substr 13 - final:", cadena.substr(13));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_8
Cadena: Coche, Moto, Avión, Helicóptero
slice 7 - 11: Moto
slice 13 - final: Avión, Helicóptero
slice (-18) - (-13): Avión
substring 7 - 11: Moto
substring 13 - final: Avión, Helicóptero
substr 7 - 4: Moto
substr 13 - final: Avión, Helicóptero
Alfre:Objetivo 5 alfre$ []
```

El cuarto ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **toUpperCase**: método que convierte todos los caracteres de la cadena a mayúsculas.

- **toLowerCase**: método que convierte todos los caracteres de la cadena a minúsculas.

El código fuente del ejercicio es el siguiente:

```
let cadena = "en un lugar de la mancha";
console.log("Cadena:", cadena);
console.log("Cadena mayúsculas:", cadena.toUpperCase());
cadena = "EN UN LUGAR DE LA MANCHA";
console.log("Cadena:", cadena);
console.log("Cadena minúsculas:", cadena.toLowerCase());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_9
Cadena: en un lugar de la mancha
Cadena mayúsculas: EN UN LUGAR DE LA MANCHA
Cadena: EN UN LUGAR DE LA MANCHA
Cadena minúsculas: en un lugar de la mancha
Alfre:Objetivo 5 alfre$
```

El quinto ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **concat**: método que devuelve una cadena formada por la cadena que ejecuta el método unida a las cadenas pasadas como parámetros.
- **replace**: método que devuelve la cadena habiendo realizado el reemplazo de la primera cadena de los parámetros por la segunda en la cadena original. El método únicamente realiza el reemplazo de la primera ocurrencia en la cadena y distingue entre mayúsculas y minúsculas, por lo que tendrás que especificar la cadena a sustituir tal y como aparece en la cadena.

El código fuente del ejercicio es el siguiente:

```
let cadena = "uno, dos, tres";
console.log("Cadena:", cadena);
cadena = cadena.concat(", cuatro, cinco");
```

```
console.log("Cadena:", cadena);
let cadena2 = "seis, siete";
cadena = cadena.concat(", ", cadena2);
console.log("Cadena:", cadena);
cadena = cadena.replace("dos", "two");
console.log("Cadena reemplazada:", cadena);
cadena = cadena.replace("Uno", "one");
console.log("Cadena que no reemplaza Uno:", cadena);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_10
Cadena: uno, dos, tres
Cadena: uno, dos, tres, cuatro, cinco
Cadena: uno, dos, tres, cuatro, cinco, seis, siete
Cadena reemplazada: uno, two, tres, cuatro, cinco, seis, siete
Cadena que no reemplaza Uno: uno, two, tres, cuatro, cinco, seis, siete
Alfre:Objetivo 5 alfre$
```

El sexto ejercicio de la fase consiste en aprender los siguientes métodos de la clase string:

- **trim**: método que elimina todos los espacios en blanco al principio y al final de la cadena texto.
- **startsWith**: método que indica si la cadena empieza por la cadena pasada por parámetros.
- **endsWith**: método que devuelve si la cadena termina por la cadena pasada por parámetros.

El código fuente del ejercicio es el siguiente:

```
let cadena = " uno, dos tres ";
console.log("Cadena:", cadena, ".");
cadena = cadena.trim();
console.log("Cadena:", cadena, ".");
console.log("¿Empieza por uno?:", cadena.startsWith("uno"));
console.log("¿Empieza por dos?:", cadena.startsWith("dos"));
console.log("¿Termina por tres?:", cadena.endsWith("tres"));
console.log("¿Termina por dos?:", cadena.endsWith("dos"));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_11
Cadena: uno, dos tres .
Cadena: uno, dos tres .
¿Empieza por uno?: true
¿Empieza por dos?: false
¿Termina por tres?: true
¿Termina por dos?: false
Alfre:Objetivo 5 alfre$ []
```

El séptimo ejercicio de la fase consiste en aprender a convertir números en cadenas de texto. El método que se utiliza para esto es `toString()` y forma parte de todas las variables en JavaScript, independientemente del tipo de dato que contengan. El código fuente es el siguiente:

```
let numero1 = 79;
let numero2 = 4.56;
console.log("Entero convertido a string:", numero1.toString());
console.log("Real convertido a string:", numero2.toString());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 5 alfre$ node Ej5_12
Entero convertido a string: 79
Real convertido a string: 4.56
Alfre:Objetivo 5 alfre$ []
```

Ahora eres capaz de...

En este quinto objetivo has adquirido los siguientes conocimientos:

- Utilización de cadenas de texto.
- Utilización de métodos propios de las cadenas de texto.

OBJETIVO 6: FECHAS

El sexto objetivo del libro consiste en el aprendizaje y uso del tipo de datos fecha en JavaScript.

El objetivo está compuesto por una única fase en la que aprenderás a utilizar diferentes aspectos de las fechas.

FASE 1: Fechas

El primer ejercicio de la fase consiste en aprender a crear fechas y mostrarlas por pantalla. Existen diferentes formas de crear fechas, en el libro vamos a aprender las formas más utilizadas.

La primera forma de crear una fecha es utilizando el método de creación sin utilizar ningún parámetro, lo que creará una fecha con la fecha actual.

La segunda forma de crear una fecha es pasándole los parámetros de año, mes, día, horas, minutos, segundos y milisegundos al método de creación en este orden. El resto de formas es modificando la segunda forma de creación incluyendo menos parámetros, estas son las posibilidades:

- Año, mes, día, hora, minuto y segundo.
- Año, mes, día, hora y minuto.
- Año, mes, día y hora.
- Año, mes y día.
- Año y mes.

Puedes omitir todos los parámetros a excepción de año y mes a la hora de utilizar el método de creación de fechas.

Tienes que tener en cuenta que para los meses en JavaScript se utilizan los valores de 0 a 11, es decir, Enero será el 0, Febrero será el 1, etc, hasta llegar a Diciembre que será el 11.

Para mostrar una fecha por pantalla hay que utilizar el método `toString()`. Existe un método que permite mostrar la fecha como cadena de texto omitiendo la corrección horaria y mostrar la fecha UTC (fecha sin corrección), el método es `toUTCString()`.

El código fuente del ejercicio es el siguiente:

```
let fechaactual = new Date();
let fechapasada = new Date(2016,1,21,20,30,0,0);
console.log("Fecha actual:", fechaactual.toString());
console.log("Fecha pasada:", fechapasada.toString());
fechapasada = new Date(2016,1,21,20,30,0);
console.log("Fecha pasada 2:", fechapasada.toString());
fechapasada = new Date(2016,1,21,20,30);
console.log("Fecha pasada 3:", fechapasada.toString());
fechapasada = new Date(2016,1,21,20);
console.log("Fecha pasada 4:", fechapasada.toString());
fechapasada = new Date(2016,1,21);
console.log("Fecha pasada 5:", fechapasada.toString());
fechapasada = new Date(2016,1);
console.log("Fecha UTC:", fechaactual.toUTCString())
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 6 alfre$ node Ej6_1
Fecha actual: Wed Mar 17 2021 05:39:01 GMT+0100 (Central European Standard Time)
Fecha pasada: Sun Feb 21 2016 20:30:00 GMT+0100 (Central European Standard Time)
Fecha pasada 2: Sun Feb 21 2016 20:30:00 GMT+0100 (Central European Standard Time)
Fecha pasada 3: Sun Feb 21 2016 20:30:00 GMT+0100 (Central European Standard Time)
Fecha pasada 4: Sun Feb 21 2016 20:00:00 GMT+0100 (Central European Standard Time)
Fecha pasada 5: Sun Feb 21 2016 00:00:00 GMT+0100 (Central European Standard Time)
Fecha UTC: Wed, 17 Mar 2021 04:39:01 GMT
Alfre:Objetivo 6 alfre$ []
```

El segundo ejercicio de la fase consiste en aprender a utilizar los métodos que tiene el tipo de datos fecha para obtener los valores del año, mes, etc. Los métodos son los siguientes:

- **getFullYear**: devuelve el valor del año como entero de 4 dígitos.
- **getMonth**: devuelve el valor del mes. Recuerda que el valor devuelto va de 0 a 11.
- **getDate**: devuelve el valor del día en valor de 1 a 31.
- **getHours**: devuelve el valor de la hora en valor de 0 a 23.
- **getMinutes**: devuelve el valor de los minutos en valor de 0 a 59.
- **getSeconds**: devuelve el valor de los segundos en valor de 0 a 59.

- **getMilliseconds**: devuelve el valor de los milisegundos en valor de 0 a 999.
- **getDay**: devuelve el valor del día de la semana en valor de 0 a 6. El valor 0 será el domingo, el 1 el lunes, etc.

Además, en el ejercicio vas a aprender a mostrar la fecha como objeto de tipo fecha, para ello únicamente tienes que añadir la variable sin utilizar el método *toString()* al comando *console.log*. La fecha será mostrada en formato UTC, sin la corrección horaria aplicada.

El código fuente del ejercicio es el siguiente:

```
let fecha = new Date();
console.log("Fecha:", fecha);
console.log("Fecha a cadena:", fecha.toString());
console.log("Año:", fecha.getFullYear());
console.log("Mes:", fecha.getMonth());
console.log("Día:", fecha.getDate());
console.log("Hora:", fecha.getHours());
console.log("Minutos:", fecha.getMinutes());
console.log("Segundos:", fecha.getSeconds());
console.log("Milisegundos:", fecha.getMilliseconds());
console.log("Día de la semana:", fecha.getDay())
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 6 alfre$ node Ej6_2
Fecha: 2021-03-17T04:51:24.789Z
Fecha a cadena: Wed Mar 17 2021 05:51:24 GMT+0100 (Central European Standard Time)
Año: 2021
Mes: 2
Día: 17
Hora: 5
Minutos: 51
Segundos: 24
Milisegundos: 789
Día de la semana: 3
Alfre:Objetivo 6 alfre$
```

El tercer ejercicio de la fase consiste en aprender a utilizar los métodos que tiene el tipo de datos fecha para modificar los valores del año, mes, etc. Los métodos son los siguientes:

- **setFullYear**: modifica el valor del año. El valor pasado como parámetro tiene que ser un entero de 4 dígitos.
- **setMonth**: modifica el valor del mes. El valor pasado como parámetro tiene que ser un valor de 0 a 11.
- **setDate**: modifica el valor del día. El valor pasado como parámetro tiene que ser un valor de 1 a 31.
- **setHours**: modifica el valor de la hora. El valor pasado como parámetro tiene que ser un valor de 0 a 23.
- **setMinutes**: modifica el valor de los minutos. El valor pasado como parámetro tiene que ser un valor de 0 a 59.
- **setSeconds**: modifica el valor de los segundos. El valor pasado como parámetro tiene que ser un valor de 0 a 59.
- **setMilliseconds**: modifica el valor de los milisegundos. El valor pasado como parámetro tiene que ser un valor de 0 a 999.

El código fuente del ejercicio es el siguiente:

```
let fecha = new Date();
console.log("Fecha:", fecha.toString());
fecha.setFullYear(2000);
console.log("Fecha (Año):", fecha.toString());
fecha.setMonth(0);
console.log("Fecha (Mes):", fecha.toString());
fecha.setDate(1);
console.log("Fecha (Día):", fecha.toString());
fecha.setHours(8);
console.log("Fecha (Hora):", fecha.toString());
fecha.setMinutes(30);
console.log("Fecha (Minutos):", fecha.toString());
fecha.setSeconds(54);
console.log("Fecha (Segundos):", fecha.toString());
fecha.setMilliseconds(333);
console.log("Fecha (Milisegundos):", fecha.toString());
console.log(fecha);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 6 alfre$ node Ej6_3
Fecha: Wed Mar 17 2021 06:05:09 GMT+0100 (Central European Standard Time)
Fecha (Año): Fri Mar 17 2000 06:05:09 GMT+0100 (Central European Standard Time)
Fecha (Mes): Mon Jan 17 2000 06:05:09 GMT+0100 (Central European Standard Time)
Fecha (Día): Sat Jan 01 2000 06:05:09 GMT+0100 (Central European Standard Time)
Fecha (Hora): Sat Jan 01 2000 08:05:09 GMT+0100 (Central European Standard Time)
Fecha (Minutos): Sat Jan 01 2000 08:30:09 GMT+0100 (Central European Standard Time)
Fecha (Segundos): Sat Jan 01 2000 08:30:54 GMT+0100 (Central European Standard Time)
Fecha (Milisegundos): Sat Jan 01 2000 08:30:54 GMT+0100 (Central European Standard Time)
2000-01-01T07:30:54.333Z
Alfre:Objetivo 6 alfre$ 
```

Ahora eres capaz de...

En este sexto objetivo has adquirido los siguientes conocimientos:

- Utilización de tipos de datos fecha.
- Utilización de métodos propios de los tipos de datos fecha.

OBJETIVO 7: BOOLEANOS

El séptimo objetivo del libro consiste en el aprendizaje y uso de los tipos de datos booleanos. Además, aprenderás a utilizar los operadores relacionales y lógicos.

El objetivo está compuesto por tres fases diferentes. En la primera fase aprenderás a utilizar los booleanos de forma independiente. En la segunda fase aprenderás a utilizar los operadores relacionales y en la tercera los operaciones lógicos.

Conceptos teóricos

En este apartado vamos a explicarte la teoría de todo lo que necesitas saber para aprender a utilizar tipos de datos booleanos, operadores relacionales y operadores lógicos.

Booleanos

El tipo de datos booleano es un tipo de datos que sólo puede tomar dos posibles valores: *True* (verdadero o 1) o *False* (falso o 0).

Operadores relacionales

Los operadores relacionales son aquellos que van a permitirte realizar comparaciones entre dos elementos. Son los siguientes:

Operador	Significado
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual que
!=	Distinto que

El resultado de una operación relacional es un valor booleano, es decir, únicamente tendrá dos posibles valores:

- **true**: la comparación se cumple.
- **false**: la comparación no se cumple.

Veamos algunos ejemplos:

- $7 < 5$

- $9 == 3$
- $2 < 12$
- $88 >= 4$

En el primer ejemplo la comprobación devolverá *false*, en el segundo devolverá *false*, en el tercero devolverá *true* y en el cuarto devolverá *true*.

Operadores lógicos

Los operadores lógicos permiten combinar las operaciones relacionales del punto anterior o valores booleanos independientes para obtener un único resultado. Los operadores lógicos que puedes utilizar son los siguientes:

- **AND**: operador lógico que realiza la operación lógica 'Y' entre dos elementos. El resultado será *true* si ambos elementos son *true*, en caso contrario será *false*.
- **OR**: operador lógico que realiza la operación lógica 'O' entre dos elementos. El resultado será *true* si uno de los dos elementos es *true*, en caso contrario será *false*.
- **NOT**: operador lógico que realiza la operación lógica 'NO'. El resultado será *true* si el elemento es *false*, y será *false* si es *true*.

Los operadores lógicos pueden utilizarse en expresiones combinadas, es por ello que te aconsejamos que utilices paréntesis para separar las diferentes expresiones.

Veamos algunos ejemplos:

- $(5 < 3) \text{ AND } (4 == 7)$
- $(1 < 7) \text{ OR } (3 == 3)$
- $\text{NOT}(6 == 7)$
- True AND False

En el primer ejemplo la comprobación devolverá el valor *false*, en el segundo devolverá el valor *true*, en el tercero devolverá el valor *true* y en el cuarto, *false*.

FASE 1: Booleanos

La primera fase del objetivo consiste en aprender a utilizar tipos de datos booleanos.

El primer y único ejercicio de la fase consiste en aprender a crear variables con valores booleanos. Para ello, inicializaremos las variables de dos formas diferentes:

- Asignando el valor *true* o *false* a las variables.
- Utilizando la sentencia *new Boolean(true)* para asignar el valor *True* a la variable o la sentencia *new Boolean(false)* para asignar el valor *False*.

Fíjate que los valores *true* y *false* son introducidos como valores propiamente dicho, no como cadenas de texto entre comillas; ésto se debe a que dichos valores son parte del lenguaje de programación.

El código fuente es el siguiente:

```
let valortrue = true;
let valorfalse = false;
console.log("Valor true:", valortrue);
console.log("Valor false:", valorfalse);
console.log("Valor true a string:", valortrue.toString());
console.log("Valor false a string:", valorfalse.toString());
valortrue = new Boolean(true);
valorfalse = new Boolean(false);
console.log("Tipo dato boolean con valor true:", valortrue);
console.log("Tipo dato boolean con valor false:", valorfalse);
console.log("Tipo dato boolean con valor true a string:", valortrue.toString());
console.log("Tipo dato boolean con valor false a string:", valorfalse.toString());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 7 alfre$ node Ej7_1
Valor true: true
Valor false: false
Valor true a string: true
Valor false a string: false
Tipo dato boolean con valor true: [Boolean: true]
Tipo dato boolean con valor false: [Boolean: false]
Tipo dato boolean con valor true a string: true
Tipo dato boolean con valor false a string: false
Alfre:Objetivo 7 alfre$ []
```

Fíjate en la diferencia a la hora de mostrar las variables sin convertir a cadena dependiendo del método utilizado para su inicialización.

FASE 2: Operadores relacionales

La segunda fase del objetivo consiste en aprender a utilizar los operadores relacionales.

El primer ejercicio de la fase consiste en utilizar el operador AND. El operador es representado en JavaScript como **&&**. El ejercicio mostrará por pantalla el resultado de realizar una operación AND sobre diferentes valores booleanos.

El código fuente es el siguiente:

