**Bienvenido a Fundamentos de JavaScript 1**

Este curso es **el primero de una serie de dos cursos** que te prepararán para los examenes de certificación [JSE – Certified JavaScript Entry-Level Programmer](https://javascriptinstitute.com/jse-certification) y [JSA – Certified Associate JavaScript Programmer](https://javascriptinstitute.com/jsa-certification).

El objetivo principal de este curso es llevarte a un nivel de conocimiento de programación que te permita **diseñar**, **escribir**, **depurar**, y **ejecutar** programas codificados en el lenguaje de programación JavaScript, y comprender los **conceptos basicos de tecnologia en el desarrollo de software**.

El curso te preparará para trabajos y carreras relacionadas con el **desarrollo de software**, que incluye no solo la creación de código como desarrollador junior, sino también el diseño de **sistemas** de computadoras, y **pruebas de software**.

**Estructura del Curso**

El curso se divide en 6 módulos:

1. **Módulo 1**  
   Introducción a JavaScript y la Programación Informática.
2. **Módulo 2**  
   Variables, Tipos de Datos, Conversión de Tipos de Datos y Comentarios.
3. **Módulo 3**  
   Operadores e Interacción con el Usuario.
4. **Módulo 4**  
   Control de Flujo: Ejecución Condicional y Bucles.
5. **Módulo 5**  
   Funciones.
6. **Módulo 6**  
   Errores, Excepciones, Depuración y Resolución de Problemas.

**Temario del Curso**

En este curso aprenderás:

**Módulo 1**

**Introducción a JavaScript y la Programación Informática**

* Acerca de JavaScript (cómo comunicarse con la computadora, qué es JS, ventajas y limitaciones de JS, dónde se emplea JS hoy)
* Configuración del entorno de programación (herramientas de desarrollo, entorno de desarrollo en línea, entorno de desarrollo local)
* Primer programa JS - ¡Hola, mundo! (algunos comentarios sobre HTML, cómo ejecutar tu código JavaScript, ejecutar el código directamente en la consola)

**Módulo 2**

**Variables, Tipos de Datos, Conversión de Tipos de Datos y Comentarios**

* Variables (nombrado, declaración e inicialización de variables, el modo estricto, cambiar valores de variables, constantes y alcances)
* Tipos de datos primitivos (boolean, number, bigint, undefined, null, conversión de tipos de datos - funciones de construcción primitivas y conversiones primitivas, conversiones implícitas)
* Tipos de datos complejos (objetos, arreglos, propiedades y métodos básicos de los arreglos)
* Comentarios (comentarios de una sola línea, comentarios de varias líneas, documentación)

**Módulo 3**

**Operadores e Interacción con el Usuario**

* Operadores de asignación, aritméticos y lógicos (qué son los operadores, operadores de asignación, operadores aritméticos, operadores lógicos, operadores de asignación compuestos)
* Cadenas, comparación y otros operadores JS (concatenación de cadenas y asignaciones compuestas, operadores de comparación, operadores condicionales, operadores typeof, instanceof y delete, precedencia de operadores)
* Interacción con el usuario (cuadros de diálogo: alerta, confirmación, aviso)

**Módulo 4**

**Control de Flujo – Ejecución Condicional y Bucles**

* Ejecución condicional (qué es la ejecución condicional, la instrucción if-else, el operador condicional, la instrucción switch-case)
* Bucles (qué son los bucles, el bucle while, el bucle do-while, el bucle for, el bucle for-of, el bucle for-in, las sentencias break y continue)

**Módulo 5**

**Funciones**

* Conceptos básicos de funciones (qué son las funciones, funciones declaradas, funciones de llamada, variables locales, la sentencia return, los parámetros de funciones, sombreado)
* Las funciones como miembros de primera clase (expresiones de las funciones, pasar una función como parámetro, funciones callback
* Funciones arrow (declarar e invocar)
* Recursividad (concepto básica)

**Módulo 6**

**Errores, Excepciones, Depuración y Resolución de Problemas**

* Errores y excepciones: introducción (lenguajes naturales y errores de comunicación, errores frente a excepciones, errores sin excepciones, confianza limitada)
* Tipos básicos de errores en JS (SyntaxError, ReferenceError, TypeError, RangeError)
* Manejo de excepciones (la sentencia try-catch, la sentencia finally, la sentencia throw, y errores personalizados)
* Depuración de código y solución de problemas (qué es la depuración, ejecución paso a paso, visualización y modificación de variables, la opción de salida, medición del tiempo de ejecución del código)

**Objetivos**

**Módulo 1**

Después de completar el Módulo 1, el estudiante:

* Comprenderá los conceptos fundamentales de programación, tales como: la interpretación y el intérprete, la compilación y el compilador, la programación del lado del cliente frente al lado del servidor.
* Tendrá conocimientos básicos sobre cómo configurar y utilizar el entorno de programación básico (en línea o local).
* Adquirirá habilidades que le permitirá ejecutar su primer programa JavaScript en el lado del cliente (tanto como un elemento incrustado en la página HTML, así como directamente en la consola del navegador).

**Módulo 2**

Después de completar el Módulo 2, el estudiante:

* Tendrá el conocimiento y las habilidades para trabajar con variables (es decir, nombrar, declarar, inicializar y modificar sus valores)
* Comprenderá conceptos como el alcance, los bloques de código, el sombreado y el hoising.
* Conocerá las propiedades básicas de los tipos de datos primitivos, como boolean, number, bigint, undefined, null, y ser capaz de utilizarlos.
* Estará familiarizado con las propiedades básicas del tipo de dato primitivo string o cadena, incluidos los literales de cadena: comillas simples o dobles, el carácter de escape, la interpolación de cadenas, propiedades y métodos básicos.
* Conocerá las propiedades básicas de tipos de datos complejos como Array y Object (tratados como registros) y ser capaz de usarlos en la práctica.

**Módulo 3**

Después de completar el Módulo 3, el estudiante:

* Sabrá qué son los operadores y cómo clasificarlos (por tipo de operando, por número de operandos, etc.)
* Será capaz de utilizar operadores de asignación, aritméticos, lógicos y de comparación en la práctica.
* Comprenderá la operación del operador condicional y los operadores typeof, instanceof y delete.
* Comprenderá cuáles son la precedencia y la asociatividad de los operadores básicos y ser capaz de influir en ellos mediante la agrupación de paréntesis.
* Será capaz de realizar una comunicación bidireccional básica con el usuario del programa utilizando los cuadros de diálogo de alerta, confirmación y solicitud.