```
let valortrue = true;
let valorfalse = false;
console.log("Valor TRUE AND TRUE:", (valortrue && valortrue).toString());
console.log("Valor TRUE AND FALSE:", (valortrue && valorfalse).toString());
console.log("Valor FALSE AND FALSE:", (valorfalse && valorfalse).toString());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 7 alfre$ node Ej7_2
Valor TRUE AND TRUE: true
Valor TRUE AND FALSE: false
Valor FALSE AND FALSE: false
Alfre:Objetivo 7 alfre$
```

El segundo ejercicio de la fase consiste en utilizar el operador OR. El operador es representado en JavaScript como **||**. El ejercicio mostrará por pantalla el resultado de realizar una operación OR sobre diferentes valores booleanos.

El código fuente es el siguiente:

```
let valortrue = true;
let valorfalse = false;
console.log("Valor TRUE OR TRUE:", (valortrue || valortrue).toString());
console.log("Valor TRUE OR FALSE:", (valortrue || valorfalse).toString());
console.log("Valor FALSE OR FALSE:", (valorfalse || valorfalse).toString());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 7 alfre$ node Ej7_3
Valor TRUE OR TRUE: true
Valor TRUE OR FALSE: true
Valor FALSE OR FALSE: false
Alfre:Objetivo 7 alfre$ []
```

El tercer ejercicio de la fase consiste en utilizar el operador NOT. El operador es representado en JavaScript como !. El ejercicio mostrará por pantalla el resultado de realizar una operación NOT sobre los diferentes valores booleanos.

El código fuente es el siguiente:

```
let valortrue = true;
let valorfalse = false;
console.log("NOT TRUE:", (!valortrue).toString());
console.log("NOT FALSE:", (!valorfalse).toString());
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 7 alfre$ node Ej7_4
NOT TRUE: false
NOT FALSE: true
Alfre:Objetivo 7 alfre$ []
```

FASE 3: Operadores lógicos

La tercera fase del objetivo consiste en aprender a utilizar los operadores lógicos.

El primer ejercicio de la fase consiste en la realización de todas las operaciones relacionales que explicamos en la parte teórica del objetivo con los valores que almacenan las variables *numero1* y *numero2*. El resultado de todas las operaciones se muestra por pantalla. El código fuente es el siguiente.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero1 = parseInt(prompt('Escribe el primer número: '));
let numero2 = parseInt(prompt('Escribe el segundo número: '));

console.log("Resultado (", numero1, ">", numero2, "):", (numero1>numero2));
console.log("Resultado (", numero1, " >= ", numero2, "):", (numero1 >= numero2));
console.log("Resultado (", numero1, " < ", numero2, "):", (numero1 < numero2));
console.log("Resultado (", numero1, " <= ", numero2, "):", (numero1 <= numero2));
console.log("Resultado (", numero1, " == ", numero2, "):", (numero1 == numero2));
console.log("Resultado (", numero1, " != ", numero2, "):", (numero1 != numero2));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 7 alfre$ node Ej7_5
Escribe el primer número: 3
Escribe el segundo número: 7
Resultado ( 3 > 7 ): false
Resultado ( 3 >= 7 ): false
Resultado ( 3 < 7 ): true
Resultado ( 3 <= 7 ): true
Resultado ( 3 == 7 ): false
Resultado ( 3 != 7 ): true
Alfre:Objetivo 7 alfre$ []
```

Ahora eres capaz de...

En este séptimo objetivo has adquirido los siguientes conocimientos:

- Utilización de tipos de datos booleanos.
- Utilización de operadores relacionales.
- Utilización de operadores lógicos.

OBJETIVO 8: CONTROL DE FLUJO

El octavo objetivo consiste en familiarizarte y a aprender a utilizar las instrucciones que van a permitirte controlar el flujo de los programas.

El objetivo está compuesto por dos fases. La primera fase consistirá en aprender a utilizar la sentencia *if* y la segunda fase consistirá en aprender a utilizar la sentencia *switch*.

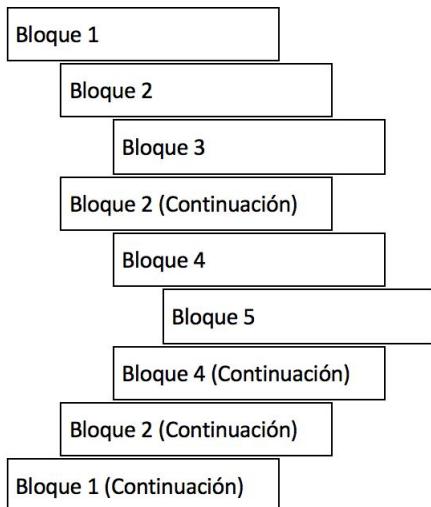
Conceptos teóricos

En este apartado vamos a explicarte la teoría sobre qué son los bloques de instrucciones, la sentencia *if* y la sentencia *switch*.

Bloques de instrucciones

Un bloque de instrucciones es un conjunto de sentencias de código fuente compuesto por una o más sentencias. Los bloques están delimitados por su inicio y su fin, y la forma de delimitarlos es específica de cada lenguaje de programación.

Junto a los bloques de instrucciones, se utilizan indentaciones, que no es otra cosa que mover un bloque de texto hacia la derecha insertando espacios o tabulaciones para así separarlo del margen izquierdo y distinguirlo más fácilmente dentro del texto. Todos los lenguajes utilizan la indentación para aumentar la legibilidad del código fuente. Veamos un ejemplo con una imagen:



En la imagen puedes ver diferentes bloques de código que a su vez tienen otros bloques dentro de ellos. Tal y como puedes ver, un bloque de código puede contener más de un bloque de código dentro, y los bloques internos pueden estar en la mitad del bloque

padre, es decir, que el bloque padre tiene sentencias antes y después del bloque de sentencias interno.

En JavaScript los bloques de código fuente están delimitados por corchetes. Un corchete de apertura indicará que comienza un bloque de instrucciones y un corchete de cierre indicará el cierre del bloque de instrucciones abierto. Existe una excepción, y es que aquellos bloques que únicamente contengan una instrucción se podrán poner sin corchetes.

Sentencia IF

La sentencia *if* nos va a permitir tener diferentes posibles caminos a la hora de ejecutar el código fuente. Cada camino se ejecuta o no en función de una condición o condiciones.

El formato de la sentencia *if* en todos los lenguajes de programación es la siguiente:

```
if (Condición)
    BloqueInstrucciones
else
    BloqueInstrucciones
```

Veámoslo en detalle:

- **if**: te va a permitir generar un bloque de código que se ejecutará si se cumple la condición de entrada que tiene.
- **Condición**: en caso de que la condición tenga como resultado *True* el bloque de instrucciones dentro de la sentencia *if* será ejecutado. En caso contrario, se ejecutará la sentencia *else* en caso de existir.
- **else**: te va a permitir generar un camino alternativo que se ejecutará siempre que no se haya cumplido la condición de la instrucción *if*. No es obligatorio utilizar *else* en las

bifurcaciones, puedes simplemente tener un *if* sin bloque alternativo.

- **BloqueInstrucciones:** conjunto de sentencias que se ejecutarán en cada uno de los casos.

Vamos a verlo con un ejemplo lo más completo posible.

```
numero1 = ValorAleatorio
```

```
numero2 = ValorAleatorio
```

```
if numero1>numero2
    BloqueInstrucciones1
else
    BloqueInstrucciones2
```

En el ejemplo hemos definido dos variables cuyo valor es generado de forma aleatoria. Utilizando esas variables, hemos generado dos posibles caminos:

- *if numero1>numero2*: en caso de que el primer número sea mayor que el segundo se ejecutará el bloque de instrucciones llamado *BloqueInstrucciones1*.
- *else*: en caso de que el primer número sea menor o igual que el segundo número se ejecutará el bloque de instrucciones llamado *BloqueInstrucciones2*.

Por último, indicarte que dentro de los bloques de instrucciones que están dentro de los posibles caminos es posible incluir nuevas sentencias *if*.

Sentencia SWITCH

La sentencia *switch* te va a permitir crear diferentes caminos posibles de ejecución en función de un valor. Básicamente, la sentencia *switch* funciona de forma parecida a como lo hace la

bifurcación con la sentencia *if / else* anidada pero añadiendo cierta funcionalidad adicional.

Switch evalúa una variable y podrá definir tantos caminos como posibles valores pueda tener dicha variable.

El formato de la sentencia *switch* es el siguiente:

```
switch(Variable)
    case X:
        BloqueSentencias
        break
    case Y:
        BloqueSentencias
        break;
    default:
        BloqueSentencias
        break
```

Vamos a verlo en detalle:

- **switch**: indica el comienzo de la instrucción *switch*.
- **Variable**: variable que se evaluará y será utilizada para ejecutar un bloque de sentencias u otro.
- **case**: indica un posible camino dentro de *switch*. X e Y son los valores que deberá tener Variable para ir por un camino u otro.
- **break**: indica el fin de un camino dentro de *switch*.
- **default**: indica el camino por defecto en el caso de que no se pueda tomar ninguno de los caminos especificados por *case*.

A continuación te mostramos un ejemplo de sentencia *switch*:

```
vocal = CaracterAleatorio
switch(vocal)
```

```
case 'a':  
    SentenciasA()  
    break  
case 'e':  
    SentenciasE()  
    break  
case 'i':  
    SentenciasI()  
    break  
case 'o':  
    SentenciasO()  
    break  
case 'u':  
    SentenciasU()  
    break  
default:  
    SentenciasNoVocal()  
    break
```

En el ejemplo se ha utilizado la instrucción *switch* para evaluar la lectura de un carácter. Los posibles caminos que se pueden tomar son los valores de las vocales. Cada uno ejecutará su conjunto de sentencias correspondiente y, para aquellos valores diferentes de las 5 vocales, se ejecutará el conjunto de instrucciones correspondiente a los caracteres no vocales.

FASE 1: Sentencia IF

La primera fase del objetivo consiste en aprender a utilizar la sentencia *if*.

El primer ejercicio de la fase consiste en comprobar si un valor introducido por el usuario es mayor o igual que 10 y mostrar un mensaje indicándolo en caso de que así sea.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero = parseInt(prompt('Escribe un número menor que 10: '));

if(numero>=10)
{
    console.log("¡El número que has escrito es mayor o igual que 10!")
}
console.log("Has escrito el número", numero);
```

La ejecución del código fuente anterior tendrá las siguientes posibles salidas en función del número introducido:

Número mayor o igual que 10:

```
Alfre:Objetivo 8 alfre$ node Ej8_1
Escribe un número menor que 10: 23
¡El número que has escrito es mayor o igual que 10!
Has escrito el número 23
Alfre:Objetivo 8 alfre$
```

Número menor que 10:

```
Alfre:Objetivo 8 alfre$ node Ej8_1
Escribe un número menor que 10: 3
Has escrito el número 3
Alfre:Objetivo 8 alfre$
```

El segundo ejercicio de la fase consiste en extender el primer ejercicio añadiendo la sentencia `else` para mostrar un mensaje en caso de que el número introducido por el usuario no sea mayor o igual que 10.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero = parseInt(prompt('Escribe un número menor que 10: '));

if(numero>=10)
{
    console.log("¡El número que has escrito es mayor o igual que 10!")
}
else
{
    console.log("¡El número que has escrito es menor que 10!")
}
console.log("Has escrito el número", numero);
```

La ejecución del código fuente anterior tendrá las siguientes posibles salidas en función del número introducido:

Número menor que 10:

```
Alfre:Objetivo 8 alfre$ node Ej8_2
Escribe un número menor que 10: 7
¡El número que has escrito es menor que 10!
Has escrito el número 7
Alfre:Objetivo 8 alfre$ []
```

Número mayor o igual que 10:

```
Alfre:Objetivo 8 alfre$ node Ej8_2
Escribe un número menor que 10: 67
¡El número que has escrito es mayor o igual que 10!
Has escrito el número 67
Alfre:Objetivo 8 alfre$ []
```

El tercer ejercicio de la fase consiste en utilizar sentencias `if` y `else` anidadas para evaluar dos números introducidos por el usuario y

decir cuál de los dos es mayor o si son iguales.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let numero1 = parseInt(prompt('Escribe el primer número: '));
let numero2 = parseInt(prompt('Escribe el segundo número: '));

if(numero1 > numero2)
{
    console.log("Resultado comparación: el primer número es mayor que el segundo");
}
else
{
    if(numero2 > numero1)
    {
        console.log("Resultado comparación: el segundo número es mayor que el primero");
    }
    else
    {
        console.log("Resultado comparación: ambos números son iguales");
    }
}
```

La ejecución del código fuente anterior tendrá las siguientes posibles salidas en función de los valores que se introduzcan:

Primer número mayor que el segundo:

```
Alfre:Objetivo 8 alfre$ node Ej8_3
Escribe el primer número: 9
Escribe el segundo número: 4
Resultado comparación: el primer número es mayor que el segundo
Alfre:Objetivo 8 alfre$
```

Ambos números iguales:

```
Alfre:Objetivo 8 alfre$ node Ej8_3
Escribe el primer número: 4
Escribe el segundo número: 4
Resultado comparación: ambos números son iguales
Alfre:Objetivo 8 alfre$
```

Segundo número mayor que el primero:

```
Alfre:Objetivo 8 alfre$ node Ej8_3
Escribe el primer número: 3
Escribe el segundo número: 5
Resultado comparación: el segundo número es mayor que el primero
Alfre:Objetivo 8 alfre$ []
```

FASE 2: Sentencia Switch

La segunda fase del objetivo consiste en aprender a utilizar la sentencia *switch*.

El primer ejercicio de la fase consiste en evaluar mediante la sentencia *switch* un valor introducido por el usuario y mostrar un mensaje dependiendo de dicho valor.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

console.log('Códigos promocionales: ORO, PLATA, BRONCE o PLATINO');
let codigo = prompt('Introduzca su código promocional para saber su descuento: ');
switch(codigo)
{
    case "ORO":
        console.log("El descuento aplicado es del 25%");
        break;
    case "PLATA":
        console.log("El descuento aplicado es del 15%");
        break;
    case "BRONCE":
        console.log("El descuento aplicado es del 5%");
        break;
    case "PLATINO":
        console.log("El descuento aplicado es del 40%");
        break;
    default:
        console.log("Código promocional no válido");
        break;
}
```

La ejecución del código fuente anterior tendrá las siguientes posibles salidas en función del código introducido:

Si se introduce uno de los valores especificados se mostrará el descuento:

```
Alfre:Objetivo 8 alfre$ node Ej8_4
Códigos promocionales: ORO, PLATA, BRONCE o PLATINO
Introduzca su código promocional para saber su descuento: BRONCE
El descuento aplicado es del 5%
Alfre:Objetivo 8 alfre$ 
```

Si se introduce un valor no especificado se mostrará el siguiente mensaje:

```
Alfre:Objetivo 8 alfre$ node Ej8_4
Códigos promocionales: ORO, PLATA, BRONCE o PLATINO
Introduzca su código promocional para saber su descuento: diamante
Código promocional no válido
Alfre:Objetivo 8 alfre$ 
```

El segundo ejercicio de la fase consiste en realizar lo mismo que el ejercicio anterior pero haciéndolo mediante sentencias *if / else* en lugar de un *switch*.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

console.log('Códigos promocionales: ORO, PLATA, BRONCE o PLATINO');
let codigo = prompt('Introduzca su código promocional para saber su descuento: ');
if (codigo == "ORO")
{
    console.log("El descuento aplicado es del 25%");
}
else if(codigo=="PLATA")
{
    console.log("El descuento aplicado es del 15%");
}
else if(codigo=="BRONCE")
{
    console.log("El descuento aplicado es del 5%");
}
else if(codigo=="PLATINO")
{
    console.log("El descuento aplicado es del 40%");
}
else
{
    console.log("Código promocional no válido");
}
```

Si se introduce uno de los valores especificados se mostrará el descuento:

```
Alfre:Objetivo 8 alfre$ node Ej8_5
Códigos promocionales: ORO, PLATA, BRONCE o PLATINO
Introduzca su código promocional para saber su descuento: PLATINO
El descuento aplicado es del 40%
Alfre:Objetivo 8 alfre$ 
```

Si se introduce un valor no especificado se mostrará el siguiente mensaje:

```
Alfre:Objetivo 8 alfre$ node Ej8_5
Códigos promocionales: ORO, PLATA, BRONCE o PLATINO
Introduzca su código promocional para saber su descuento: PLUS
Código promocional no válido
Alfre:Objetivo 8 alfre$ 
```

Ahora eres capaz de...

En este octavo objetivo has adquirido los siguientes conocimientos:

- Utilización de sentencias if.
- Utilización de sentencias switch.

OBJETIVO 9: BUCLES

El noveno objetivo del libro consiste en el aprendizaje y uso de las estructuras de programación conocidas como bucles.

El objetivo está compuesto por cuatro fases. En la primera aprenderás a utilizar los bucles *while*, en la segunda aprenderás a utilizar los bucles *for*, en la tercera aprenderás a utilizar los bucles *do*, y por último, en la cuarta, los bucles anidados.

Conceptos teóricos

En este apartado vamos a explicarte qué es un bucle y los diferentes tipos de bucles que están disponibles en JavaScript.

Bucle

Los bucles consisten en la repetición de la ejecución de un bloque de instrucciones en la que, a cada repetición, se llama iteración. En programación existen diferentes tipos de bucles, cada uno de ellos está recomendado para usarse dentro de un contexto concreto.

En un bucle tienes que especificar lo siguiente:

- Punto de inicio del bucle.
- Punto de fin del bucle.
- Número de iteraciones.

Cada tipo de bucle especifica los puntos anteriores de forma diferente, pero con el mismo significado teórico.

Veamos los diferentes bucles que tenemos disponibles en JavaScript.

WHILE

El tipo de bucle *while* está recomendado para contextos en los que no se sabe exactamente el número de iteraciones que se tienen que ejecutar, pero sí se sabe que hay que ejecutar iteraciones hasta que se deje de cumplir una condición.

La condición que se utiliza para comprobar si se tiene que ejecutar una iteración deberá de ser *true* para que se execute. Si en caso contrario la condición es *false*, la ejecución del bucle finalizará. La

condición es comprobada en cada iteración del bucle. Las variables que se utilizan en la condición del bucle se llaman variables de control.

Los bucles *while* tienen la siguiente sintaxis:

```
while (Condición)
    BloqueInstrucciones
```

Veamos los elementos en detalle:

- **while**: indicador de comienzo del bucle.
- **Condición**: condición que debe de cumplirse para que siga repitiéndose la ejecución del bucle.
- **BloqueInstrucciones**: conjunto de instrucciones que se ejecutarán en cada iteración.

En la utilización de bucles *while* puedes encontrarte con los siguientes problemas:

- Bucles que no se ejecutan nunca: pon especial atención a la inicialización de las variables de control del bucle para asegurarte de que la condición es *true*, ya que si la condición es *false* desde el principio, el bucle jamás se ejecutará.
- Bucles infinitos: pon especial atención a la modificación de los valores de las variables de control del bucle dentro del bucle, ya que, si dichos valores no se ven alterados jamás, el bucle nunca parará de ejecutarse.

FOR

El tipo de bucle *for* está recomendado para contextos en los que se sabe el número de iteraciones exactas que se van a dar en su ejecución, es decir, es un bucle que busca ejecutar un conjunto de

instrucciones de forma repetitiva hasta llegar al número máximo de iteraciones definidas.

Los bucles *for* tienen la siguiente sintaxis:

```
for      (ValorPrimeralteración      ;      CondiciónFinalización      ;  
         ModificaciónValor)  
         BloqueInstrucciones
```

Veamos los elementos en detalle:

- **for**: indicador de comienzo del bucle.
- **ValorPrimeralteración**: indica el valor inicial de la variable que se ejecutará para controlar el número de iteraciones del bucle.
- **CondiciónFinalización**: indica la condición que debe cumplirse para seguir repitiéndose el bucle. Lo normal es que la variable utilizada para controlar el número de iteraciones sea parte de la condición de finalización.
- **ModificaciónValor**: indica cómo va a modificarse el valor de la variable utilizada para las iteraciones en cada iteración del bucle. Lo más normal es que la modificación sea sumarle uno a dicha variable.
- **BloqueInstrucciones**: conjunto de instrucciones que se ejecutarán en cada iteración.

DO

El tipo de bucle *do* está recomendado en los mismos casos que el bucle *while* pero añadiendo que siempre se ejecutará, al menos una vez, el bloque de instrucciones. La gran diferencia respecto a *while* es que el bucle *do* realiza la comprobación de la condición al final de la iteración mientras que *while* lo hace al comienzo de la iteración, por ello que en los bucles *do* el bloque de instrucciones del bucle se

ejecutará al menos una vez, cosa que no se puede garantizar en los bucles *while*.

Los bucles *do* tienen la siguiente sintaxis:

do

BloqueInstrucciones

while (Condición)

Veamos los elementos en detalle:

- **do**: indicador de comienzo del bucle.
- **BloqueInstrucciones**: conjunto de instrucciones que se ejecutarán en cada iteración.
- **while**: indicador para realizar la comprobación de la condición para verificar que se tiene que realizar una nueva iteración.
- **Condición**: condición que debe de cumplirse para que siga repitiéndose la ejecución del bucle.

FASE 1: Bucle WHILE

La primera fase del objetivo consiste en aprender a utilizar bucles *while*.

El primer ejercicio de la fase consiste ejecutar un bucle *while* mostrando por pantalla la variable *i* que se utiliza en la evaluación de la condición, la variable de control. Presta atención a dónde se inicializa la variable *i* y cómo se incrementa su valor dentro del bucle para evitar bucles infinitos y que el bucle termine de forma correcta.

El código fuente es el siguiente:

```
let i = 1;
while(i<10)
{
    console.log(i);
    i++;
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_1
1
2
3
4
5
6
7
8
9
Alfre:Objetivo 9 alfre$ []
```

El segundo ejercicio de la fase consiste en ejecutar un bucle *while* hasta que el usuario introduzca un número mayor que 100. En el momento en el que el usuario introduzca un valor superior a 100 la variable que se evalúa en la condición (*fin*) cambiará su valor a *true* haciendo que el bucle finalice.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let valor = 0;
let fin = false;
while(!fin)
{
    valor = parseInt(prompt("Introduzca un número mayor que 100: "));
    if(valor > 100)
    {
        fin = true;
    }
    else
    {
        console.log("El valor introducido no es mayor que 100");
    }
}
console.log("Fin del programa");
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_2
Introduzca un número mayor que 100: 69
El valor introducido no es mayor que 100
Introduzca un número mayor que 100: 23
El valor introducido no es mayor que 100
Introduzca un número mayor que 100: 100
El valor introducido no es mayor que 100
Introduzca un número mayor que 100: 101
Fin del programa
Alfre:Objetivo 9 alfre$ []
```

FASE 2: Bucle FOR

La segunda fase del objetivo consiste en aprender a utilizar bucles *for*.

El primer y único ejercicio de esta fase consiste ejecutar un bucle *for* y mostrar por pantalla la variable *i* que se utiliza como condición en su ejecución. Es muy importante que prestes atención a cómo se inicializa la variable en el bucle, a cómo se define la condición de finalización y a cómo se define cómo va cambiando de valor la variable.

El código fuente es el siguiente:

```
for(let i = 1; i<10;i++)
{
    console.log(i);
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_3
1
2
3
4
5
6
7
8
9
Alfre:Objetivo 9 alfre$ □
```

FASE 3: Bucles DO

La tercera fase del objetivo consiste en aprender a utilizar bucles *do*.

El primer y único ejercicio de esta fase consiste en solicitar al usuario que introduzca un número mayor que 100 y repetir la petición hasta que lo introduzca. Presta atención a la condición de finalización, el bucle se repetirá siempre que el usuario introduzca un valor menor o igual a 100, es decir, finalizará en el momento en el que el usuario introduzca un valor superior a 100.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

let valor = 0;
do
{
    valor = parseInt(prompt("Introduzca un número mayor que 100: "));
    if(valor <= 100)
    {
        console.log("El valor introducido no es mayor que 100");
    }
} while (valor<=100)
console.log("Fin del programa");
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_4
Introduzca un número mayor que 100: 97
El valor introducido no es mayor que 100
Introduzca un número mayor que 100: 44
El valor introducido no es mayor que 100
Introduzca un número mayor que 100: 123
Fin del programa
Alfre:Objetivo 9 alfre$ []
```

FASE 4: Bucles anidados

La cuarta fase de este objetivo trata de aprender a usar bucles anidados, que consisten en la utilización de bucles como parte de los bloques de instrucciones de otros bucles. El bucle que se encuentra dentro de otro bucle se suele llamar bucle interno o interior, mientras que el bucle que contiene un bucle interior se llama bucle externo o exterior. Puedes tener tantos niveles de anidamiento como necesites, es decir, un bucle dentro de otro bucle, que a su vez esté dentro de otro bucle que está dentro de otro bucle, etc.

El primer ejercicio de la fase consiste en ejecutar dos bucles *for* que están anidados. Cada iteración mostrará la iteración en la que se encuentra cada bucle para que puedas observar el flujo del mismo (item1 indica la iteración del primer bucle e item2 la del segundo).

El código fuente es el siguiente:

```
for(let i = 0; i<3;i++)
{
    for(let j = 0; j<5;j++)
    {
        console.log("item1 = %s, item2 = %s",i,j);
    }
}
```

El flujo de ejecución será así:

- Iteración del primer bucle sobre el primer elemento de la primera lista (0).
 - Iteración de todos los elementos de la lista del segundo bucle (0 al 4).
- Iteración del primer bucle sobre el segundo elemento de la primera lista (1).
 - Iteración de todos los elementos de la lista del segundo bucle (0 al 4).

- Iteración del primer bucle sobre el tercer elemento de la primera lista (2).
 - Iteración de todos los elementos de la lista del segundo bucle (0 al 4).

El bucle interno se ejecuta tantas veces como iteraciones tenga el bucle externo. En este caso, el bucle interno se ejecutará completamente un total de tres veces, mientras que el externo únicamente una.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_5
item1 = 0, item2 = 0
item1 = 0, item2 = 1
item1 = 0, item2 = 2
item1 = 0, item2 = 3
item1 = 0, item2 = 4
item1 = 1, item2 = 0
item1 = 1, item2 = 1
item1 = 1, item2 = 2
item1 = 1, item2 = 3
item1 = 1, item2 = 4
item1 = 2, item2 = 0
item1 = 2, item2 = 1
item1 = 2, item2 = 2
item1 = 2, item2 = 3
item1 = 2, item2 = 4
Alfre:Objetivo 9 alfre$ []
```

El segundo ejercicio de la fase consiste en realizar el mismo bucle que en el ejercicio anterior, pero siendo el bucle exterior un bucle *while* en vez de *for*. El número de iteraciones del bucle *while* viene dado por *i*, variable inicializada antes de declarar el bucle, y que irá incrementando su valor dentro del bucle a medida que se ejecuta cada iteración del bucle.

El código fuente es el siguiente:

```
let i = 0;
while(i<3)
{
  for (let j = 0; j < 5; j++)
```

```

    {
      console.log("item1 = %s, item2 = %s", i, j);
    }
    i++;
}

```

Recuerda que la sentencia *i++* equivale a hacer *i=i+1*.

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 9 alfre$ node Ej9_6
item1 = 0, item2 = 0
item1 = 0, item2 = 1
item1 = 0, item2 = 2
item1 = 0, item2 = 3
item1 = 0, item2 = 4
item1 = 1, item2 = 0
item1 = 1, item2 = 1
item1 = 1, item2 = 2
item1 = 1, item2 = 3
item1 = 1, item2 = 4
item1 = 2, item2 = 0
item1 = 2, item2 = 1
item1 = 2, item2 = 2
item1 = 2, item2 = 3
item1 = 2, item2 = 4
Alfre:Objetivo 9 alfre$ []

```

El tercer ejercicio de la fase consiste en modificar el ejercicio anterior cambiando el bucle interno a bucle *while*. En este ejercicio tienes que tener en cuenta que para cada iteración del bucle exterior tienes que inicializar las variables de control del bucle interior. El número de iteraciones del bucle interno viene dado por *j*, variable inicializada antes de la declaración del bucle, y que irá incrementando su valor dentro del bucle a medida que se ejecuta cada iteración del bucle.

El código fuente es el siguiente:

```

let i = 0;
let j = 0;
while (i < 3)
{
  j = 0;
  while(j<5)

```

```
{  
    console.log("item1 = %s, item2 = %s", i, j);  
    j++;  
}  
i++;  
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 9 alfre$ node Ej9_7  
item1 = 0, item2 = 0  
item1 = 0, item2 = 1  
item1 = 0, item2 = 2  
item1 = 0, item2 = 3  
item1 = 0, item2 = 4  
item1 = 1, item2 = 0  
item1 = 1, item2 = 1  
item1 = 1, item2 = 2  
item1 = 1, item2 = 3  
item1 = 1, item2 = 4  
item1 = 2, item2 = 0  
item1 = 2, item2 = 1  
item1 = 2, item2 = 2  
item1 = 2, item2 = 3  
item1 = 2, item2 = 4  
Alfre:Objetivo 9 alfre$ []
```

Ahora eres capaz de...

En este noveno objetivo has adquirido los siguientes conocimientos:

- Utilización de bucles while.
- Utilización de bucles for.
- Utilización de bucles do.
- Utilización de bucles anidados.

PROYECTO INTERMEDIO 1

Ha llegado el momento de realizar un pequeño proyecto que incorpore todo lo que has aprendido hasta el momento.

El primer proyecto que realizarás consiste en el desarrollo de una pequeña calculadora que realice las operaciones básicas. En el proyecto utilizarás los siguientes conocimientos adquiridos:

- Escritura de texto por pantalla.
- Lectura de información introducida por los usuarios.
- Control de flujo mediante *switch*.
- Bucle *while*.
- Operador lógico NOT.
- Operadores matemáticos (+, -, * y /).

Código fuente y ejecución

El código fuente del proyecto es el siguiente:

```
const prompt = require('prompt-sync')();

console.log("*****\nCALCULADORA\n*****");
console.log("Menú:");
console.log("1.- Sumar");
console.log("2.- Restar");
console.log("3.- Multiplicación");
console.log("4.- División");
console.log("5.- Salir");

let fin = false;
let opcion = 0;
while(!fin)
{
    opcion = parseInt(prompt("Opción: "));
    switch(opcion)
    {
        case 1:
            let sumando1 = parseInt(prompt("Inserte primer sumando: "));
            let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
            console.log("Resultado de la suma: " + (sumando1 + sumando2));
            break;
        case 2:
            let minuendo = parseInt(prompt("Inserte minuendo: "));
            let sustraendo = parseInt(prompt("Inserte sustraendo: "));
            console.log("Resultado de la resta: " + (minuendo - sustraendo));
            break;
        case 3:
            let multiplicando = parseInt(prompt("Inserte multiplicando: "));
            let multiplicador = parseInt(prompt("Inserte multiplicador: "));
            console.log("Resultado de la multiplicación: " + (multiplicando * multiplicador));
            break;
        case 4:
            let dividendo = parseInt(prompt("Inserte dividendo: "));
            let divisor = parseInt(prompt("Inserte divisor: "));
            console.log("Resultado de la división: " + (dividendo / divisor));
            break;
        case 5:
            fin = true;
            break;
    }
}
```

```
console.log("Programa finalizado");
```

Vamos a verlos en detalle.

El programa está compuesto por un bucle *while* que se repite indefinidamente hasta que la variable de control *fin* toma el valor *true* y mediante la operación lógica *not* hace que la condición del bucle sea falsa.

Antes de empezar la ejecución del bucle se mostrará por pantalla el menú de opciones que el usuario podrá elegir. Se inicializa la variable de control *fin* y la variable *opcion*.

Dentro del bucle *while* está la lectura de la opción elegida por el usuario y la estructura *switch*, que dependiendo de la operación seleccionada por el usuario tomará un camino u otro.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Proyecto 1 alfre$ node proyecto1.js
*****
CALCULADORA
*****
Menú:
1.- Sumar
2.- Restar
3.- Multiplicación
4.- División
5.- Salir
Opción: 1
Inserte primer sumando: 6
Inserte segundo sumando: 8
Resultado de la suma: 14
Opción: 2
Inserte minuendo: 9
Inserte sustraendo: 3
Resultado de la resta: 6
Opción: 3
Inserte multiplicando: 5
Inserte multiplicador: 6
Resultado de la multiplicación: 30
Opción: 4
Inserte dividendo: 45
Inserte divisor: 5
Resultado de la division: 9
Opción: 5
Programa finalizado
Alfre:Proyecto 1 alfre$ []
```

OBJETIVO 10: FUNCIONES

El décimo objetivo del libro consiste en el aprendizaje y uso de funciones en el código fuente que escribes.

El objetivo está compuesto por dos fases. En la primera vas a aprender a utilizar funciones de forma simple y en la segunda vas a aprender a utilizar funciones anidadas.

Conceptos teóricos

En este apartado vamos a explicarte la teoría de todo lo que necesitas saber para aprender a utilizar funciones.

Funciones

Una función es un bloque de código fuente que contiene un conjunto de instrucciones y que puede ser utilizada desde el código fuente que escribes tantas veces como necesites.

Las funciones tienen la capacidad de:

- Recibir datos de entrada para su ejecución.
- Devolver datos como resultado de la ejecución.

Ambas capacidades son opcionales, es decir, puedes tener funciones que no reciben datos y que no devuelven nada, funciones que reciben datos y que no devuelven nada, funciones que no reciben datos y que devuelven datos y por último funciones que reciben datos y que devuelven datos.

La utilización de funciones es beneficiosa ya que aporta las siguientes características al código fuente:

- Simplificación del código.
- Mejor organización del código.
- Reutilización de código fuente.

Resumiendo, una función es un bloque de código fuente independiente, que puede recibir datos de entradas y que como resultado de su ejecución puede devolver datos.

La sintaxis de las funciones en JavaScript es la siguiente:

```
function NombreFuncion (parámetros)
{
    BloqueInstrucciones;
    return ValorRetorno;
}
```

Veamos los elementos en detalle:

- **function**: palabra utilizada para definir una función en JavaScript.
- **NombreFuncion**: nombre que tendrá la función. Te aconsejamos que utilices nombres de funciones descriptivos que representen lo que la función hace.
- **parámetros**: conjunto de elementos de entrada que tiene la función. Los parámetros son opcionales, es decir, puede haber 0 o más. En caso de ser más de uno los parámetros, irán separados por coma.
- **BloqueInstrucciones**: bloque de código que ejecuta la función.
- **return**: retorna datos al código fuente que utilizó la función. Es opcional, ya que el retorno de datos no es obligatorio en las funciones.
- **ValorRetorno**: datos que se retornan.