**Módulo 4**

Después de completar el Módulo 4, el estudiante:

* Será capaz de forzar la ejecución condicional de un grupo de sentencias (tomar decisiones y bifurcar el flujo) usando comandos if-else y switch.
* Será capaz de forzar la repetición de un grupo de sentencias en un bucle usando los comandos for, while y do-while, usando condiciones dependientes e independientes en el número de iteraciones.
* Entenderá y será capaz de usar instrucciones de interrupción y continuación específicas del bucle.
* Podrá usar la instrucción for-in para iterar sobre las propiedades de un objeto.
* Podrá usar la instrucción for-of para recorrer los elementos de un arreglo.

**Módulo 5**

Después de completar el Módulo 5, el estudiante:

* Podrá declarar y mandar llamar o invocar funciones.
* Sabrá cómo pasar argumentos a una función y devolver el resultado de su operación.
* Comprender el concepto de una variable local y el efecto de sombrear variables con los mismos nombres dentro de una función.
* Sabrá que una función en JS es un miembro de primera clase y podrá aprovechar esto declarando funciones usando expresiones de función y pasando funciones como argumentos a llamadas de otras funciones.
* Comprenderá el concepto de recursividad en el contexto de las funciones y será capaz de resolver problemas de programación simples usándola.
* Tendrá una comprensión básica de la función callback y será capaz de usarla de forma asíncrona junto con los métodos setTimeout y setInterval.
* Tendrá una comprensión clara de la notación de las funciones arrow y podrá escribir funciones alternativas como declaraciones regulares, expresiones de funciones y funciones arrow.

**Módulo 6**

Después de completar el Módulo 6, el estudiante:

* Obtendrá una comprensión de las diferencias entre errores sintácticos, semánticos y lógicos.
* Entenderá el concepto de una excepción y distinguirá entre las excepciones básicas generadas por JS cuando ocurre un error: SyntaxError, ReferenceError, TypeError, RangeError.
* Tendrá la capacidad de manejar excepciones utilizando la instrucción try-catch-final.
* Será capaz de generar sus propias excepciones usando la instrucción throw.
* Tendrá las habilidades para usar el depurador para el análisis básico de su propio código, que incluye: ejecución paso a paso, visualización y modificación de variables, y medición del tiempo de ejecución del código.

**El egresado del curso:**

* Conoce la sintaxis central del lenguaje JavaScript en un grado que le permite trabajar con variables, operadores, control de flujo y funciones.
* Conoce los conceptos básicos de los tipos de datos de JavaScript, distingue entre tipos primitivos y complejos, y es capaz de elegir el tipo adecuado a sus necesidades.
* Piensa algorítmicamente y puede analizar un problema utilizando el concepto programático.
* Puede elegir un tipo de dato adecuado al problema que se está resolviendo y usar medios de control de flujo adecuados.
* Puede diseñar, desarrollar y mejorar programas JavaScript muy simples.
* Puede interpretar y manejar excepciones básicas relacionadas con errores en la ejecución del programa.
* Comprende el trabajo de un programador en el proceso de desarrollo de software y el papel de las herramientas de desarrollo fundamentales.
* Sabe cómo se interpreta y ejecuta un programa en un entorno informático real, local o remoto.
* Puede crear y desarrollar su propio portafolio de programación.

**Obtén la certificación JSE**

Al finalizar el curso Fundamentos de JavaScript 1, estarás preparado para intentar obtener la certificación JSE – Certified JavaScript Entry-Level Programmer.

La certificación JSE – Certified JavaScript Entry-Level Programmer certification demuestra que estás familiarizado con los conceptos universales de programación como tipos de datos, contenedores, funciones, condiciones, y bucles, así como los aspectos más importantes del lenguaje de programación JavaScript, su síntaxis, semántica, y el entorno de ejecución.

Obtener la certificación JSE garantiza que estás **familiarizado con los medios más esenciales proporcionados por JavaScript**, que te permiten comenzar tus propios estudios en un nivel intermedio y continuar tu desarrollo profesional.  
  
La certificación JSE te permite demostrar que no solo estás al día con los asuntos relacionados con JavaScript, sino que también puedes manejarlos de manera competente.  
  
Además, con OpenEDG JavaScript Institute, obtienes acceso a una gran red de profesionales de la programación de JavaScript, un recurso valioso para resolver problemas relacionados con JavaScript y para desarrollar soluciones innovadoras.

La certificación *JSE – Certified Entry-Level JavaScript Programmer* es **el punto de partida** hacia la certificación *JSA: Certified Associate JavaScript Programmer* y **el comienzo** de una carrera en el desarrollo de software, programación JavaScript y tecnologías relacionadas.

Obtener la certificación JSE ayudará al titular del certificado a destacarse entre otros candidatos y dar el primer paso hacía una carrera de desarrollador profesional.

**¿Dónde se utiliza JavaScript?**

En general, JavaScript es una excelente opción para:

* Desarrollo de aplicaciones Web (por ejemplo, **Google Maps**)
* Aplicaciones Moviles (por ejemplo, **React Native** - framework para la creación de aplicaciones móviles)
* Juegos (motores de juego, por ejemplo, **PhysicsJS**, **Pixi.js**, **WebGL** (bibliotecas de gráficos web))
* Presentaciones (librerías: **RevealJs**, **BespokeJs**)
* Servidores Web y aplicaciones (por ejemplo, **Node.js**)  
  (Fuente: <https://www.javatpoint.com/>)

Y muchos, muchos otros proyectos y herramientas de desarrollo.

**Fundamentos de JavaScript 1 (JSE)  
Módulo 1**

**Introducción a JavaScript y la Programación Informática**

Después de completar el Módulo 1, el estudiante:

* Comprenderá los conceptos fundamentales de programación, tales como: la interpretación y el intérprete, la compilación y el compilador, la programación del lado del cliente y al lado del servidor.
* Tendrá los conocimientos básicos sobre cómo configurar y utilizar el entorno de programación básico (en línea o local).
* Adquirirá habilidades que le permitirán ejecutar su primer programa JavaScript en el lado del cliente (tanto como un elemento incrustado en la página HTML como directamente en la consola del navegador).

**Fundamentos de JavaScript 1 (JSE):**

**Módulo 1**

**Sección 1**

Acerca de JavaScript

Temas en la sección:

¿Cómo comunicarse con la computadora?

¿Qué es JS?

JS – Ventajas y Limitaciones

¿Dónde se usa JS hoy?

**Introducción**

¿Quieres aprender a programar en JavaScript? ¡Genial!

Este curso será un viaje bastante largo, pero si llegas hasta el final, podrás leer, comprender y, por supuesto, escribir aplicaciones y programas en JavaScript. Estas nuevas habilidades pueden ayudarte en tu trabajo actual o permitirte alcanzar nuevas oportunidades profesionales en un mercado de TI en constante crecimiento. Comencemos esta aventura sin más preámbulos y descubramos qué es JavaScript.

**Cómo comunicarse con la computadora**

Comencemos con algo obvio: las computadoras están en todas partes. Es casi seguro que estás leyendo este curso en una computadora; tu teléfono es una computadora. Encontrarás computadoras en televisores y otros electrodomésticos. Estamos rodeados de computadoras. Las computadoras se utilizan en la ciencia, la medicina, los bancos y los negocios. Sospechamos que hoy en día sería difícil encontrar algún ámbito de la vida en el que las computadoras no participen activamente.

Usamos computadoras principalmente porque son capaces de realizar ciertas tareas incomparablemente más rápido y con mayor precisión que las personas. Sin embargo, las computadoras no pueden adivinar qué tipo de tareas tenemos en mente o cómo ayudarnos a realizarlas. Tenemos que decirles eso. ¿Cómo? Es mejor hacerlo de forma similar a cuando se pasa información a otras personas, es decir, utilizando un lenguaje que sea comprensible para ambas partes. Usando dicho lenguaje, escribimos un programa, una solución formal a nuestro problema, que puede ser ejecutada por la computadora.

Desafortunadamente, un lenguaje que es directamente entendible por una computadora sería absolutamente ilegible para un ser humano normal. Sería una secuencia de instrucciones extrañas, escritas en forma numérica, refiriéndose a componentes informáticos que ni siquiera sabíamos que existían (y hablando francamente, no necesariamente tenemos que saberlo). Esta forma de comunicación, que data de los comienzos de la informática, hoy en día se usa muy raramente y solo en situaciones muy específicas.

Entonces, para ayudar en la comunicación con las computadoras, se inventaron los lenguajes de programación que son algo similares a los lenguajes naturales (es decir, los que se usan para la comunicación entre personas). Hay miles de lenguajes de programación, y difieren en propósito (aparte de los lenguajes de propósito general, hay muchos especializados, por ejemplo, lenguajes diseñados solo para cálculos estadísticos), nivel de abstracción (en términos simples: cuanto mayor sea el nivel del lenguaje , menos necesitamos saber sobre el hardware en el que se ejecuta el programa), facilidad de uso, efectividad de los programas escritos en ellos, etc.

**JavaScript como lenguaje interpretado**

JavaScript es un típico **lenguaje interpretado**. Si ejecutamos un código escrito en JavaScript en un navegador web, como está sucediendo en la página que estamos leyendo actualmente (sí, sí, hay elementos escritos en JavaScript en esta página también), el intérprete será el Motor JavaScript integrado en el navegador. Esta no es la única forma de ejecutar código JavaScript.

Tal vez hayas oído hablar de **node.js**. También es un intérprete, pero se instala de manera independiente a los navegadores, en un entorno en el sistema operativo de la computadora (puede ser macOS, Windows o Linux). El uso de node.js te permite escribir programas en JavaScript que, por ejemplo, convertirán tu computadora en un servidor.

Al comienzo de este párrafo, simplificamos un poco las cosas. JavaScript es un lenguaje interpretado, de eso no hay duda. Y, de hecho, ejecutar un programa escrito en JavaScript parece como si estuviéramos ejecutando nuestro código fuente (es decir, el código que escribimos) paso a paso. Sin embargo, puedes llegar a encontrar información sobre este lenguaje, y sobre intérpretes, que son un poco diferentes.

La mayoría de los motores de JavaScript modernos utilizan la técnica *Compilación Just In Time - Justo a Tiempo* (*Compilación JIT*). Esta técnica consiste en compilar fragmentos de código durante la ejecución del programa (más de una sola instrucción) y permite aumentar su rendimiento. Sin embargo, desde el punto de vista del usuario, dicho cambio es prácticamente imperceptible: sigue pareciendo que solo el intérprete está ejecutando el código fuente, instrucción por instrucción.

Independientemente del lenguaje que elijas, algunas cosas permanecen igual mientras escribes el programa. En primer lugar, una etapa importante, y probablemente la más difícil, de este proceso es definir correctamente el problema que queremos resolver. Solo entonces tratamos de encontrar la solución óptima, que finalmente presentaremos en forma de un programa escrito en el lenguaje elegido.

Entonces, antes de comenzar a explicarle algo a la computadora, en otras palabras, escribir un programa, debes comprender exactamente qué deseas lograr y cómo deseas lograrlo. En segundo lugar, la solución que proponemos y escribimos en forma de programa debe ser 100% precisa: la computadora no puede adivinar nada.

Un ejemplo simple de un campo ligeramente diferente: en algún momento de tu vida, probablemente compraste un mueble que requería ensamblaje. Montarlo es un problema con el que el comprador, ha tenido que cargar. Para que pueda hacer frente a esta tarea, obtiene un conjunto de instrucciones que lo guiarán a través de todo el proceso. Está actuando como intérprete en este punto, utilizando un programa que le permitirá completar la tarea. El éxito de tu misión depende de la calidad de estas instrucciones, de que sean precisas, y de que no provengan de otro mueble. Al final, puede resultar que no hayas construido los muebles de tus sueños, sino una construcción surrealista de otra dimensión.

Para que las instrucciones sean buenas, quien las desarrolle debe saber exactamente qué deben ilustrar, en qué orden deben realizar ciertas acciones, en qué etapas es más fácil confundirse, etc. Por supuesto, deben saber qué efecto se va a lograr al final.

**Algunas palabras más sobre JavaScript**

Como mencionamos antes, JavaScript es un lenguaje de programación interpretado. Como la mayoría de los lenguajes interpretados, también es un lenguaje de alto nivel (es decir, relativamente fácil de entender para las personas y que nos separa de los detalles del hardware).

A principios de los 90, todas las páginas web eran estáticas. Las cosas cambiaron en 1995 cuando la corporación Netscape contrató a Brendan Eich y le encargó que desarrollara un nuevo lenguaje para su producto, el navegador web Netscape Navigator. El nuevo lenguaje se llamó LiveScript, pero poco después se cambió su nombre a JavaScript. Su tarea principal era agregar dinámicas a los sitios web, lo que permitiría, por ejemplo, una interacción más compleja con el usuario. Y así comenzó la carrera de JavaScript.