Hasta aquí te hemos explicado cómo se definen funciones en JavaScript. Para poder utilizar funciones en JavaScript desde el código fuente tienes que hacerlo de la siguiente manera:

```
Variable = NombreFuncion(parámetros);
```

Veamos los elementos en detalle:

- **Variable**: almacenará el valor que devuelve la función. Es opcional, ya que se suprime si la función no devuelve nada.

- **NombreFuncion**: nombre de la función que vamos a utilizar.
- **Parámetros**: parámetros de entrada que tiene la función y que se escriben separados por comas en caso de ser más de uno. Es opcional, por lo que, si la función no recibe parámetros, se suprimirá.

FASE 1: Uso de una función

La primera fase del objetivo consiste en el aprendizaje del uso de funciones, tanto la definición de éstas como el uso desde otros puntos del código fuente.

El primer ejercicio de la fase consiste en la definición de la función *Saludar* y posterior uso (también llamado invocación) desde el programa. La función *Saludar* no recibe ningún parámetro de entrada ni devuelve nada, únicamente escribe por pantalla.

El código fuente es el siguiente:

```
function Saludar()
{
    console.log("¡Hola Time of Software!");
}

Saludar();
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 10 alfre$ node Ej10_1
¡Hola Time of Software!
Alfre:Objetivo 10 alfre$ []
```

El segundo ejercicio de la fase consiste en la definición de la función *EsMayorQueCero*, que comprobará si el parámetro que recibe es mayor que cero o no y mostrará un mensaje por pantalla indicándolo. La invocación de la función se realiza pasándole el número introducido por el usuario como parámetro de la misma.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function EsMayorQueCero(valor)
{
    if (valor > 0)
```

```
        console.log("El valor introducido es mayor que cero");
    else
        console.log("El valor introducido no es mayor que cero");
}

let valorLeido = parseInt(prompt("Introduce un número: "));
EsMayorQueCero(valorLeido);
```

La ejecución del código fuente anterior tendrá las siguientes salidas dependiendo del número introducido:

Número menor que cero:

```
Alfre:Objetivo 10 alfre$ node Ej10_2
Introduce un número: -4
El valor introducido no es mayor que cero
Alfre:Objetivo 10 alfre$
```

Número mayor que cero:

```
Alfre:Objetivo 10 alfre$ node Ej10_2
Introduce un número: 7
El valor introducido es mayor que cero
Alfre:Objetivo 10 alfre$
```

El tercer ejercicio de la fase consiste en la definición de la función *Sumar*, que realizará la suma de los dos parámetros que recibe como entrada y devolverá el resultado de la suma. La invocación de la función se realiza pasándole los dos números introducidos por el usuario. El resultado de la suma se muestra por pantalla.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function Sumar(sumando1, sumando2)
{
    return sumando1 + sumando2;
}

let sum1 = parseInt(prompt("Introduzca el primer sumando: "));
let sum2 = parseInt(prompt("Introduzca el segundo sumando: "));
```

```
console.log("Resultado: %s",Sumar(sum1,sum2));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 10 alfre$ node Ej10_3
Introduzca el primer sumando: 8
Introduzca el segundo sumando: 5
Resultado: 13
Alfre:Objetivo 10 alfre$ []
```

FASE 2: Funciones anidadas

La segunda fase del objetivo consiste en el uso de funciones desde dentro de funciones, es decir, aprender a utilizar funciones anidadas. De esta forma, puedes simplificar el código y reutilizar funciones que ya hayas desarrollado en las nuevas funciones que desarrollas.

El primer y único ejercicio de la fase consiste en la definición de una función que realice la suma de dos números solicitados al usuario cuya solicitud se hace utilizando otra función.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function SolicitarNumero()
{
    let valor = parseInt(prompt("Introduzca un número: "));
    return valor;
}

function Sumar()
{
    let valor1 = SolicitarNumero();
    let valor2 = SolicitarNumero();
    console.log("Resultado suma: %s", (valor1 + valor2));
}

Sumar();
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 10 alfre$ node Ej10_4
Introduzca un número: 9
Introduzca un número: 6
Resultado suma: 15
Alfre:Objetivo 10 alfre$ []
```

Ahora eres capaz de...

En este décimo objetivo has adquirido los siguientes conocimientos:

- Utilización de funciones.
- Utilización de funciones anidadas.

PROYECTO INTERMEDIO 2

Ha llegado el momento de realizar el segundo proyecto del libro, un proyecto evolutivo del proyecto número uno en el que desarrollaste una calculadora.

El proyecto consiste en aplicar los conocimientos que has adquirido en el objetivo anterior, relativo a las funciones, para crear una versión más sencilla de leer, ordenada y reutilizable.

Código fuente y ejecución

El proyecto evolutivo consiste en la creación de las siguientes funciones:

- Función *Sumar*: se encargará de realizar todo el proceso de suma.
- Función *Restar*: se encargará de realizar todo el proceso de restar.
- Función *Multiplicar*: se encargará de realizar todo el proceso de multiplicar.
- Función *Dividir*: se encargará de realizar todo el proceso de dividir.
- Función *Calculadora*: se encargará de ejecutar el bucle y pedir la opción a ejecutar al usuario.

Una vez tengas las funciones creadas, tienes que crear el código fuente para mostrar las opciones y la sentencia de invocación de la función *Calculadora*. El código fuente quedaría así:

```
const prompt = require('prompt-sync')();

function Sumar()
{
    let sumando1 = parseInt(prompt("Inserte primer sumando: "));
    let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
    console.log("Resultado de la suma: " + (sumando1 + sumando2));
}

function Restar()
{
    let minuendo = parseInt(prompt("Inserte minuendo: "));
    let sustraendo = parseInt(prompt("Inserte sustraendo: "));
    console.log("Resultado de la resta: " + (minuendo - sustraendo));
}

function Multiplicar()
{
    let multiplicando = parseInt(prompt("Inserte multiplicando: "));
    let multiplicador = parseInt(prompt("Inserte multiplicador: "));
```

```

        console.log("Resultado de la multiplicación: " + (multiplicando * multiplicador));
    }

function Dividir()
{
    let dividendo = parseInt(prompt("Inserte dividendo: "));
    let divisor = parseInt(prompt("Inserte divisor: "));
    console.log("Resultado de la division: " + (dividendo / divisor));
}

function Calculadora()
{
    let fin = false;
    let opcion = 0;
    while (!fin)
    {
        opcion = parseInt(prompt("Opción: "));
        switch (opcion)
        {
            case 1:
                Sumar();
                break;
            case 2:
                Restar();
                break;
            case 3:
                Multiplicar();
                break;
            case 4:
                Dividir();
                break;
            case 5:
                fin = true;
                break;
        }
    }
}

console.log("*****\nCALCULADORA\n*****");
console.log("Menú:");
console.log("1.- Sumar");
console.log("2.- Restar");
console.log("3.- Multiplicación");
console.log("4.- División");
console.log("5.- Salir");
Calculadora();
console.log("Programa finalizado");

```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Proyecto 2 alfre$ node proyecto2.js
*****
CALCULADORA
*****
Menú:
1.- Sumar
2.- Restar
3.- Multiplicación
4.- División
5.- Salir
Opción: 1
Inserte primer sumando: 5
Inserte segundo sumando: 9
Resultado de la suma: 14
Opción: 2
Inserte minuendo: 6
Inserte sustraendo: 1
Resultado de la resta: 5
Opción: 3
Inserte multiplicando: 8
Inserte multiplicador: 7
Resultado de la multiplicación: 56
Opción: 4
Inserte dividendo: 77
Inserte divisor: 7
Resultado de la division: 11
Opción: 5
Programa finalizado
Alfre:Proyecto 2 alfre$ []
```

Ahora eres capaz de...

En este segundo proyecto has aprendido a:

- Organizar el código fuente mediante funciones.

OBJETIVO 11: ARRAYS

El décimo primer objetivo del libro consiste en el aprendizaje y uso de los arrays.

El objetivo está compuesto por tres fases. En la primera fase aprenderás a utilizar los arrays de forma básica, en la segunda aprenderás una serie de métodos propios que tienen y en la tercera aprenderás a iterarlos.

FASE 1: Arrays básico

En la primera fase del objetivo vas a aprender las nociones básicas de los arrays.

Pero ¿qué es un array?: simplemente es un conjunto de elementos del mismo tipo. Los arrays tienes que verlos como una lista de elementos.

En el primer ejercicio aprenderás a crear arrays. La creación puedes realizarla de dos formas:

- Utilizando corchetes con los elementos separados por comas.
- Utilizando la sentencia *new Array* con los elementos como parámetros separados por coma.

El código fuente del primer ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
let equipos = new Array("FCBarcelona", "Atlético de Madrid", "Real Madrid");
console.log(animales);
console.log(equipos);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_1
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
[ 'FCBarcelona', 'Atlético de Madrid', 'Real Madrid' ]
Alfre:Objetivo 11 alfre$ ]
```

En el segundo ejercicio de la fase aprenderás a acceder a los elementos del array. Para acceder a los elementos se utilizan los corchetes con el número del elemento al que se accede entre ellos.

Array[Elemento]

Ten en cuenta que el primer elemento del array es el 0. Por ejemplo, si quisieras acceder al primer elemento del array tendrías que utilizar `array[0]`, si quisieras acceder al tercero tendrías que utilizar `array[2]`, etc.

Los elementos de los arrays pueden ser leídos y modificados. Para leerlos simplemente tienes que utilizar la sentencia que hemos visto para acceder, y para modificarlos lo que tienes que hacer es asignarles un valor del mismo modo que haces con las variables.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log(animales);
console.log(animales[0]);
console.log(animales[1]);
console.log(animales[2]);
console.log(animales[3]);
animales[1] = "Caballo";
animales[3] = "Cangrejo";
console.log(animales);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_2
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Perro
Gato
Periquito
Conejo
[ 'Perro', 'Caballo', 'Periquito', 'Cangrejo' ]
Alfre:Objetivo 11 alfre$ []
```

FASE 2: Métodos propios

La segunda fase del objetivo consiste en aprender a usar una serie de métodos propios que tienen los arrays.

En el primer ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **length**: propiedad que devuelve el número de elementos que contiene el array. No es un método propiamente dicho.
- **sort**: método que realiza una ordenación de los elementos del array.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log(animales);
console.log("Número de elementos: ", animales.length);
animales = animales.sort();
console.log(animales);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_3
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Número de elementos: 4
[ 'Conejo', 'Gato', 'Periquito', 'Perro' ]
Alfre:Objetivo 11 alfre$ []
```

En el segundo ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **push**: método que añade un elemento al final del array.
- **pop**: método que elimina el último elemento del array. El elemento que es eliminado es devuelto como resultado de la ejecución.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log("Número de elementos: ", animales.length);
console.log(animales);
animales.push("Caballo");
console.log("Número de elementos: ", animales.length);
console.log(animales);
let eliminado = animales.pop();
console.log("Elemento eliminado: ", eliminado);
console.log("Número de elementos: ", animales.length);
console.log(animales);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_4
Número de elementos: 4
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Número de elementos: 5
[ 'Perro', 'Gato', 'Periquito', 'Conejo', 'Caballo' ]
Elemento eliminado: Caballo
Número de elementos: 4
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Alfre:Objetivo 11 alfre$
```

En el tercer ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **toString**: método que convierte el array en una cadena de texto de elementos separados por comas.
- **join**: método que convierte el array en una cadena de texto utilizando como separador el carácter que se pasa como parámetro.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log(animales);
console.log(animales.toString());
console.log(animales.join(" | "));
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_5
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Perro,Gato,Periquito,Conejo
Perro | Gato | Periquito | Conejo
Alfre:Objetivo 11 alfre$
```

En el cuarto ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **shift**: método que elimina el primer elemento del array. El elemento que es eliminado es devuelto como resultado de la ejecución.
- **unshift**: método que añade un elemento como primer elemento del array.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log(animales);
let eliminado = animales.shift();
console.log("Elemento eliminado: ", eliminado);
console.log(animales);
animales.unshift("Caballo");
console.log(animales);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_6
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
Elemento eliminado: Perro
[ 'Gato', 'Periquito', 'Conejo' ]
[ 'Caballo', 'Gato', 'Periquito', 'Conejo' ]
Alfre:Objetivo 11 alfre$
```

En el quinto ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **splice**: método que permite añadir y eliminar elementos al array. El método tiene los siguientes parámetros:

- Primer parámetro: indica la posición en la que se añadirán los elementos.
- Segundo parámetros: indica el número de elementos que se borrarán a partir de la posición indicada en el primer parámetro.
- Resto de parámetros: número de parámetros variable que serán los elementos que se insertarán en el array.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito"];
console.log(animales);
animales.splice(1, 0, "Caballo", "Pez");
console.log(animales);
animales.splice(2, 1, "Rana", "Tiburón");
console.log(animales);
animales.splice(1,3);
console.log(animales);
```

La primera sentencia *splice* añadirá los dos elementos a partir del índice 1 y sin borrar elementos. La segunda sentencia añadirá los dos elementos borrando un elemento (*Pez*). La tercera sentencia eliminará tres elementos (*Caballo*, *Rana* y *Periquito*).

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_7
[ 'Perro', 'Gato', 'Periquito' ]
[ 'Perro', 'Caballo', 'Pez', 'Gato', 'Periquito' ]
[ 'Perro', 'Caballo', 'Rana', 'Tiburón', 'Gato', 'Periquito' ]
[ 'Perro', 'Gato', 'Periquito' ]
Alfre:Objetivo 11 alfre$ []
```

En el sexto ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **concat**: método que devuelve la concatenación del array que ejecuta el método con los arrays o elementos que se pasan por parámetros.

El código fuente del ejercicio es el siguiente:

```
let animales1 = ["Perro", "Gato", "Periquito"];
let animales2 = new Array("Caballo", "Pez");
let animales3 = ["Rana", "Conejo"];
console.log(animales1);
console.log(animales2);
console.log(animales3);
let animalesUnion = animales1.concat(animales2);
console.log(animalesUnion);
let animalesUnion2 = animales1.concat(animales2, animales3);
console.log(animalesUnion2);
let animalesUnion3 = animales2.concat("Ballena");
console.log(animalesUnion3);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_8
[ 'Perro', 'Gato', 'Periquito' ]
[ 'Caballo', 'Pez' ]
[ 'Rana', 'Conejo' ]
[ 'Perro', 'Gato', 'Periquito', 'Caballo', 'Pez' ]
[ 'Perro', 'Gato', 'Periquito', 'Caballo', 'Pez', 'Rana', 'Conejo' ]
[ 'Caballo', 'Pez', 'Ballena' ]
Alfre:Objetivo 11 alfre$ []
```

En el séptimo ejercicio de la fase aprenderás a utilizar los siguientes métodos:

- **reverse**: método que invierte el orden de los elementos del array.
- **slice**: método que obtiene un array a partir del array original. El método recibe dos parámetros. El primero es obligatorio e indica la posición del primer elemento en el array origen. El segundo es opcional e indica la posición del elemento (no incluido) hasta el que se obtendrá el array. En caso de no especificar el segundo se obtendrán todos los elementos hasta el final del array.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Caballo", "Pez", "Rana", "Conejo"];
console.log(animales);
console.log(animales.reverse());
let subanimales = animales.slice(2,5);
console.log(subanimales);
let subanimales2 = animales.slice(4);
console.log(subanimales2);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_9
[ 'Perro', 'Gato', 'Periquito', 'Caballo', 'Pez', 'Rana', 'Conejo' ]
[ 'Conejo', 'Rana', 'Pez', 'Caballo', 'Periquito', 'Gato', 'Perro' ]
[ 'Pez', 'Caballo', 'Periquito' ]
[ 'Periquito', 'Gato', 'Perro' ]
Alfre:Objetivo 11 alfre$ []
```

FASE 3: Iterando arrays

La tercera fase del objetivo consiste en aprender a iterar el array accediendo a todos los elementos que lo componen.

El primer ejercicio de la fase consiste en recorrer el array utilizando un bucle *for*. La variable de control del bucle es la que se utilizará para acceder a los elementos del array, se inicializará con valor cero e irá aumentando de uno en uno hasta obtener el valor del número de elementos que componen el array. El código fuente es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Caballo", "Pez", "Rana", "Conejo"];
console.log(animales);
for(let i = 0; i < animales.length; i++)
{
    console.log(animales[i]);
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_10
[ 'Perro', 'Gato', 'Periquito', 'Caballo', 'Pez', 'Rana', 'Conejo' ]
Perro
Gato
Periquito
Caballo
Pez
Rana
Conejo
Alfre:Objetivo 11 alfre$ []
```

El segundo ejercicio de la fase consiste en aprender a recorrer los arrays utilizando dos variantes del bucle *for*.

La primera variante del bucle *for* es la siguiente:

```
for(VariableControl in Array)
{
    BloqueCódigo;
```

}

En el bucle, la variable de control irá tomando los diferentes valores de los índices de los elementos que lo componen. Utilizando la variable de control podrás acceder a los elementos.

La segunda variante del bucle *for* es la siguiente:

```
for(Variable of Array)
{
    Bloque Código;
}
```

En el bucle, la variable irá obteniendo los diferentes valores de los elementos que componen el array.