**Programación del lado del cliente y del lado del servidor**

El uso de JavaScript en los sitios web, que con el tiempo se ha vuelto cada vez más complejo y, a menudo, contiene una lógica muy sofisticada, se denomina programación del **lado del cliente**. El código a ejecutar se carga junto con la página en el navegador, del lado del usuario, y el intérprete que forma parte del navegador web permite su ejecución.

Hoy en día, JavaScript es el único lenguaje compatible con todos los principales navegadores web, y aproximadamente el 95 % de las páginas web de todo el mundo incorporan código JavaScript en ellas. Desde el principio, las páginas web utilizaron JavaScript en el lado del cliente para agregar interactividad y cambiar dinámicamente el contenido.

Ahora es mucho más que eso, ya que JavaScript ofrece muchos frameworks sobre los cuales podemos construir aplicaciones web y redes sociales enormes y complejas (probablemente hayas escuchado los nombres de frameworks como **React** o **Angular**).

Todo esto puede funcionar en una variedad de equipos, desde estaciones de trabajo de alto rendimiento hasta simples teléfonos inteligentes. Gracias al poder de JavaScript, podemos pedir comida, jugar en el navegador, ver películas en plataformas de transmisión y estar en contacto constante con las personas importantes para nosotros. JavaScript es tan popular que continuamente se dedica más y más esfuerzo a usarlo, no solo como una solución del lado del cliente.

Con el tiempo, JavaScript comenzó a aparecer en otras áreas, como en la programación de lado del servidor de aplicaciones web complejas, también llamadas back-end. Estos programas se ejecutan en servidores, procesando datos (por ejemplo, de bases de datos), que después del procesamiento estarán disponibles en el lado del cliente. La flexibilidad de este lenguaje y su relativa sencillez lo han hecho mucho más aplicable, por ejemplo, en aplicaciones móviles, o incluso en la programación de UAVs - Vehículo Volador No Tripulado (algunos drones ejecutan programas escritos en este lenguaje).

**¿Es este el lenguaje de programación perfecto? - Desventajas**

Decimos que JavaScript es un lenguaje maduro, lo que significa que la mayoría de las funciones ya están implementadas y son estables, y probablemente no veremos grandes cambios en el lenguaje. Desde 2015, muchos aspectos de JavaScript han cambiado y se han agregado muchas características nuevas. Muchos de estos cambios se introdujeron para facilitar la migración a JavaScript para los programadores que conocen otros lenguajes populares, de los cuales JavaScript originalmente difería bastante en ciertos aspectos, como el manejo de objetos. Todavía podemos usar el lenguaje de la manera antigua, pero se recomienda usar el JavaScript moderno.

No existen soluciones ideales, por lo que no existen buenos lenguajes de programación para todas las aplicaciones. Cada uno de ellos tiene sus propias limitaciones, y JavaScript no es diferente. A pesar de su popularidad y éxito, JavaScript no es un lenguaje de programación perfecto. Debido a su naturaleza, no es adecuado para ciertas aplicaciones. Por ejemplo, no tiene sentido usarlo para escribir programas que requieran cálculos matemáticos avanzados o un rendimiento muy alto.

Algunas limitaciones se deben al propio concepto del lenguaje, pero la gran mayoría están relacionadas con la plataforma en la que lo usamos. Esto es especialmente visible cuando se escribe código para ser ejecutado en un navegador, que como dijimos anteriormente se denomina del lado del cliente. En tal situación, la funcionalidad de JavaScript está limitada por el hecho de que los navegadores, por razones de seguridad, ejecutan código de secuencia de comandos en un entorno *sandbox* (un entorno separado del mundo exterior), que no permite acceso a archivos y recursos locales (es decir, aquellos archivos que están en la computadora donde se inicia el navegador).

Otro inconveniente es que como el código no está compilado, entra en el navegador de la misma forma, o muy similar, a la que lo escribimos nosotros. ¿Por qué es esto una desventaja? Esto se debe a que todos pueden ver nuestra solución en una forma fácil de leer y usarla (ya sea en fragmentos o incluso en su totalidad) sin nuestro permiso.

Algo de ayuda aquí puede ser la ofuscación del código, que consiste en transformar nuestro script listo en una forma un poco menos legible (por ejemplo, generando nombres aleatorios cortos de variables y funciones, eliminando los signos de final de línea, etc.), pero el simple hecho es que si alguien quiere robar nuestro código JavaScript, es muy poco lo que podemos hacer para detenerlo.

**¿Es este el lenguaje de programación perfecto? - Ventajas**

Por otro lado, JavaScript tiene muchas ventajas sobre otros lenguajes de programación, y una de las más grandes es una comunidad muy activa y solidaria. Es fácil encontrar soluciones a problemas comunes y encontrar ayuda en general. Esto también significa que se desarrollan activamente herramientas que funcionan con JavaScript.

Otra gran ventaja es una gran cantidad de frameworks y librerías listas para usarse que brindan la mayoría de las funcionalidades y características comúnmente requeridas. El lenguaje en sí es relativamente fácil de aprender y nos permite centrarnos en el trabajo en lugar de luchar con la sintaxis (es decir, la forma de construir las instrucciones que componen el código de nuestro programa).

Además, JavaScript no requiere que compres herramientas costosas para trabajar con él, y las herramientas realmente buenas ya están integradas dentro de tu navegador web. Por último, pero no menos importante, los grandes jugadores como Google, Facebook y Mozilla apoyan activamente las herramientas de JavaScript y su desarrollo.

Sin embargo, lo que es una ventaja para unos puede resultar una desventaja para otros. Un ejemplo de esto es el *manejo dinámico de los tipos de datos* en JavaScript. Consiste en que podemos almacenar datos de cualquier tipo en una variable (una variable es un contenedor en el que almacenamos los datos que utilizaremos).

Por ejemplo, durante la ejecución del programa, podemos almacenar el número 10 en una variable, y en el siguiente paso usar la misma variable para almacenar la cadena "abc" (borrando el valor anterior automáticamente. No te preocupes si no entiendes en este momento, porque cubriremos todos estos términos más adelante).

Por lo general, esto es muy conveniente, pero varias personas han encontrado que esta característica del lenguaje es una desventaja. En su opinión, facilita que un programador cometa errores en ciertas situaciones. Al agregar *manejo estático de los tipos de datos*, donde una variable solo puede contener un tipo de dato (por ejemplo, números) durante la ejecución del programa, se creo un nuevo lenguaje llamado **TypeScript**.

Recuerda también que si aprendes a programar en un lenguaje, normalmente te será mucho más fácil aprender el siguiente, que por alguna razón puede ser mejor para resolver un problema en particular.

Pero comencemos ahora con JavaScript, que, debido a su sintaxis flexible y simple, es perfecto para aprender como primer lenguaje.

**Preparémonos para trabajar**

Como mencionamos anteriormente, JavaScript se puede usar en varios entornos, aunque la mayoría de las veces será un navegador web o un servidor con un entorno node.js. Cada entorno impone una manera un poco diferente de usar este lenguaje, y aparecen algunos mecanismos o funciones propias del mismo. Sin embargo, la parte esencial del lenguaje, su núcleo, sigue siendo el mismo. En esta parte del curso, aprenderemos a programar utilizando esta parte central e invariable de JavaScript: cómo declarar variables, escribir funciones, instrucciones condicionales o bucles; todo esto será igualmente utilizable en cualquier entorno en el que decidamos utilizar este lenguaje.

Programar en cualquier lenguaje no es algo fácil de aprender, y es posible que te sientas abrumado por tanta información nueva. Si eres persistente y te concentras, estarás escribiendo scripts simples en poco tiempo, y no hay otra forma de aprender a programar que escribir muchísimo código.

**Fundamentos de JavaScript 1 (JSE):  
Módulo 1**

**Sección 2**

**Configuración del entorno de programación**

Temas en esta sección:

* Herramientas de desarrollo
* Entorno de desarrollo en línea
* Entorno de desarrollo local (editor de código, intérprete, depurador)

Lo más importante es que no te rindas incluso cuando estés atascado: tómate un descanso, sal a caminar, vuelve a hacerlo con una mente fresca e inténtalo de nuevo. Al final, el ser constante te hace ganar la carrera.

¡Ahora, vamos a empezar!

**Herramientas de desarrollo**

Como cualquier otra tarea, la programación requiere las herramientas y el espacio de trabajo adecuados. El desarrollo de software, en la mayoría de los casos, requiere un **editor de código** y un **compilador** o **intérprete** de un lenguaje determinado. Este es el conjunto mínimo, que podemos ampliar según sea necesario con otras herramientas.

En esta etapa del curso, además del editor e intérprete de código JavaScript, también podemos utilizar el **depurador**, que es una herramienta que nos permite, entre otras cosas, pausar el programa en el lugar indicado y analizar su estado actual (por ejemplo, los valores de las variables indicadas).

Por supuesto, las herramientas en cuestión deberán ejecutarse en la computadora. En esta etapa, su rendimiento no es particularmente importante, y cualquier equipo que pueda manejar las tareas normales de oficina será suficiente, por lo que es muy recomendable trabajar desde una computadora de escritorio o portátil.

No se puede negar que el tamaño del monitor afectará la comodidad de tu trabajo. Cuanto más grande sea el monitor, más fácil será colocar el editor de código, el intérprete y otro contenido (por ejemplo, este curso) uno al lado del otro. En circunstancias normales de trabajo, los programadores suelen utilizar varios monitores.

No importa el sistema operativo, ya que se puede encontrar la herramienta adecuada para Windows, macOS y Linux.

En este momento, existen dos opciones. Puedes instalar todas las herramientas necesarias en tu máquina y trabajar en el entorno local. Este es el enfoque preferido, ya que así es como se ve en proyectos comerciales reales la mayor parte del tiempo. También puedes personalizar todo para satisfacer tus necesidades.

Todo el código que verás en este curso se probó en entornos locales y en línea, por lo que ambas opciones son válidas. Finalmente, podemos pasar a la elección de las herramientas.

**Entorno de desarrollo en línea**

**Los entornos en línea**, son sitios que actúan como un editor sencillo y un entorno de ejecución. Todos ellos tienen conjuntos similares de características. Tienen diferentes interfaces de usuario, pero en principio se comportan de manera similar. Te permiten escribir código, ejecutarlo con fines de prueba y, en la mayoría de los casos, compartirlo con otros usuarios.

En el caso de JavaScript, donde la preparación de un entorno local que funcione se reduce a instalar un editor de código y ejecutar el navegador, no son tan importantes como los entornos de desarrollo regulares. Se utilizan principalmente como plataformas de prueba y capacitación, o lugares para publicar soluciones de muestra a problemas de programación.

Entre los programadores de JavaScript, los más populares son los siguientes:

* [*JSFiddle*](https://jsfiddle.net/)
* [*CodePen*](https://codepen.io/pen/)
* [*JsBin*](https://jsbin.com/)
* [*Plunker*](https://plnkr.co/)

Durante el curso, utilizaremos un entorno en línea integrado con la plataforma de capacitación. OpenEDG proporciona un entorno simple para escribir y ejecutar código en varios lenguajes de programación, incluido JavaScript. Gracias a eso, podrás practicar todo lo que hablamos de inmediato.

Sin embargo, no olvides que esta plataforma es una solución puramente didáctica y de prueba, y ciertamente no puede usarse como un entorno de desarrollo completo. Sin embargo, es ideal para nuestras necesidades, ya que en la mayoría de los casos podremos olvidarnos del entorno web de los programas escritos en JavaScript, incluidos los elementos HTML. Esto nos permitirá centrarnos únicamente en aprender el lenguaje JavaScript en sí.

Sin embargo, se recomienda encarecidamente que también configures tu propio **entorno de desarrollo local**. No es difícil, como descubrirás de inmediato, y te permitirá realizar algunos ejercicios de una forma mucho más parecida a cómo lo harías durante el desarrollo normal de software. Si, durante el curso, alguno de los ejercicios debe realizarse en un entorno de este tipo, lo indicaremos claramente.

**Entorno de desarrollo local**

Como escribimos anteriormente, los requisitos de JavaScript para el entorno de desarrollo son muy modestos. En la mayoría de los casos, especialmente al comienzo del desarrollo, solo tres elementos son suficientes: un editor de código, un intérprete (es decir, un entorno de arranque) y un depurador.

Dependiendo del nivel de sofisticación, la complejidad del proyecto escrito o el entorno para el que escribimos nuestros programas (del lado del cliente, o del lado del servidor), es posible que también se necesiten otras herramientas.