El código fuente del ejercicio es el siguiente:

```
let animales = ["Perro", "Gato", "Periquito", "Conejo"];
console.log(animales);
for(i in animales)
{
    console.log(i, ":", animales[i]);
}
for(animal of animales)
{
    console.log(animal);
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 11 alfre$ node Ej11_11
[ 'Perro', 'Gato', 'Periquito', 'Conejo' ]
0 : Perro
1 : Gato
2 : Periquito
3 : Conejo
Perro
Gato
Periquito
Conejo
Alfre:Objetivo 11 alfre$ 
```

Ahora eres capaz de...

En este décimo primer objetivo has adquirido los siguientes conocimientos:

- Utilización de arrays.
- Utilización de métodos propios de arrays.

OBJETIVO 12: PROGRAMACIÓN ORIENTADA A OBJETOS

El décimo segundo objetivo del libro consiste en el aprendizaje y uso de la programación orientada a objetos y cómo utilizarla en tus programas.

El objetivo está compuesto por tres fases. En la primera fase aprenderás a utilizar clases simples para que afiances de forma correcta los nuevos conceptos. En la segunda fase aprenderás a utilizar la composición de clases y en la tercera la herencia de clases.

Conceptos teóricos

En este apartado vamos a explicarte todos los conceptos teóricos que necesitas saber para aprender la programación orientada a objetos.

Cambio de paradigma

El mundo del desarrollo de software es un mundo en constante evolución y cambio. Allá por los años 60 se empezó a hablar de un nuevo paradigma de desarrollo, que era la programación orientada a objetos. El objetivo de la programación orientada a objetos no era otro que paliar las deficiencias existentes en la programación en ese momento, que eran las siguientes:

- **Distinta abstracción del mundo:** la programación en ese momento se centraba en comportamientos representados por verbos normalmente, mientras que la programación orientada a objetos se centra en seres, representados por sustantivos normalmente. Se pasa de utilizar funciones que representan verbos, a utilizar clases, que representan sustantivos.
- **Dificultad de modificación y actualización:** los datos suelen ser compartidos por los programas, por lo que cualquier ligera modificación de los datos podía provocar que otro programa dejara de funcionar de forma indirecta.
- **Dificultad de mantenimiento:** la corrección de errores que existía en ese momento era bastante costosa y difícil de realizar.
- **Dificultad de reutilización:** las funciones/rutinas suelen ser muy dependientes del contexto en el que se crearon y eso dificulta reaprovecharlas en nuevos programas.

La programación orientada a objetos, básicamente, apareció para aportar lo siguiente:

- Nueva abstracción del mundo centrándolo en seres y no en verbos mediante nuevos conceptos como clase y objeto que veremos en el siguiente apartado.
- Control de acceso a los datos mediante encapsulación de éstos en las clases.
- Nuevas funcionalidades de desarrollo para clases, como por ejemplo herencia y composición, que permiten simplificar el desarrollo.

Concepto de clase y objeto

Antes de empezar, vamos a hacer un símil de la programación orientada a objetos con el mundo real. Mira a tu alrededor, ¿qué ves?. La respuesta es: objetos. Estamos rodeados de objetos, como pueden ser coches, lámparas, teléfonos, mesas... El nuevo paradigma de programación orientada a objetos está basado en una abstracción del mundo real que nos va a permitir desarrollar programas de forma más cercana a cómo vemos el mundo, pensando en objetos que tenemos delante y acciones que podemos hacer con ellos.

Una clase es un tipo de dato cuyas variables se llaman objetos o instancias. Es decir, la clase es la definición del concepto del mundo real y los objetos o instancias son el propio “objeto” del mundo real. Piensa por un segundo en un coche, antes de ser fabricado un coche tiene que ser definido, tiene que tener una plantilla que especifique sus componentes y lo que puede hacer, pues esa plantilla es lo que se conoce como Clase. Una vez el coche es construido, ese coche sería un objeto o instancia de la clase Coche, que es quien define qué es un coche y qué se puede hacer con un coche.

Las clases están compuestas por dos elementos:

- **Atributos:** información que almacena la clase.
- **Métodos:** operaciones que pueden realizarse con la clase.

Piensa ahora en el coche de antes, la clase coche podría tener atributos tales como número de marchas, número de asientos, cilindrada... y podría realizar las operaciones tales como subir marcha, bajar marcha, acelerar, frenar, encender el intermitente... Un objeto es un modelo de coche concreto.

Encapsulación

La encapsulación en programación orientada a objetos es un mecanismo que permite proteger el acceso y uso no controlados de los atributos y métodos que componen la clase. La encapsulación de datos es el pilar básico de la programación orientada a objetos.

Los atributos y métodos que componen una clase pueden ser de dos tipos:

- **Públicos:** los datos son accesibles sin control.
- **Privados:** los datos son accesibles de forma controlada.

Para poder encapsular los atributos lo que se tiene que hacer es definirlos como privados y generar un método público en la clase para poder acceder a ellos. De esta forma, únicamente son accesibles de forma directa los atributos por la clase, mientras que el acceso a los mismos por elementos externos a la clase se deberá de hacer utilizando dichos métodos públicos.

Tal y como hemos comentado, la encapsulación no sólo puede realizarse sobre los atributos de la clase, también es posible realizarla sobre los métodos, es decir, aquellos métodos que

indiquemos que son privados no podrán ser utilizados por elementos externos a la clase.

Composición

La composición en programación orientada a objetos nos va a permitir utilizar otras clases como atributos de otras clases. La composición te va a permitir reutilizar código fuente.

Cuando hablamos de composición tienes que pensar que entre las dos clases existe una relación del tipo “tiene un”.

Herencia

La herencia consiste en la definición de una clase utilizando como base / plantilla una clase ya existente. La nueva clase derivada tendrá todas las características de la clase base y ampliará el concepto de ésta, es decir, tendrá todos los atributos y métodos de la clase base además de los propios que definas dentro de la misma. Por tanto, la herencia te va a permitir reutilizar código fuente.

Cuando hablamos de herencia tienes que pensar que entre las dos clases existe una relación del tipo “es un”.

Sintaxis en JavaScript

En JavaScript las clases se definen de la siguiente manera:

```
class NombreClase {  
    constructor(ParámetrosConstructor)  
    BloqueCódigoConstructor; }  
    método1(Parámetros1) { BloqueCódigo1; }  
    método2(Parámetros2) { BloqueCódigo2; }
```

}

Veámosla la definición en detalle:

- **class**: palabra reservada para definir clases en JavaScript.
- **NombreClase**: indica el nombre de la clase.
- **constructor**: palabra reservada para definir el método constructor de la clase. El constructor es el método que se ejecutará cada vez que crees un objeto de la clase.
- **ParámetrosConstructor**: conjunto de parámetros que tiene el constructor. Los parámetros son opcionales, puedes tener desde cero hasta los que necesites.
- **BloqueCódigoConstructor**: código fuente que se ejecuta como parte del constructor cuando creas un objeto de la clase.
- **métodoX**: indica el nombre del método.
- **ParámetrosX**: conjunto de parámetros que tiene el método. Los parámetros sonopcionales, puedes tener desde cero hasta los que necesites.
- **BloqueCódigoX**: código fuente que ejecuta el método.

La forma de crear objetos de la clase es la siguiente:

```
Variable = new NombreClase(ParámetrosConstructor);
```

Veámosla en detalle:

- **Variable**: variable que almacenará el objeto devuelto por parte de la creación de este.
- **new**: palabra reservada en JavaScript para crear objetos de una clase.
- **NombreClase**: nombre de la clase de la que se quiere crear el objeto.
- **ParámetrosConstructor**: conjunto de parámetros que tiene el constructor de la clase.

FASE 1: Clases simples

La primera fase del objetivo consiste en el aprendizaje y uso de las clases y objetos.

El primer ejercicio de la fase consiste en la creación de una clase que represente un punto, con su coordenada X y su coordenada Y. El código fuente creará un objeto estableciendo los valores del punto y mostrándolo por pantalla. El código fuente es el siguiente:

```
class Punto
{
    constructor(X, Y)
    {
        this.X = X;
        this.Y = Y;
    }
}

let punto = new Punto(3,3);
console.log(punto);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 12 alfre$ node Ej12_1
Punto { X: 3, Y: 3 }
Alfre:Objetivo 12 alfre$ []
```

El segundo ejercicio de la fase consiste en crear un método en la clase punto para que muestre la información que tiene almacenada la clase. El método se llamará *mostrarPunto* y no recibirá parámetros, lo único que hará será mostrar la información por pantalla. En el ejercicio se crearán dos objetos diferentes y se mostrarán por pantalla utilizando el nuevo método creado. El código fuente es el siguiente:

```
class Punto
{
    constructor(X, Y)
```

```

    {
        this.X = X;
        this.Y = Y;
    }
    mostrarPunto()
    {
        console.log("El punto es (%s, %s)", this.X, this.Y);
    }
}

let punto = new Punto(3,3);
punto.mostrarPunto();
let punto2 = new Punto(8,7);
punto2.mostrarPunto();

```

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 12 alfre$ node Ej12_2
El punto es (3, 3)
El punto es (8, 7)
Alfre:Objetivo 12 alfre$ []

```

El tercer ejercicio de la fase consiste en crear métodos que modifiquen y devuelvan los atributos de la clase para poder encapsularlos. Estos métodos son especiales en JavaScript y tienen una sintaxis concreta. Veámosla.

Los métodos de lectura de atributos (aquellos que devuelven el valor) tienen la siguiente sintaxis:

```

get Atributo()
{
    return this.NombreAtributo;
}

```

Veámosla en detalle:

- **get**: palabra que se utiliza para indicar que es un método que devuelve el valor de un atributo.
- **atributo**: nombre que se le da al método.

- **return**: palabra que se utiliza para devolver el valor en el método.
- **this.NombreAtributo**: atributo de la clase que se devolverá.

Los métodos de escritura de atributos (aquellos que modifican el valor) tienen la siguiente sintaxis:

```
set Atributo(valor)
{
    this.NombreAtributo = valor;
}
```

Veámosla en detalle:

- **set**: palabra que se utiliza para indicar que es un método que modifica el valor de un atributo.
- **atributo**: nombre que se le da al método.
- **valor**: nuevo valor que se le asignará al atributo.
- **this.NombreAtributo**: atributo de la clase que se modificará.

Nosotros te recomendamos que sigas el siguiente patrón para el nombre de los atributos y los método:

- Nombre de los atributos: precede siempre el nombre con un guión bajo, por ejemplo: _X, _Nombre, etc.
- Nombre de los métodos: nombre normal, por ejemplo: X, Nombre, etc.

El código fuente del ejercicio es el siguiente:

```
class Punto
{
    constructor(X, Y)
    {
        this._X = X;
```

```

        this._Y = Y;
    }
    mostrarPunto()
    {
        console.log("El punto es (%s, %s)", this.X, this.Y);
    }
    get X()
    {
        return this._X;
    }
    set X(nuevaX)
    {
        this._X = nuevaX;
    }
    get Y()
    {
        return this._Y;
    }
    set Y(nuevaY)
    {
        this._Y = nuevaY;
    }
}

let punto = new Punto(3,3);
punto.mostrarPunto();
console.log("Coordenada X: %s", punto.X);
console.log("Coordenada Y: %s", punto.Y);
console.log("--- Punto modificado ---");
punto.X = 7;
punto.Y = 5;
punto.mostrarPunto();
console.log("Coordenada X: %s", punto.X);
console.log("Coordenada Y: %s", punto.Y);

```

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 12 alfre$ node Ej12_3
El punto es (3, 3)
Coordenada X: 3
Coordenada Y: 3
--- Punto modificado ---
El punto es (7, 5)
Coordenada X: 7
Coordenada Y: 5
Alfre:Objetivo 12 alfre$ []

```

FASE 2: Composición

La segunda fase del objetivo consiste en el aprendizaje y uso de la composición de clases.

El ejercicio de la fase consiste en el aprendizaje de la utilización de clases como atributos de otras clases. Teniendo la clase que hemos estado utilizando en la fase anterior, definiremos una nueva clase llamada *Triangulo*, que contendrá tres atributos *Punto* y un método para mostrar el triángulo por pantalla utilizando el método de mostrar punto de la clase *Punto*. El ejercicio consiste en la creación de tres objetos de la clase *Punto* y un objeto de la clase *Triangulo* que recibirá los tres objetos de la clase *Punto* como parámetro del constructor de la clase. El objeto de la clase *Triángulo* se mostrará se mostrará utilizando el método *MostrarTriangulo*. Por último, modificaremos la información de los puntos para volver a mostrar la información por pantalla y comprobar que ha cambiado.

El código fuente es el siguiente:

```
class Punto
{
    constructor(X, Y)
    {
        this._X = X;
        this._Y = Y;
    }
    mostrarPunto()
    {
        console.log("(%s, %s)", this.X, this.Y);
    }
    get X()
    {
        return this._X;
    }
    set X(nuevaX)
    {
        this._X = nuevaX;
    }
    get Y()
    {
        return this._Y;
    }
    set Y(nuevaY)
    {
        this._Y = nuevaY;
    }
}
```

```
{  
    return this._Y;  
}  
set Y(nuevaY)  
{  
    this._Y = nuevaY;  
}  
}  
  
class Triangulo  
{  
    constructor(punto1, punto2, punto3)  
    {  
        this._punto1 = punto1;  
        this._punto2 = punto2;  
        this._punto3 = punto3;  
    }  
    get Punto1()  
    {  
        return this._punto1;  
    }  
    set Punto1(punto1)  
    {  
        this._punto1 = punto1;  
    }  
    get Punto2()  
    {  
        return this._punto2;  
    }  
    set Punto2(punto2)  
    {  
        this._punto2 = punto2;  
    }  
    get Punto3()  
    {  
        return this._punto3;  
    }  
    set Punto3(punto3)  
    {  
        this._punto3 = punto3;  
    }  
    MostrarTriangulo()  
    {  
        console.log("--- Triángulo ---")  
        this._punto1.mostrarPunto();  
        this._punto2.mostrarPunto();  
        this._punto3.mostrarPunto();  
        console.log("-----")  
    }  
}
```

```
}

let p1 = new Punto(3, 4);
let p2 = new Punto(6, 8);
let p3 = new Punto(9, 2);
let triangulo = new Triangulo(p1, p2, p3);
triangulo.MostrarTriangulo();
p1.X = 4;
p2.Y = 1;
p3.X = 5;
p3.Y = 5;
triangulo.Punto1 = p1;
triangulo.Punto2 = p2;
triangulo.Punto3 = p3;
triangulo.MostrarTriangulo();
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 12 alfre$ node Ej12_4
--- Triángulo ---
(3, 4)
(6, 8)
(9, 2)
-----
--- Triángulo ---
(4, 4)
(6, 1)
(5, 5)
-----
Alfre:Objetivo 12 alfre$ []
```

FASE 3: Herencia

La tercera fase del objetivo consiste en el aprendizaje y uso de la herencia en programación orientada a objetos.

El primer ejercicio de la fase consiste en la utilización de una clase que será heredada por otra. La clase que será heredada será una clase que llamaremos *Electrodomestico*, que contendrá una serie de atributos y métodos que pueden ser heredados por otros electrodomésticos más concretos, como puede ser la clase *Lavadora*. La clase *Lavadora* tendrá disponibles los atributos y métodos de la clase *Electrodomestico*. El ejercicio consiste en la creación de un objeto *Lavadora* y utilizando los métodos de ambas clases (*Lavadora* y *Electrodomestico*) llenar toda la información y mostrarla por pantalla.

El código fuente es el siguiente:

```
class Electrodomestico
{
    constructor()
    {
        this._Encendido = false;
        this._Tension = 220;
    }
    Encender() { this._Encendido = true; }
    Apagar() { this._Encendido = false; }
    EstaEncendido(){ return this._Encendido; }
    get Tension(){ return this._Tension; }
    set Tension(tension){ this._Tension = tension;}
}

class Lavadora extends Electrodomestico
{
    constructor()
    {
        super();
        this._RPM = 0;
        this._Kilos = 0;
    }
    get RPM(){ return this._RPM; }
```

```

set RPM(rpm){ this._RPM = rpm; }
get Kilos(){ return this._Kilos; }
set Kilos(kilos){ this._Kilos = kilos; }
Mostrar()
{
    console.log("--- Lavadora ---");
    console.log("RPM: ", this._RPM);
    console.log("Kilos: ", this._Kilos);
    console.log("Tensión: ", this.Tension);
    if(this.EstaEncendido())
        console.log("¡La lavadora está encendida!");
    else
        console.log("¡La lavadora está apagada!");
    console.log("-----");
}
}

let l = new Lavadora();
l.Mostrar();
l.Encender();
l.Kilos = 7;
l.RPM = 1200;
l.Tension = 125;
l.Mostrar();

```

La herencia en JavaScript se define añadiendo a la definición de la clase que va a heredar la siguiente información en la cabecera: “*extends NombreClaseDeLaQueHereda*”. Esto indica que la clase heredará de la clase especificada en *extends*.

Fíjate en el constructor de la clase *Lavadora*, verás que tiene una sentencia que es *super()*. La sentencia lo que hace es ejecutar el constructor de la clase de la que hereda.

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 12 alfre$ node Ej12_5
--- Lavadora ---
RPM: 0
Kilos: 0
Tensión: 220
¡La lavadora está apagada!
-----
--- Lavadora ---
RPM: 1200
Kilos: 7
Tensión: 125
¡La lavadora está encendida!
-----
Alfre:Objetivo 12 alfre$ []

```

El segundo ejercicio de la fase consiste en modificar un poco el ejercicio anterior. La modificación consiste en modificar los constructores de ambas clases para que reciban parámetros.