Existirán, entre otras:

* **Administradores de paquetes**: que permiten la gestión de librerías (que contienen soluciones listas para usar, que podemos emplear en nuestros programas) o componentes del entorno de desarrollo (por ejemplo: npm o yarn).
* **Ejecutores de tareas y empaquetadores de módulos**: utilizados, en términos simples, para automatizar el proceso de desarrollo de software y unir el código resultante de muchos archivos y librerías (por ejemplo Grunt o Webpack).
* **Frameworks de pruebas**: permiten realizar pruebas automáticas para la corrección de nuestro programa, al buscar posibles errores (por ejemplo Mocha, Jasmine, o Jest).
* **Analizadores de seguridad**: como puede adivinar, se usan para controlar la seguridad de nuestra solución (por ejemplo, Snyk, RetireJS u OWASP Dependency Check).

La apertura de los entornos de desarrollo web es tanto una bendición como una maldición. Podemos elegir entre cientos de componentes, a partir de los cuales podemos crear el entorno más cómodo para nosotros.

Sin embargo, su cantidad, más los cambios dinámicos de herramientas particulares o incluso las tendencias entre los programadores hacen que sea difícil mantenerse al día con todo lo que sucede dentro de estos entornos.

Pero para nosotros, este es un problema para el futuro lejano.

Por ahora, necesitamos el trío mínimo: un **editor de código**, **intérprete** y **depurador**

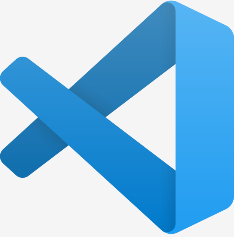
**Editor de Código**

El código de casi todos los lenguajes de programación se compone de alguna forma de texto. Entonces, para escribir el código, necesitamos un editor de texto. Pero debe ser una aplicación que escriba texto sin formato (no puede ser un editor de texto enriquecido, como MS Word). En otras palabras, solo un bloc de notas simple que pueda escribir archivos .txt es suficiente para escribir código, aunque es mucho más fácil si se usa un editor de código dedicado. El mercado está repleto de editores de código profesionales, tanto gratuitos como de paga. Algunos de ellos son universales, mientras que otros son exclusivos de lenguajes específicos. La principal ventaja de usar un editor de código dedicado es el resaltado de sintaxis, el autocompletado de texto y la verificación de errores. Esto mejora la eficiencia del trabajo y la comprensión del código, y reduce la cantidad de errores y errores tipográficos. Hay muchos buenos editores de código, pero puede ser realmente difícil seleccionar uno que funcione bien para ti.

Aquí se mencionan algunos populares:

* [***Visual Studio Code***](https://code.visualstudio.com/)

*[Windows, macOS, Linux]*

  
Potente editor de código gratuito para uso personal y comercial. Se ha convertido rápidamente en uno de los favoritos cuando se trata de desarrollo web. Tiene funciones integradas como un depurador de JavaScript y herramientas para optimizar los proyectos web. También es altamente personalizable a través del sistema de extensiones (existen muchas adiciones especialmente para el lenguaje JavaScript).

* [***WebStorm***](https://www.jetbrains.com/webstorm/)

*[Windows, macOS, Linux]*

  
Un entorno de desarrollo comercial popular, en el que el editor de código es solo uno de los elementos más pequeños en un gran conjunto de herramientas que mejoran el desarrollo de código (por ejemplo, soporta realizar pruebas al código). Destinado para proyectos grandes, puede resultar demasiado pesado y complejo para programas pequeños. Aunque está destinado a un uso comercial, es posible obtener una licencia educativa gratuita.

* [***Sublime Text***](https://www.sublimetext.com/)

[Windows, macOS, Linux]

  
Editor de código rápido y fácil de usar con muchas funciones avanzadas, como edición de varias líneas, búsqueda rápida y otras. Hay una versión de prueba disponible, pero para uso a largo plazo, se debe comprar una licencia ya sea para un uso privado y/o comercial.

* [***Notepad++***](https://notepad-plus-plus.org/)

[Windows]

Editor de código y texto gratuito y ligero. El programa es pequeño y rápido, admite docenas de lenguajes de programación y se puede ampliar con complementos (plugins). Puede ser viejo y feo, pero sigue siendo muy útil.

Existen muchos otros editores de código, tanto gratuitos como de paga, y puedes usar el que prefieras. Muchos desarrolladores usan, entre otras cosas, editores de consola, incluido el legendario vim. Los editores de consola no se ejecutan en un entorno gráfico, sino en una consola de texto. Sin embargo, solo puedes buscar tales soluciones si las tareas que vas a hacer resultan ser demasiado simples y quieres hacer tu vida un poco más difícil.

**Intérprete**

Ya hemos hablado un poco sobre el **intérprete** y su función. Funciona como un entorno de ejecución para nuestro programa. Comprueba si hemos cometido algún error formal, por ejemplo, cometer un error tipográfico en el nombre de una función u olvidar cerrar un paréntesis, y luego ejecuta el programa instrucción por instrucción.

La elección del intérprete de JavaScript dependerá de la plataforma para la que escribamos nuestro software. Por ejemplo, si queremos escribir una aplicación sencilla del lado del servidor, es casi seguro que elegiremos el entorno **node.js**, que tendremos que instalar directamente en nuestro sistema operativo. En el caso del software del lado del cliente, nuestro intérprete será simplemente el navegador web que ya tienes instalado (porque, ¿de qué otra forma estarías leyendo este curso?).

Nuestro curso trata sobre los fundamentos de JavaScript, es decir, aquellos elementos del lenguaje que serán igualmente útiles en soluciones móviles, del lado del cliente y del lado del servidor. Así podemos practicarlos en cualquier entorno, utilizando cualquier intérprete. La forma más fácil de hacer esto es limitarse a un navegador web.

Como hemos dicho anteriormente, prácticamente todos los navegadores tienen motores (o intérpretes) JavaScript integrados, pero recomendamos encarecidamente usar **Chrome** de Google, o **FireFox** de Mozilla . Ambos son conocidos por su eficiencia y herramientas avanzadas integradas para desarrolladores web (ese eres tú). Están disponibles para Windows, macOS y Linux.

**Depurador**

Los programas de computadora son bestias complicadas, miles o incluso millones de líneas de código (pero tranquilo, comenzaremos con solo unas pocas). Con tal complejidad y tamaño, es imposible producir código sin errores. Algunos tipos de errores, especialmente los lógicos (formalmente, el programa está escrito correctamente, pero probablemente inventamos la solución incorrecta al problema), solo se pueden encontrar mientras el programa se está ejecutando y, a menudo, solo en circunstancias especiales. Es realmente difícil averiguar qué sucede exactamente dentro de un programa que se ejecuta a la velocidad del rayo, y para esos problemas existen los depuradores.

Un **depurador** es una herramienta que te permite alentar o incluso detener la ejecución de un programa, ejecutar instrucciones paso a paso, ver y analizar el estado del programa en un momento dado.

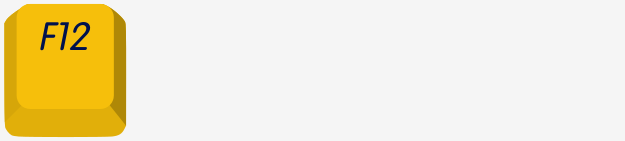
Afortunadamente, en el momento en que decidimos usar el navegador web como nuestro entorno de arranque e intérprete de JavaScript, también obtuvimos un depurador. Todos los navegadores modernos están equipados con las herramientas de desarrollo. Durante el funcionamiento normal, son invisibles y tenemos que habilitarlos en las opciones del navegador (más sobre esto en el próximo capítulo).

Dependiendo del navegador, allí encontraremos varias herramientas, pero seguramente habrá:

* **El inspector**, que nos permitirá, por ejemplo, analizar los elementos HTML individuales de un sitio web abierto.
* **La consola de JavaScript**, que en primer lugar muestra toda la información sobre los errores y, en segundo lugar, nos permite ejecutar comandos de JavaScript únicos en el contexto de la página actual.

* **El depurador**, que entre otras cosas, muestra los valores actuales de las variables y te permite pausar la ejecución del código en el lugar indicado y ejecutarlo paso a paso (es decir, ejecutar instrucciones del programa).

¿Cómo habilitas las herramientas para desarrolladores? Desafortunadamente, no hay una respuesta única; depende del navegador que estés utilizando (a veces también de su versión) y del sistema operativo. Las interfaces del navegador cambian con bastante frecuencia, por lo que es mejor aprender los atajos correctos en lugar de buscar la opción correcta en el menú. Prueba las siguientes combinaciones de teclas:

* Sistemas operativos Windows y Linux, todos los navegadores comunes excepto Internet Explorer y Edge:  
  
* Sistema operativo Windows, Internet Explorer y Edge:  
  
* Sistema operativo macOS, todos los navegadores comunes:  
  

En el próximo capítulo, volveremos a este tema y aprenderemos algunas cosas más sobre estas útiles herramientas.

En el próximo capítulo escribiremos nuestra primera pieza de código JavaScript. Lo probaremos en primer lugar en el entorno de ejecución integrado con nuestra plataforma de formación. También lo utilizaremos para comprobar cómo funciona nuestro entorno de desarrollo local. Por lo tanto, hay que asegúrarse de que las herramientas seleccionadas estén instaladas y de que puedas iniciarlas. Si aún no sabes qué elegir, te sugerimos usar el entorno local con **Visual Studio Code** y el **Chrome** (navegador web con intérprete de JavaScript y depurador).

**Fundamentos de JavaScript 1 (JSE):  
Módulo 1**

**Sección 3**

**¡Hola, Mundo!**

Temas en esta sección:

* Algunas palabras sobre HTML
* ¿Cómo puedes ejecutar tu código JavaScript?
* Ejecutando el código directamente en la consola

**El Programa "¡Hola, Mundo!"**

¿Por qué "¡Hola, Mundo!"? Durante casi 50 años, esta frase y sus derivados han marcado a alguien aprendiendo un nuevo lenguaje de programación, aunque es más una tradición que otra cosa. La frase se usó hace mucho tiempo en un libro muy importante sobre el lenguaje C, pero el texto en sí no es relevante ahora.

La idea es escribir algo en la pantalla utilizando un lenguaje específico. Primero, nos permite ver la sintaxis básica del lenguaje y compararlo con otros lenguajes de programación. En segundo lugar, es un programa muy simple y cualquiera puede escribirlo o copiarlo fácilmente de Internet y verificar si sus herramientas y su entorno están configurados correctamente. En tercer lugar, es un programa que genera algo, por lo que proporciona información sobre si se ejecutó correctamente o no.

En el caso de JavaScript del lado del cliente, mostrar algo en la pantalla se puede entender de dos maneras.

Primero, el JavaScript del lado del cliente siempre se ejecuta en el contexto de un sitio web y te permite manipular elementos de ese sitio web. Entonces podemos, por ejemplo, usar la función apropiada para insertar algún texto, cambiar un título, crear una tabla, etc. en la página. De esta manera, controlamos la parte visual del sitio web.

Segundo, podemos usar la consola como una pantalla para escribir alguna información. La consola, como mencionamos en el capítulo anterior, es parte de las herramientas del desarrollador. Por lo tanto, no está visible de forma predeterminada y debe estar habilitada correctamente (también escribimos sobre esto en el capítulo anterior). Para nuestras necesidades, será mucho más cómodo utilizar la consola, ya que evitaremos la necesidad de un análisis exhaustivo de la estructura del sitio web.

Pero, ¿qué es realmente una consola? En primer lugar, es un lugar donde se muestran varios mensajes, normalmente invisibles para el usuario del navegador. Estos mensajes pueden, por ejemplo, ser generados por el intérprete de JavaScript tras encontrar un error o si lo imprimimos, llamando a la función adecuada. En segundo lugar, podemos ejecutar comandos de JavaScript individuales en la consola, que se ejecutarán en el contexto de la página web actualmente cargada (se hablará un poco más sobre eso en un momento).

La función básica que nos permite escribir información en la consola es llamada console.log. Entonces, para referirnos al eterno "¡Hola, Mundo!", deberíamos mandarlo llamar de la siguiente manera:

**console.log("¡Hola, Mundo!");**

Podemos tratar a console.log como una función\*. De hecho, la función es solo un registro y la consola es el objeto al que pertenece la función.

\*Este tipo de función, perteneciente a un objeto, generalmente se denomina método. Pero una vez más, por el momento, para simplificar ciertas cosas, supongamos que se trata de una función ordinaria: no nos molestará en absoluto (aprenderemos sobre los objetos mucho más adelante).

Una función es un fragmento de código que te permite realizar una tarea específica (en nuestro caso, mostrar algo en la consola). Las funciones a menudo toman argumentos, en otras palabras, datos que usarán durante la operación. En JavaScript, ejecutamos una función llamándola, y la llamamos escribiendo su nombre seguido de un par de paréntesis, donde se proporcionan los argumentos (si la función no necesita argumentos, los paréntesis se dejan vacíos). En nuestro ejemplo, el argumento es el texto que queremos mostrar. Toma en cuenta que para indicar que "¡Hola, Mundo!" es el texto, lo ponemos entre comillas.