```

class Electrodomestico
{
    constructor(encendido, tension)
    {
        this._Encendido = encendido;
        this._Tension = tension;
    }
    Encender() { this._Encendido = true; }
    Apagar() { this._Encendido = false; }
    EstaEncendido(){ return this._Encendido; }
    get Tension(){ return this._Tension; }
    set Tension(tension){ this._Tension = tension;}
}

class Lavadora extends Electrodomestico
{
    constructor(rpm, kilos, encendido, tension)
    {
        super(encendido, tension);
        this._RPM = rpm;
        this._Kilos = kilos;
    }
    get RPM(){ return this._RPM; }
    set RPM(rpm){ this._RPM = rpm; }
    get Kilos(){ return this._Kilos; }
    set Kilos(kilos){ this._Kilos = kilos; }
    Mostrar()
    {

```

```

        console.log("--- Lavadora ---");
        console.log("RPM: ", this._RPM);
        console.log("Kilos: ", this._Kilos);
        console.log("Tensión: ", this.Tension);
        if(this.EstaEncendido())
            console.log("¡La lavadora está encendida!");
        else
            console.log("¡La lavadora está apagada!");
        console.log("-----");
    }
}

let l = new Lavadora(1200, 7, false, 125);
l.Mostrar();
l.Encender();
l.Kilos = 8;
l.RPM = 1500;
l.Tension = 220;
l.Mostrar();

```

Fíjate en el constructor de la clase *Lavadora*, verás cómo se le pueden añadir parámetros a la sentencia *super*.

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 12 alfre$ node Ej12_6
--- Lavadora ---
RPM: 1200
Kilos: 7
Tensión: 125
¡La lavadora está apagada!
-----
--- Lavadora ---
RPM: 1500
Kilos: 8
Tensión: 220
¡La lavadora está encendida!
-----
Alfre:Objetivo 12 alfre$ []

```

El tercer ejercicio de la fase consiste en extender el ejercicio anterior creando una clase nueva que herede también de la clase *Electrodomestico*. En este ejercicio crearás la clase *Microondas*, que será una clase completamente diferente a *Lavadora*, pero, heredando ambas de la clase *Electrodomestico*. El ejercicio consiste

en la creación de un objeto de ambas clases y rellenar su información para posteriormente mostrarla por pantalla.

El código fuente es el siguiente:

```
class Electrodomestico
{
    constructor(encendido, tension)
    {
        this._Encendido = encendido;
        this._Tension = tension;
    }
    Encender() { this._Encendido = true; }
    Apagar() { this._Encendido = false; }
    EstaEncendido(){ return this._Encendido; }
    get Tension(){ return this._Tension; }
    set Tension(tension){ this._Tension = tension;}
}

class Lavadora extends Electrodomestico
{
    constructor(rpm, kilos, encendido, tension)
    {
        super(encendido, tension);
        this._RPM = rpm;
        this._Kilos = kilos;
    }
    get RPM(){ return this._RPM; }
    set RPM(rpm){ this._RPM = rpm; }
    get Kilos(){ return this._Kilos; }
    set Kilos(kilos){ this._Kilos = kilos; }
    Mostrar()
    {
        console.log("--- Lavadora ---");
        console.log("RPM: ", this._RPM);
        console.log("Kilos: ", this._Kilos);
        console.log("Tensión: ", this.Tension);
        if(this.EstaEncendido())
            console.log("¡La lavadora está encendida!");
        else
            console.log("¡La lavadora está apagada!");
        console.log("-----");
    }
}

class Microondas extends Electrodomestico
```

```

{
    constructor(potenciamaxima, grill, encendido, tension)
    {
        super(encendido, tension);
        this._PotenciaMaxima = potenciamaxima;
        this._Grill = grill;
    }
    get PotenciaMaxima(){ return this._PotenciaMaxima; }
    set PotenciaMaxima(potenciamaxima){ this._PotenciaMaxima = potenciamaxima; }
    get Grill(){ return this._Grill; }
    set Grill(grill){ this._Grill = grill; }
    Mostrar()
    {
        console.log("--- Microondas ---");
        console.log("Potencia máxima: ", this._PotenciaMaxima);
        console.log("Tiene grill: ", this._Grill);
        console.log("Tensión: ", this.Tension);
        if(this.EstaEncendido())
            console.log("¡El microondas está encendido!");
        else
            console.log("¡El microondas está apagado!");
        console.log("-----");
    }
}

let l = new Lavadora(1200, 7, false, 125);
l.Mostrar();
let m = new Microondas(800, true, false, 220);
m.Mostrar();

```

La ejecución del código fuente anterior tendrá la siguiente salida:

```

Alfre:Objetivo 12 alfre$ node Ej12_7
---- Lavadora ---
RPM: 1200
Kilos: 7
Tensión: 125
¡La lavadora está apagada!
-----
---- Microondas ---
Potencia máxima: 800
Tiene grill: true
Tensión: 220
¡El microondas está apagado!
-----
Alfre:Objetivo 12 alfre$ []

```

Ahora eres capaz de...

En este décimo segundo objetivo has adquirido los siguientes conocimientos:

- Definición de clases.
- Utilización de objetos.
- Utilización de la composición de clases.
- Utilización de la herencia de clases.

OBJETIVO 13: CONTROL DE EXCEPCIONES

El decimo tercer objetivo del libro consiste en el aprendizaje y uso del control de excepciones y errores que pueden darse en los programas que escribes.

El objetivo está compuesto únicamente por una fase en la que aprenderás cómo se controlan excepciones en JavaScript.

Conceptos teóricos

En este apartado vamos a explicarte los conceptos teóricos que tienes que conocer para trabajar con excepciones.

Excepciones

Una excepción es un error que ocurre mientras se ejecuta el programa y que no ocurre frecuentemente.

Mediante programación podemos realizar las operaciones para controlar las excepciones, que no son otra cosa que guardar el estado en el que se encontraba el programa en el momento justo del error e interrumpir el programa para ejecutar un código fuente concreto. En muchos casos, dependiendo del error ocurrido, el control de la excepción implicará que el programa siga ejecutándose después de controlarla, aunque en muchos casos ésto no será posible.

El proceso de controlar excepciones es similar en todos los lenguajes de programación. En primer lugar, es necesario incluir el código fuente de ejecución normal dentro de un bloque con la sentencia *try*. Posteriormente, se crea un bloque de código dentro de una sentencia *catch* que es la que se ejecutará en caso de error. En el control de excepciones existe la posibilidad de crear un bloque de código que se ejecute siempre al final, independientemente de si ocurre error o no. Dicho bloque de código se escribe como parte de la sentencia *finally*.

El control de excepciones en JavaScript tiene un aspecto así:

```
try
    BloqueInstruccionesPrograma
catch
```

```
BloqueInstruccionesError:
finally
  BloqueCodigoFinally
```

Veamos en detalle cada elemento:

- **try**: indicador de comienzo del bloque de código fuente que se controlará.
- **BloqueInstruccionesPrograma**: conjunto de instrucciones que componen el programa.
- **catch**: indicador de comienzo de excepción controlada.
- **BloqueInstruccionesError**: conjunto de instrucciones que se ejecuta si se produce un error.
- **finally**: indicador de comienzo del bloque de código final. La sección es opcional.
- **BloqueCodigoFinally**: conjunto de instrucciones que se ejecutan al acabar independientemente de que se haya producido o no una excepción.

FASE 1: Controlando excepciones

La primera y única fase del objetivo consiste en el aprendizaje de qué es una excepción, cómo se producen y qué formas existen de controlarlas.

El primer ejercicio de la fase consiste en ejecutar un programa que lanza una excepción al intentar utilizar una función que no existe.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function Sumar()
{
    let sumando1 = parseInt(prompt("Inserte primer sumando: "));
    let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
    console.log("Resultado de la suma: " + (sumando1 + sumando2));
}

Smar();
```

Fíjate en el nombre de la función (*Sumar*), el programa está intentan utilizar una función que se llamar *Smar*.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 13 alfre$ node Ej13_1
/Users/alfre/Dropbox/Libro JavaScript/Ejercicios/Objetivo 13/Ej13_1.js:10
  Smar();
^

ReferenceError: Smar is not defined
  at Object.<anonymous> (/Users/alfre/Dropbox/Libro JavaScript/Ejercicios/Objetivo 13/Ej13_1.js:10:1)
  at Module._compile (internal/modules/cjs/loader.js:1063:30)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
  at Module.load (internal/modules/cjs/loader.js:928:32)
  at Function.Module._load (internal/modules/cjs/loader.js:769:14)
  at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
  at internal/main/run_main_module.js:17:47
Alfre:Objetivo 13 alfre$ []
```

El segundo ejercicio de la fase consiste en controlar el código fuente del programa anterior y capturar la excepción que lanza la utilización

de un método que no existe. El objetivo es que no se muestre el error de antes por pantalla y mostremos un error personalizado.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function Sumar()
{
    let sumando1 = parseInt(prompt("Inserte primer sumando: "));
    let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
    console.log("Resultado de la suma: " + (sumando1 + sumando2));
}

try
{
    Smar();
}
catch
{
    console.log("ERROR: Se ha producido un error")
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 13 alfre$ node Ej13_2
ERROR: Se ha producido un error
Alfre:Objetivo 13 alfre$ []
```

El tercer ejercicio de la fase consiste en incluir un bloque de código final e indicar que el programa ha terminado.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function Sumar()
{
    let sumando1 = parseInt(prompt("Inserte primer sumando: "));
    let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
    console.log("Resultado de la suma: " + (sumando1 + sumando2));
}
```

```
try
{
    Smar();
}
catch
{
    console.log("ERROR: Se ha producido un error")
}
finally
{
    console.log("Programa finalizado")
}
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 13 alfre$ node Ej13_3
ERROR: Se ha producido un error
Programa finalizado
Alfre:Objetivo 13 alfre$ []
```

El cuarto ejercicio de la fase consiste en aprender a mostrar el mensaje que describe el error que se produce. Para ello, al bloque *catch* hay que añadirle un parámetro en la definición que es una variable que contiene toda la información de la excepción. La descripción del error producido se encuentra en la propiedad *message* de dicha variable.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();

function Sumar()
{
    let sumando1 = parseInt(prompt("Inserte primer sumando: "));
    let sumando2 = parseInt(prompt("Inserte segundo sumando: "));
    console.log("Resultado de la suma: " + (sumando1 + sumando2));
}

try
{
    Smar();
}
catch (e)
```

```
{  
    console.log("ERROR: %s", e.message)  
}  
finally  
{  
    console.log("Programa finalizado")  
}
```

La variable que almacena la información de la descripción es `e`, puedes ponerle el nombre que quieras, es simplemente cambiar `e` por otro nombre.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 13 alfre$ node Ej13_4  
ERROR: Smar is not defined  
Programa finalizado  
Alfre:Objetivo 13 alfre$ []
```

El quinto ejercicio de la fase consiste en aprender a lanzar excepciones personalizadas desde el código fuente. Para ello tenemos la sentencia `throw` que tiene el siguiente formato:

throw CadenaTexto;

La sentencia provocará una excepción controlada dentro del código fuente que será controlada en el código que ejecuta dicha porción de código. En el ejercicio hemos añadido a la función `SolicitarNumero` diversos `throw` que se lanzarán dependiendo del número que introduzca el usuario. La utilización de la función `SolicitarNumero` se encuentra dentro de un bloque try – catch, por tanto, todos los `throw` que se ejecuten dentro de la misma serán controlados en el catch del programa principal.

El código fuente es el siguiente:

```
const prompt = require('prompt-sync')();  
  
function SolicitarNumero()
```

```

{
  let numero = parseInt(prompt("Inserte número entre 0 y 10: "));
  if(isNaN(numero)) throw "El valor introducido no es un número";
  if(numero > 10) throw "Número muy grande";
  if(numero < 0) throw "Número muy pequeño";
  return numero;
}

try
{
  console.log("Número introducido: %s", SolicitarNumero());
}
catch (e)
{
  console.log("ERROR: %s", e)
}
finally
{
  console.log("Programa finalizado")
}

```

La siguiente imagen muestra una ejecución introduciendo un valor que comprendido entre 0 y 10:

```

Alfre:Objetivo 13 alfre$ node Ej13_5
Inserte número entre 0 y 10: 7
Número introducido: 7
Programa finalizado
Alfre:Objetivo 13 alfre$ []

```

La siguiente imagen muestra una ejecución introduciendo un valor que no es un número:

```

Alfre:Objetivo 13 alfre$ node Ej13_5
Inserte número entre 0 y 10: texto
ERROR: El valor introducido no es un número
Programa finalizado
Alfre:Objetivo 13 alfre$ []

```

La siguiente imagen muestra una ejecución introduciendo un valor superior a 10:

```
Alfre:Objetivo 13 alfre$ node Ej13_5
Inserte número entre 0 y 10: 13
ERROR: Número muy grande
Programa finalizado
Alfre:Objetivo 13 alfre$ []
```

La siguiente imagen muestra una ejecución introduciendo un valor inferior a 0:

```
Alfre:Objetivo 13 alfre$ node Ej13_5
Inserte número entre 0 y 10: -7
ERROR: Número muy pequeño
Programa finalizado
Alfre:Objetivo 13 alfre$ []
```

Ahora eres capaz de...

En este décimo tercer objetivo has adquirido los siguientes conocimientos:

- Control de excepciones en el código fuente.

OBJETIVO 14: MANIPULACIÓN DE FICHEROS

En este décimo cuarto y último objetivo vas a aprender a manejar la lectura y la escritura de ficheros de texto. El manejo de ficheros de texto es algo muy importante ya que te va a permitir guardar la información que utilizas en tus programas en ficheros para poder usarlos en un futuro.

El objetivo está compuesto por dos fases. En la primera fase aprenderás a leer información almacenada en ficheros de texto y en la segunda fase, aprenderás a escribir información en ficheros de texto.

FASE 1: Lectura de ficheros de texto

La primera fase del objetivo consiste en aprender a utilizar todos los comandos necesarios para realizar lecturas de ficheros de texto.

En todos los ejercicios de la fase vas a utilizar un fichero de texto con contenido, por lo que deberás de crear un fichero y escribir algo dentro de él. La ruta del fichero puede ser la que prefieras, en el ejercicio hemos supuesto que el fichero se llama “*prueba.txt*” y que se encuentra en el mismo directorio que el programa.

El fichero “*prueba.txt*” que hemos utilizado en los ejercicios contiene la siguiente información:

```
Time of Software  
http://www.timeofsoftware.com  
Aprende JavaScript en un fin de semana  
Aprende C# en un fin de semana  
Aprende Python en un fin de semana  
Aprende Arduino en un fin de semana
```

Para poder utilizar ficheros en tus programas es necesario que importes el módulo *fs* en tu programa de la siguiente forma:

```
const fs = require('fs');
```

El módulo *fs* contiene todos los métodos necesarios para poder leer y escribir ficheros.

La lectura del fichero la haremos con el método *readFileSync* que tiene el siguiente formato:

```
datosLeidos = fs.readFileSync(RutaFichero, Codificación);
```

Veámoslo en detalle:

- **datosLeidos**: variable o constante que almacenará el texto leído con el método.
- **RutaFichero**: cadena de texto que especifica la ruta en la que se encuentra el fichero en el ordenador.
- **Codificación**: cadena de texto que especifica la codificación del fichero, en el libro utilizaremos únicamente *utf8*.

El primer ejercicio de la fase consiste en la lectura del fichero de texto y en mostrar su contenido por pantalla. El código fuente es el siguiente:

```
const fs = require('fs');

const datos = fs.readFileSync('fichero.txt', 'utf8');
console.log(datos);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 14 alfre$ node Ej14_1
Time of Software
http://www.timeofsoftware.com
Aprende JavaScript en un fin de semana
Aprende C# en un fin de semana
Aprende Python en un fin de semana
Aprende Arduino en un fin de semana
Alfre:Objetivo 14 alfre$ []
```

En caso de que el fichero no exista en la ruta especificada el programa mostrará el siguiente error:

```

Alfre:Objetivo 14 alfre$ node Ej14_1
internal/fs/utils.js:307
    throw err;
^

Error: ENOENT: no such file or directory, open 'fichero2.txt'
  at Object.openSync (fs.js:476:3)
  at Object.readFileSync (fs.js:377:35)
  at Object.<anonymous> (/Users/alfre/Dropbox/Libro JavaScript/Ejercicios/Objetivo 14/Ej14_1.js:3:18)
  at Module._compile (internal/modules/cjs/loader.js:1063:30)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
  at Module.load (internal/modules/cjs/loader.js:928:32)
  at Function.Module._load (internal/modules/cjs/loader.js:769:14)
  at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
  at internal/main/run_main_module.js:17:47 {
  errno: -2,
  syscall: 'open',
  code: 'ENOENT',
  path: 'fichero2.txt'
}
Alfre:Objetivo 14 alfre$ 

```

El segundo ejercicio consiste en aprender el uso del método *statSync(RutaFichero)*. El método nos va a permitir comprobar si existe un fichero antes de acceder a él evitando de esta forma que se produzcan excepciones por no existir. El método en caso de no encontrar el fichero provocará una excepción que controlaremos con un try catch. El código fuente es el siguiente:

```

const fs = require('fs');

try
{
  fs.statSync('fichero.txt')
  console.log("--- El archivo existe ---");
  const datos = fs.readFileSync('fichero.txt', 'utf8');
  console.log(datos);
}
catch(e)
{
  console.log("ERROR: %s",e.message);
}

```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 14 alfre$ node Ej14_2
--- El archivo existe ---
Time of Software
http://www.timeofsoftware.com
Aprende JavaScript en un fin de semana
Aprende C# en un fin de semana
Aprende Python en un fin de semana
Aprende Arduino en un fin de semana
Alfre:Objetivo 14 alfre$
```

En caso de que el fichero no exista en la ruta especificada el programa mostrará el siguiente error:

```
Alfre:Objetivo 14 alfre$ node Ej14_2
ERROR: ENOENT: no such file or directory, stat 'fichero2.txt'
Alfre:Objetivo 14 alfre$
```

FASE 2: Escritura en ficheros de texto

La segunda fase del objetivo consiste en aprender a utilizar todos los comandos necesarios para realizar escrituras de ficheros de texto.

El método para escribir en ficheros de texto es el siguiente:

```
writeFileSync(RutaFichero, CadenaTexto);
```

El método creará el fichero y escribirá la información pasada como parámetro. En caso de que el fichero exista lo que hará será sobrescribir la información que contiene por la cadena de texto que indiques en el método.

El primer ejercicio de la fase consiste en escribir en un fichero, leer a posteriori el fichero y mostrarlo por pantalla.

El código fuente es el siguiente:

```
var fs = require('fs');

let cadena = "En un lugar de la mancha\nnde cuyo nombre no quiero acordarme";

fs.writeFileSync("nuevo.txt", cadena);
console.log("El archivo fue creado correctamente");

const datos = fs.readFileSync('nuevo.txt', 'utf8');
console.log("Fichero leído:");
console.log(datos);
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 14 alfre$ node Ej14_3
El archivo fue creado correctamente
Fichero leído:
En un lugar de la mancha
de cuyo nombre no quiero acordarme
Alfre:Objetivo 14 alfre$ []
```

El segundo ejercicio de la fase consiste en aprender a utilizar el método anterior para que añada información al final del fichero que se indica. Para ello, al método `writeFileSync` hay que añadirle un nuevo parámetro al final:

```
writeFileSync(RutaFichero, CadenaTexto, Parámetros);
```

Para escribir al final del fichero hay que añadir como parámetro lo siguiente: `{flag: "a+"}`. De esta forma estaremos indicándole al método que escriba al final del fichero en caso de que el fichero exista en vez de reemplazar el contenido de este.

El código fuente del ejercicio es el siguiente:

```
var fs = require('fs');

fs.writeFileSync('nuevolinea.txt', "En un lugar de la mancha,\n", {flag: "a+"});
fs.writeFileSync('nuevolinea.txt', "de cuyo nombre no quiero acordarme,\n", {flag: "a+"});
fs.writeFileSync('nuevolinea.txt', "no ha mucho tiempo que vivía un hidalgo de los de lanza
en astillero\n", {flag: "a+"});
fs.writeFileSync('nuevolinea.txt', "adarga antigua, rocín flaco y galgo corredor... \n", {flag:
"a+"});

console.log("El archivo fue creado correctamente");

const datos = fs.readFileSync('nuevolinea.txt', 'utf8');
console.log("Fichero leído:");
console.log(datos);
```

La primera ejecución del código fuente anterior tendrá la siguiente salida:

```
Alfre:Objetivo 14 alfre$ node Ej14_4
El archivo fue creado correctamente
Fichero leído:
En un lugar de la mancha,
de cuyo nombre no quiero acordarme,
no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero
adarga antigua, rocín flaco y galgo corredor...

Alfre:Objetivo 14 alfre$ []
```

Las siguiente ejecuciones irán añadiendo información al fichero, la siguiente imagen muestra lo que se vería en la segunda ejecución del programa:

```
Alfre:Objetivo 14 alfre$ node Ej14_4
El archivo fue creado correctamente
Fichero leido:
En un lugar de la mancha,
de cuyo nombre no quiero acordarme,
no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero
adarga antigua, rocín flaco y galgo corredor...
En un lugar de la mancha,
de cuyo nombre no quiero acordarme,
no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero
adarga antigua, rocín flaco y galgo corredor...
Alfre:Objetivo 14 alfre$
```

Ahora eres capaz de...

En este décimo cuarto objetivo has aprendido a:

- Lectura de ficheros de texto.
- Escritura de información en ficheros de texto.

PROYECTO FINAL

Ha llegado el momento de realizar el proyecto final del libro, en el que vas a utilizar todos los conocimientos aprendidos.

Código fuente

En este apartado vamos a explicarte el código fuente del proyecto con cada uno de sus componentes y realizaremos una ejecución de éste para que veas cómo funciona.

El proyecto Agenda estará compuesto por las siguientes clases:

- **Persona**: contendrá toda la información referente a la persona.
- **Dirección**: contendrá toda la información referente a la dirección.
- **Teléfono**: contendrá toda la información referente a los teléfonos.
- **Contacto**: clase que mediante composición tendrá la información de la persona, dirección y teléfonos.
- **Agenda**: contendrá toda la información de todos los contactos.

Veamos las clases en detalle:

Clase Persona

La clase persona estará compuesta por los siguientes atributos:

- Nombre: contendrá la información referente al nombre de la persona.
- Apellidos: contendrá la información referente a los apellidos de la persona.
- FechaNacimiento: contendrá la información referente a la fecha de nacimiento de la persona.

La clase persona estará compuesta por los siguientes métodos:

- get Nombre: devolverá la información del atributo Nombre.
- get Apellidos: devolverá la información del atributo Apellidos.
- get FechaNacimiento: devolverá la información del atributo FechaNacimiento.
- set Nombre: modificará la información del atributo Nombre.
- set Apellidos: modificará la información del atributo Apellidos.
- set FechaNacimiento: modificará la información del atributo FechaNacimiento.

La clase persona la crearás en el fichero *persona.js* y el código fuente será el siguiente:

```
class Persona
{
  constructor()
  {
    this._Nombre = "";
    this._Apellidos = "";
    this._FechaNacimiento = "";
  }
  get Nombre() { return this._Nombre; }
  get Apellidos() { return this._Apellidos; }
  get FechaNacimiento() { return this._FechaNacimiento; }
  set Nombre(nombre) { this._Nombre = nombre; }
  set Apellidos(apellidos) { this._Apellidos = apellidos; }
  set FechaNacimiento(fechanacimiento) { this._FechaNacimiento = fechanacimiento; }
}

module.exports = Persona;
```

En el proyecto final vas a aprender a definir clases en ficheros e incluirlos en otros ficheros para poder utilizarlos. Fíjate en la última línea del fichero, mediante la sentencia “*module.exports = Persona;*” exportarás las definiciones para que puedas utilizarlas en otros programas o ficheros. La forma de incluir la definición lo veremos en el fichero *contacto.js*.

Clase Dirección

La clase dirección estará compuesta por los siguientes atributos:

- Calle: contendrá la información referente a la calle de la dirección.
- Piso: contendrá la información referente al piso de la dirección.
- Ciudad: contendrá la información referente a la ciudad de la dirección.
- CodigoPostal: contendrá la información referente al código postal de la dirección.

La clase dirección estará compuesta por los siguientes métodos:

- get Calle: devolverá la información del atributo Calle.
- get Piso: devolverá la información del atributo Piso.
- get Ciudad: devolverá la información del atributo Ciudad.
- get CodigoPostal: devolverá la información del atributo CodigoPostal.
- set Calle: modificará la información del atributo Calle.
- set Piso: modificará la información del atributo Piso.
- set Ciudad: modificará la información del atributo Ciudad.
- set CodigoPostal: modificará la información del atributo CodigoPostal.

La clase dirección la crearás en el fichero *direccion.js* y el código fuente será el siguiente:

```
class Direccion
{
    constructor()
    {
        this._Calle = "";
        this._Piso = "";
        this._Ciudad = "";
        this._CodigoPostal = "";
    }
}
```

```

get Calle() { return this._Calle; }
get Piso() { return this._Piso; }
get Ciudad() { return this._Ciudad; }
get CodigoPostal() { return this._CodigoPostal; }
set Calle(calle) { this._Calle = calle; }
set Piso(piso) { this._Piso = piso; }
set Ciudad(ciudad) { this._Ciudad = ciudad; }
set CodigoPostal(codigopostal) { this._CodigoPostal = codigopostal; }
}

module.exports = Direccion;

```

Clase Teléfono

La clase teléfono estará compuesta por los siguientes atributos:

- **TelefonoFijo:** contendrá la información referente al teléfono fijo.
- **TelefonoMovil:** contendrá la información referente al teléfono móvil.
- **TelefonoTrabajo:** contendrá la información referente al teléfono del trabajo.

La clase teléfono estará compuesta por los siguientes métodos:

- **get TelefonoFijo:** devolverá la información del atributo **TelefonoFijo**.
- **get TelefonoMovil:** devolverá la información del atributo **TelefonoMovil**.
- **get TelefonoTrabajo:** devolverá la información del atributo **TelefonoTrabajo**.
- **set TelefonoFijo:** modificará la información del atributo **TelefonoFijo**.
- **set TelefonoMovil:** modificará la información del atributo **TelefonoMovil**.
- **set TelefonoTrabajo:** modificará la información del atributo **TelefonoTrabajo**.

La clase teléfono la crearás en el fichero *telefono.js* y el código fuente será el siguiente:

```
class Telefono
{
    constructor()
    {
        this._TelefonoMovil = "";
        this._TelefonoFijo = "";
        this._TelefonoTrabajo = "";
    }
    get TelefonoMovil() { return this._TelefonoMovil; }
    get TelefonoFijo() { return this._TelefonoFijo; }
    get TelefonoTrabajo() { return this._TelefonoTrabajo; }
    set TelefonoMovil(telefonomovil) { this._TelefonoMovil = telefonomovil; }
    set TelefonoFijo(telefonofijo) { this._TelefonoFijo = telefonofijo; }
    set TelefonoTrabajo(telefonotrabajo) { this._TelefonoTrabajo = telefonotrabajo; }
}
module.exports = Telefono;
```

Clase Contacto

La clase contacto estará compuesta por los siguientes atributos:

- Persona: contendrá la información personal.
- Teléfono: contendrá la información de los teléfonos de la persona.
- Dirección: Contendrá la información de la dirección de la persona.
- Email: contendrá la información referente al email.

La clase contacto estará compuesta por los siguientes métodos:

- get Calle: devolverá la información del atributo Calle.
- get Piso: devolverá la información del atributo Piso.
- get Ciudad: devolverá la información del atributo Ciudad.
- get CódigoPostal: devolverá la información del atributo CódigoPostal.

- set Calle: modificará la información del atributo Calle.
- set Piso: modificará la información del atributo Piso.
- set Ciudad: modificará la información del atributo Ciudad.
- set CodigoPostal: modificará la información del atributo CodigoPostal.
- get Nombre: devolverá la información del atributo Nombre.
- get Apellidos: devolverá la información del atributo Apellidos.
- get FechaNacimiento: devolverá la información del atributo FechaNacimiento.
- set Nombre: modificará la información del atributo Nombre.
- set Apellidos: modificará la información del atributo Apellidos.
- set FechaNacimiento: modificará la información del atributo FechaNacimiento.
- get TelefonoFijo: devolverá la información del atributo TelefonoFijo.
- get TelefonoMovil: devolverá la información del atributo TelefonoMovil.
- get TelefonoTrabajo: devolverá la información del atributo TelefonoTrabajo.
- set TelefonoFijo: modificará la información del atributo TelefonoFijo.
- set TelefonoMovil: modificará la información del atributo TelefonoMovil.
- set TelefonoTrabajo: modificará la información del atributo TelefonoTrabajo.
- get Email: devolverá la información del atributo Email.
- set Email: modificará la información del atributo Email.
- MostrarContacto: mostrará la información completa del contacto.

La clase contacto es una clase que tiene un atributo de la clase Persona, otro de la clase Dirección y otro de la clase Teléfono. Para no declararlos como públicos y permitir que un contacto se vea

como un único ente a la hora de programar, se han definido todos los métodos que tienen esas clases y se redirige a sus métodos de forma interna en la clase contacto.

La clase contacto la crearás en el fichero *contacto.js*. Fíjate cómo se incluyen las definiciones de las clases anteriores para estar disponibles, es exactamente igual que la inclusión de los módulo de Node.js, pero indicando la ruta del fichero en el ordenador.

```
const Telefono = require("./telefono.js");
const Persona = require("./persona.js");
const Direccion = require("./direccion.js");

class Contacto
{
    constructor()
    {
        this._Persona = new Persona();
        this._Direccion = new Direccion();
        this._Telefono = new Telefono();
        this._Email = "";
    }

    set Email(email) { this._Email = email; }
    get Email() { return this._Email; }

    set Nombre(nombre) { this._Persona.Nombre = nombre; }
    set Apellidos(apellidos) { this._Persona.Apellidos = apellidos; }
        set FechaNacimiento(fechanacimiento) { this._Persona.FechaNacimiento = fechanacimiento; }
    get Nombre() { return this._Persona.Nombre; }
    get Apellidos() { return this._Persona.Apellidos; }
    get FechaNacimiento() { return this._Persona.FechaNacimiento; }

    set TelefonoMovil(movil) { this._Telefono.TelefonoMovil = movil; }
    set TelefonoFijo(fijo) { this._Telefono.TelefonoFijo = fijo; }
    set TelefonoTrabajo(trabajo) { this._Telefono.TelefonoTrabajo = trabajo; }
    get TelefonoMovil() { return this._Telefono.TelefonoMovil; }
    get TelefonoFijo() { return this._Telefono.TelefonoFijo; }
    get TelefonoTrabajo() { return this._Telefono.TelefonoTrabajo; }

    set Calle(calle) { this._Direccion.Calle = calle; }
    set Piso(piso) { this._Direccion.Piso = piso; }
    set Ciudad(ciudad) { this._Direccion.Ciudad = ciudad; }
    set CodigoPostal(codigopostal) { this._Direccion.CodigoPostal = codigopostal; }
```

```

get Calle() { return this._Direccion.Calle; }
get Piso() { return this._Direccion.Piso; }
get Ciudad() { return this._Direccion.Ciudad; }
get CodigoPostal() { return this._Direccion.CodigoPostal; }

MostrarContacto()
{
    console.log("----- Contacto -----");
    console.log("Nombre: " + this._Persona.Nombre);
    console.log("Apellidos: " + this._Persona.Apellidos);
    console.log("Fecha de nacimiento:" + this._Persona.FechaNacimiento);
    console.log("Teléfono móvil: " + this._Telefono.TelefonoMovil);
    console.log("Teléfono fijo: " + this._Telefono.TelefonoFijo);
    console.log("Teléfono trabajo: " + this._Telefono.TelefonoTrabajo);
    console.log("Calle: " + this._Direccion.Calle);
    console.log("Piso: " + this._Direccion.Piso);
    console.log("Ciudad: " + this._Direccion.Ciudad);
    console.log("Código postal: " + this._Direccion.CodigoPostal);
    console.log("Email: " + this._Email);
    console.log("-----");
}
}

module.exports = Contacto;

```

Clase Agenda

La clase agenda estará compuesta por los siguientes atributos:

- ListaContactos: contendrá la información de todos los contactos que están en la agenda.
- Path: contendrá la información de la ruta física del fichero en el ordenador en donde están almacenados los contactos.

La clase agenda estará compuesta por los siguientes métodos:

- CargarContactos: cargará en la aplicación la lista de contactos leyéndola desde el fichero.
- CrearNuevoContacto: almacenará en el atributo *ListaContactos* un nuevo contacto.

- GuardarContactos: grabará en el fichero la lista de contactos que tiene almacenada el atributo *ListaContactos*.
- MostrarAgenda: mostrará por pantalla el contenido del atributo *ListaContactos*.
- BuscarContactoPorNombre: realizará una búsqueda de un contacto en el atributo *ListaContactos* por su nombre y lo devolverá.
- BuscarContactoPorTelefono: realizará una búsqueda de un contacto en el atributo *ListaContactos* por su teléfono y lo devolverá.
- BorrarContactoPorNombre: borrará un contacto del atributo *ListaContacto* utilizando el nombre para buscarlo.
- BorrarContactoPorTelefono: borrará un contacto del atributo *ListaContacto* utilizando el teléfono para buscarlo.

El constructor de la clase recibe como parámetro la ruta en la que se encuentra el fichero que se utiliza para almacenar los contactos.

La clase Agenda la crearás en el fichero *agenda.js* y el código fuente será el siguiente:

```
const fs = require('fs');
const Contacto = require("./contacto.js");

class Agenda
{
  constructor(path)
  {
    this._Path = path;
    this._ListaContactos = new Array();
  }

  CargarContactos()
  {
    try
    {
      fs.statSync(this._Path)

      this._ListaContactos = new Array();
    }
  }
}
```

```

const datos = fs.readFileSync(this._Path, 'utf8');

let cadenas = datos.split("\n");

if(cadenas.length > 0)
{
    for (let cad of cadenas)
    {
        let datos = cad.split("#");
        if(datos.length == 11)
        {
            let contacto = new Contacto();
            contacto.Nombre = datos[0];
            contacto.Apellidos = datos[1];
            contacto.FechaNacimiento = datos[2];
            contacto.TelefonoMovil = datos[3];
            contacto.TelefonoFijo = datos[4];
            contacto.TelefonoTrabajo = datos[5];
            contacto.Calle = datos[6];
            contacto.Piso = datos[7];
            contacto.Ciudad = datos[8];
            contacto.CodigoPostal = datos[9];
            contacto.Email = datos[10];
            this._ListaContactos.push(contacto);
        }
    }
}

console.log("INFO: Se han cargado un total de %s contactos.", this._ListaContactos.length);
return true;
}
catch(ex)
{
    console.log(ex.Message);
    return false;
}
}

CrearNuevoContacto(contacto)
{
    this._ListaContactos.push(contacto);
}

GuardarContactos()
{
    try
    {

```

```

let cadena = "";
for(let contacto of this._ListaContactos)
{
    cadena = cadena.concat(contacto.Nombre + "#");
    cadena = cadena.concat(contacto.Apellidos + "#");
    cadena = cadena.concat(contacto.FechaNacimiento + "#");
    cadena = cadena.concat(contacto.TelefonoMovil + "#");
    cadena = cadena.concat(contacto.TelefonoFijo + "#");
    cadena = cadena.concat(contacto.TelefonoTrabajo + "#");
    cadena = cadena.concat(contacto.Calle + "#");
    cadena = cadena.concat(contacto.Piso + "#");
    cadena = cadena.concat(contacto.Ciudad + "#");
    cadena = cadena.concat(contacto.CodigoPostal + "#");
    cadena = cadena.concat(contacto.Email + "\n");
}

fs.writeFileSync(this._Path, cadena);
console.log("INFO: Contactos guardados correctamente.");
return true;
}
catch(ex)
{
    console.log("ERROR: " + ex.Message);
}
return false;
}

MostrarAgenda()
{
    console.log("##### AGENDA #####");
    console.log("Número de contactos: %s", this._ListaContactos.length);
    for(let i of this._ListaContactos)
    {
        i.MostrarContacto();
    }
    console.log("#####");
}

BuscarContactosPorNombre(nombre)
{
    let listaEncontrados = new Array();
    for(let contacto of this._ListaContactos)
    {
        if(contacto.Nombre==nombre)
            listaEncontrados.push(contacto);
    }
    return listaEncontrados;
}

```

```

}

BuscarContactosPorTelefono(telefono)
{
    let listaEncontrados = new Array();
    for (let contacto of this._ListaContactos)
    {
        if (contacto.TelefonoMovil == telefono
            || contacto.TelefonoFijo == telefono
            || contacto.TelefonoTrabajo == telefono)
            listaEncontrados.push(contacto);
    }
    return listaEncontrados;
}

BorrarContactosPorNombre(nombre)
{
    let listaFinal = new Array();
    for (let contacto of this._ListaContactos)
    {
        if (!(contacto.Nombre == nombre))
            listaFinal.push(contacto);
    }
    console.log("INFO: Se han borrado %s contactos.", (this._ListaContactos.length - listaFinal.length));
    this._ListaContactos = listaFinal;
}

BorrarContactosPorTelefono(telefono)
{
    let listaFinal = new Array();
    for (let contacto of this._ListaContactos)
    {
        if (!(contacto.TelefonoMovil == telefono
            || contacto.TelefonoFijo == telefono
            || contacto.TelefonoTrabajo == telefono))
            listaFinal.push(contacto);
    }
    console.log("INFO: Se han borrado %s contactos.", (this._ListaContactos.length - listaFinal.length));
    this._ListaContactos = listaFinal;
}

module.exports = Agenda;

```

Programa

El fichero contiene el programa principal del proyecto junto con una serie de funciones que son las siguientes:

- ObtenerOpcion: leerá la opción elegida del menú por parte del usuario.
- MostrarMenu: mostrará el menú de opciones por pantalla.
- BuscarContacto: realizará el proceso completo de búsqueda a excepción de la propia búsqueda en la lista, que la realizará la propia clase Agenda con uno de los métodos de búsqueda.
- ProcesoCrearContacto: realizará el proceso completo de creación de un contacto a excepción del propio almacenamiento en la agenda que lo realizará la propia clase Agenda con el método de crear.
- BorrarContacto: realizará el proceso de borrado de un contacto a excepción del propio borrado del contacto de la agenda que lo realizará la propia clase Agenda con el método de borrar.
- Main: función principal de la aplicación que contiene el flujo del programa.

El programa principal lo crearás en el fichero *programa.js* y el código fuente será el siguiente:

```
const prompt = require('prompt-sync')();
const Contacto = require("./contacto.js");
const Agenda = require("./agenda.js");

function ObtenerOpcion(texto)
{
    let leido = false;
    let valorLeido = 0;
    while(!leido)
    {
        try
        {
            valorLeido = parseInt(prompt(texto));
        }
        catch(error)
        {
            console.log(error);
        }
    }
    return valorLeido;
}

function MostrarMenu()
{
    console.log("-----");
    console.log("1. Crear Contacto");
    console.log("2. Borrar Contacto");
    console.log("3. Buscar Contacto");
    console.log("4. Salir");
    console.log("-----");
}
```

```

        leido = true;
    }
    catch
    {
        console.log("ERROR: Tienes que introducir un número.");
    }
}
return valorLeido;
}

function MostrarMenu()
{
    console.log("--- Menú ---");
    console.log("1.- Mostrar contactos");
    console.log("2.- Buscar contactos");
    console.log("3.- Crear contacto nuevo");
    console.log("4.- Borrar contactos");
    console.log("5.- Guardar contactos");
    console.log("6.- Salir");
}

function BuscarContactos()
{
    console.log("Buscar contactos: ");
    console.log("1.- Por nombre");
    console.log("2.- Por teléfono");
    console.log("3.- Volver");
    let finBuscar = false;
    while (!finBuscar)
    {
        let opcion = ObtenerOpcion("Opción: ");
        switch (opcion)
        {
            case 1:
                let nombre = prompt("Introduzca el nombre a buscar: ");
                let encontradosNombre = AgendaContactos.BuscarContactosPorNombre(nombre);
                if (encontradosNombre.length > 0)
                {
                    console.log("##### CONTACTOS ENCONTRADOS #####");
                    for (contacto of encontradosNombre)
                        contacto.MostrarContacto();
                    console.log("#####");
                }
                else
                    console.log("INFO: No se han encontrado contactos con ese nombre.");
                finBuscar = true;
                break;
        }
    }
}

```

```

case 2:
    let telefono = prompt("Introduzca el teléfono a buscar: ");
    let encontradosTelefono = =
AgendaContactos.BuscarContactosPorTelefono(telefono);
    if (encontradosTelefono.length > 0)
    {
        console.log("##### CONTACTOS ENCONTRADOS #####");
        for (contacto of encontradosTelefono)
            contacto.MostrarContacto();
        console.log("#####");
    }
    else
        console.log("INFO: No se han encontrado contactos con ese teléfono.");
    finBuscar = true;
    break;
case 3:
    finBuscar = true;
    break;
}
}

function ProcesoCrearContactos()
{
    let contacto = new Contacto();
    contacto.Nombre = prompt("Introduzce el nombre: ");
    contacto.Apellidos = prompt("Introduzce los apellidos: ");
    contacto.FechaNacimiento = prompt("Introduzce la fecha de nacimiento: ");
    contacto.TelefonoMovil = prompt("Introduzce el teléfono móvil: ");
    contacto.TelefonoFijo = prompt("Introduzce el teléfono fijo: ");
    contacto.TelefonoTrabajo = prompt("Introduzce el teléfono del trabajo: ");
    contacto.Calle = prompt("Introduzce la calle: ");
    contacto.Piso = prompt("Introduzce el piso: ");
    contacto.Ciudad = prompt("Introduzce la ciudad: ");
    contacto.CodigoPostal = prompt("Introduzce el código postal: ");
    contacto.Email = prompt("Introduzce el email: ");
    AgendaContactos.CrearNuevoContacto(contacto);
}

function BorrarContacto()
{
    console.log("Buscar contactos a borrar por: ");
    console.log("1.- Por nombre");
    console.log("2.- Por teléfono");
    console.log("3.- Volver");
    let finBuscar = false;
    while (!finBuscar)
    {

```

```

let opcion = ObtenerOpcion("Opción: ");
switch (opcion)
{
    case 1:
        let nombre = prompt("Introduzca el nombre: ");
        AgendaContactos.BorrarContactosPorNombre(nombre);
        finBuscar = true;
        break;
    case 2:
        let telefono = prompt("Introduzca el teléfono: ");
        AgendaContactos.BorrarContactosPorTelefono(telefono);
        finBuscar = true;
        break;
    case 3:
        finBuscar = true;
        break;
}
}
}

```

```

let AgendaContactos = new Agenda("Agenda.txt");
if (!AgendaContactos.CargarContactos())
    console.log("ERROR: No se pueden cargar los contactos del fichero.");
let fin = false;
while(!fin)
{
    MostrarMenu();
    switch(ObtenerOpcion("Opción: "))
    {
        case 1:
            AgendaContactos.MostrarAgenda();
            break;
        case 2:
            BuscarContactos();
            break;
        case 3:
            ProcesoCrearContactos();
            break;
        case 4:
            BorrarContacto();
            break;
        case 5:
            AgendaContactos.GuardarContactos();
            break;
        case 6:
            fin = true;
            break;
    }
}

```

```
}
```

```
console.log("FIN DEL PROGRAMA");
```

Ejecución

Antes de ejecutar el código fuente debes crear el fichero “*agenda.txt*” vacío (en los ficheros descargables lo tienes incluido también) y modificar la ruta en la que se encuentra el fichero en caso de ser necesario (línea 128 del fichero *programa.js*).

A continuación, te mostramos un ejemplo de crear un contacto y de mostrar los contactos que hay en la agenda, opciones 3 y 1 del menú:

```
Alfre:Proyecto Final alfre$ node programa.js
INFO: Se han cargado un total de 0 contactos.
---- Menú ---
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 3
Introduzce el nombre: Alfredo
Introduzce los apellidos: Moreno
Introduzce la fecha de nacimiento: 10/05/1984
Introduzce el teléfono móvil: 666555444
Introduzce el teléfono fijo: 666777888
Introduzce el teléfono del trabajo: 654321987
Introduzce la calle: Calle Falsa
Introduzce el piso: 3
Introduzce la ciudad: Madrid
Introduzce el código postal: 28000
Introduzce el email: info@timeofsoftware.com
---- Menú ---
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 1
##### AGENDA #####
Número de contactos: 1
----- Contacto -----
Nombre: Alfredo
Apellidos: Moreno
Fecha de nacimiento: 10/05/1984
Teléfono móvil: 666555444
Teléfono fijo: 666777888
Teléfono trabajo: 654321987
Calle: Calle Falsa
Piso: 3
Ciudad: Madrid
Código postal: 28000
Email: info@timeofsoftware.com
#####
#####
```

La siguiente captura muestra la opción de buscar contacto por nombre, opción 2 del menú:

```
---- Menú ----
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 2
Buscar contactos:
1.- Por nombre
2.- Por teléfono
3.- Volver
Opción: 1
Introduzca el nombre a buscar: Sheila
##### CONTACTOS ENCONTRADOS #####
----- Contacto -----
Nombre: Sheila
Apellidos: Córcoles
Fecha de nacimiento: 19/11/1981
Teléfono móvil: 666888999
Teléfono fijo: 666333222
Teléfono trabajo: 666123456
Calle: Calle Auténtica
Piso: 3
Ciudad: Madrid
Código postal: 28001
Email: info@timeofsoftware.com
#####
#
```

La siguiente captura muestra la opción de borrar contacto por teléfono, opción 4 del menú:

```
---- Menú ----
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 4
Buscar contactos a borrar por:
1.- Por nombre
2.- Por teléfono
3.- Volver
Opción: 2
Introduzca el teléfono: 666888999
INFO: Se han borrado 1 contactos.
---- Menú ----
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 1
##### AGENDA #####
Número de contactos: 1
----- Contacto -----
Nombre: Alfredo
Apellidos: Moreno
Fecha de nacimiento: 10/05/1984
Teléfono móvil: 666555444
Teléfono fijo: 666777888
Teléfono trabajo: 654321987
Calle: Calle Falsa
Piso: 3
Ciudad: Madrid
Código postal: 28000
Email: info@timeofsoftware.com
#####
#
```

La siguiente captura muestra la ejecución de la opción 5 del menú, guardar los contactos en el fichero:

```
---- Menú ----
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 5
INFO: Contactos guardados correctamente.
```

Por último, la siguiente imagen muestra la opción 6 del menú, salir del programa:

```
--- Menú ---
1.- Mostrar contactos
2.- Buscar contactos
3.- Crear contacto nuevo
4.- Borrar contactos
5.- Guardar contactos
6.- Salir
Opción: 6
FIN DEL PROGRAMA
```

Ahora eres capaz de...

En este proyecto final has aprendido a:

- Realizar un programa completo con todos los conocimientos adquiridos en el libro.
- Exportar e importar clases.

¡CONSEGUIDO!

A continuación te mostramos un resumen de todo lo que has aprendido en el libro:

- Definición de variables y constantes.
- Utilización del operador asignación.
- Mostrar información por pantalla.
- Utilización variables.
- Inclusión y utilización de paquetes de Node.js.
- Lectura de información introducida por el usuario.
- Utilización de números enteros.
- Utilización de operadores aritméticos.
- Conversión de cadenas de texto en números enteros.
- Utilización de paréntesis en operaciones aritméticas complejas.
- Utilización de operadores de asignación compuestos.
- Utilización de números decimales.
- Utilización de cadenas de texto.
- Utilización de métodos propios de las cadenas de texto.
- Utilización de tipos de datos fecha.
- Utilización de métodos propios de los tipos de datos fecha.
- Utilización de tipos de datos booleanos.
- Utilización de operadores relacionales.
- Utilización de operadores lógicos.
- Utilización de sentencias if y switch.
- Utilización de bucles while / for / do.
- Utilización de bucles anidados.
- Utilización de funciones.
- Utilización de funciones anidadas.
- Utilización de arrays.
- Utilización de métodos propios de arrays.
- Definición de clases.
- Utilización de objetos.
- Utilización de la composición de clases.
- Utilización de la herencia de clases.

- Control de excepciones en el código fuente.
- Lectura y escritura de ficheros de texto.
- Exportar e importar clases.

ANEXOS

En el apartado de Anexos vamos a explicarte conceptos teóricos que amplían los conocimientos adquiridos en todo el libro.

Comentarios de código

Un recurso utilizado en programación para aclarar el código fuente que se escribe es la utilización de comentarios. Los comentarios son una forma de añadir documentación a los propios ficheros de código fuente. Los comentarios son ignorados por el navegador, por lo que puedes introducir todos los comentarios que deseas o necesites.

Los comentarios se pueden añadir al código fuente de dos formas diferentes:

- **Comentarios de una única línea:** comentarios que ocupan únicamente una línea del fichero.

```
// Aquí va el comentario
```

- **Bloque de comentarios o comentario de varias líneas:** son comentarios que ocupan más de una línea del fichero. Son útiles cuando el comentario que se quiere poner es largo o simplemente por estética.

```
/*
Comentario de la línea 1
Comentario de la línea 2
Comentario de la línea 3
*/
```

SOBRE LOS AUTORES Y AGRADECIMIENTOS

Este libro y todo lo que rodea a Time of Software es el resultado de muchos años dedicados a la docencia en el ámbito tecnológico. Primero con grupos de Educación Secundaria Obligatoria y Bachillerato y posteriormente mediante la docencia a formadores.

El trabajo de creación del método de aprendizaje, sintetización y ordenación de toda la información teórica relacionada con JavaScript y la elaboración de las diferentes prácticas plasmadas en el libro son responsabilidad de las personas que componen **Time of Software**, Alfredo Moreno y Sheila Córcoles, apasionados por el mundo tecnológico y por la docencia.

Queremos agradecer a nuestras familias, amigos y compañeros de trabajo el apoyo incondicional y las aportaciones que han realizado al método de aprendizaje de JavaScript que hemos desarrollado, ¡gracias por ser nuestros conejillos de indias! Sin vosotros esto no hubiera sido posible.

Y por supuesto gracias a ti por adquirir “Aprende JavaScript en un fin de semana”. Esperamos que hayas conseguido el objetivo que te propusiste cuando compraste el libro. Habrás podido comprobar que esto es sólo el principio, que JavaScript es un mundo apasionante. No tengas dudas en ponerte en contacto con nosotros para contarnos qué tal te ha ido y cómo te va, ¡NO ESTÁS SOLO!

MATERIAL DESCARGABLE

El código fuente de todos los ejercicios realizados en el libro puedes descargarlo de la siguiente URL:

<http://timeofsoftware.com/aprendejavascript/>

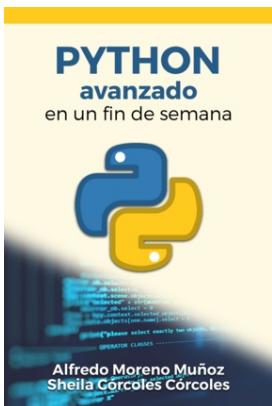
OTROS LIBROS DE LOS AUTORES

Si te ha gustado el libro y quieres seguir aprendiendo con el mismo formato, puedes encontrar en Amazon otros libros escritos por nosotros:

Aprende Python en un fin de semana



Python avanzado en un fin de semana



Aprende Arduino en un fin de semana



Aprende C# en un fin de semana



Aprende HTML y CSS en un fin de semana



Learn Python in a weekend