Para que el intérprete sepa dónde termina el comando, colocamos un punto y coma al final de la llamada a la función. En este caso, el intérprete se las arreglaría sin esa ayuda, pero es una buena costumbre terminar cada comando con un punto y coma, para que no lo olvides cuando realmente lo necesites.

Ya sabemos qué escribir, y la única pregunta ahora es, ¿dónde hacerlo?

**Entorno de desarrollo en línea**

Afortunadamente, nuestra plataforma utiliza un entorno en línea listo para usarse, como mencionamos en el capítulo anterior. El entorno OpenEDG te permite editar y ejecutar programas escritos en JavaScript. Toma en cuenta que la parte de la pantalla dedicada a este entorno se divide en tres partes. La parte superior es el editor, donde podemos elegir si editar un archivo JavaScript, HTML o CSS (diremos algunas palabras sobre HTML y CSS en un momento). Todos estos archivos juntos forman el código a ejecutar en nuestro entorno de entrenamiento. Nos interesará principalmente la pestaña del archivo JavaScript: app.js. En la parte inferior izquierda de la pantalla, hay una ventana que simula la consola, en la que aparecerán los mensajes del intérprete y la información que escribimos. La ventana del lado derecho está diseñada para mostrar la página en cuyo contexto se ejecuta nuestro código JavaScript. Esta ventana será la menos útil en esta parte del curso.

En el editor, deberías ver la pieza de código que acabamos de discutir, que contiene la función console.log. Intenta ejecutarlo. Debes presionar el botón resaltado con el ícono de reproducción, ubicado directamente sobre el editor. Como resultado, la ventana inferior que simula la consola debería mostrar:

¡Hola, Mundo!

Ve al editor nuevamente y cambia la palabra "Mundo" por tu nombre. Ejecuta el programa nuevamente y verifica lo que aparece en la ventana de la consola. Felicitaciones, acabas de modificar un programa escrito en JavaScript.

Podríamos discutir tu primer programa de JavaScript basado en este ejemplo. Aprendiste la sintaxis, lo ejecutaste en línea, se comprobó su efecto e incluso lo modificaste por ti mismo. Puedes probar todos los ejemplos que discutimos en este curso de esta manera. Sin embargo, en el capítulo anterior, te instamos a configurar tu entorno de desarrollo local. Por lo tanto, sería bueno mostrarte cómo se puede ejecutar este ejemplo en dicho entorno. Y esto requerirá una introducción un poco más larga.

**Entorno de desarrollo local**

El JavaScript del lado del cliente es un lenguaje de la web y solo existe en el ecosistema web. En esta configuración, JavaScript no puede existir por sí mismo. El código JavaScript debe estar incrustado en un documento HTML. Cuando usamos el entorno en línea para ejecutar nuestro programa, se nos ocultaron ciertos aspectos. Esta vez tendremos que mirarlos más de cerca.

**Unas palabras sobre HTML**

**HyperText Markup Language**, o **HTML**, es un conjunto de etiquetas utilizadas para describir la estructura de un sitio web. Nos permite dar a una página el formato de un documento que contiene secciones, encabezados, párrafos, listas y similares. HTML definitivamente está más allá del alcance del curso actual, por lo que presentaremos solo información básica al respecto, lo suficiente para que comprendas dónde y cómo podemos ejecutar el código JavaScript asociado con una página determinada.

Los tipos de etiquetas están predefinidos. Por ejemplo, la etiqueta que especifica un párrafo es <p> y la etiqueta para el encabezado de primer grado (el más grande) es <h1>. El nombre de la etiqueta debe colocarse entre corchetes. Las etiquetas se suelen utilizar en pares, limitando un área determinada del documento (tenemos una etiqueta de apertura y otra de cierre). La etiqueta de cierre es diferente de la etiqueta de apertura porque aparece una barra antes del nombre. Por ejemplo, un párrafo puede verse así:

<p>un párrafo ordinario</p>

A menudo, las etiquetas pueden (y a veces deben) colocarse dentro del rango de otras etiquetas. Por ejemplo, nuestro párrafo debe colocarse dentro de la etiqueta <body>, que separa la parte principal de nuestro documento.

<body>

<p>un párrafo ordinario</p>

</body>

**Documento HTML**

Intentemos crear un HTML sencillo que defina una página vacía.

<!DOCTYPE html>

<html>

<head>

<title>Empty Page</title>

</head>

<body>

</body>

</html>

Comencemos con la declaración <!DOCTYPE html>. Esta no es una etiqueta típica, ya que se utiliza para informar al navegador que todo el documento se ha preparado de acuerdo con HTML5. La descripción del documento en si comienza con la etiqueta <html>, la cual junto con la etiqueta </html> establece los límites del documento. Cualquier otra etiqueta debe estar dentro de estas. Si una etiqueta dada tiene otro contenido, habrá una etiqueta de cierre correspondiente, formando una especie de contenedor.

La siguiente etiqueta, <head>, contiene información adicional sobre el documento, que también debe colocarse en etiquetas. La más básica es la etiqueta <title>, que establece el título de la página mayormente visible en la barra de título del navegador. Después de <head> esta el elemento <body>, y allí se debe colocar el contenido visible de la página web (por ejemplo, nuestro párrafo).

**La etiqueta**<script>

El código JavaScript que ejecutará el navegador en la página debe adjuntarse al HTML utilizando la etiqueta <script>, y existen dos formas de hacerlo. El código se puede incrustar directamente dentro de las etiquetas <script> y </script>, pero esto solo se recomienda cuando el código es corto. Otro enfoque es utilizar el atributo "src" para apuntar a un archivo separado que contiene el código JavaScript. Esto es especialmente cierto cuando se va a usar el mismo código en varias páginas, porque repetir exactamente el mismo código muchas veces es una mala práctica, ya que cualquier cambio debe aplicarse a todos los archivos; y además, aumenta artificialmente el tamaño de la página. La extensión del archivo JavaScript es .js.

HTML es leído por el navegador línea por línea, y las etiquetas se ejecutan justo en el momento en que el navegador analiza la etiqueta <script> (analizar en lenguajes de programación significa un análisis formal del código por parte de una máquina para comprender su estructura). Generalmente las etiquetas <script> se insertan en el encabezado de la página entre las etiquetas <head> y </head>, y podemos insertar muchos de ellos en un archivo, por ejemplo, para incluir código JavaScript de diferentes archivos. Este comportamiento se puede cambiar para scripts externos señalados por el atributo "src" usando los atributos "defer" o "async".

* defer: significa que el script debe ejecutarse después de cargar toda la página.
* async: significa que el script se ejecutará inmediatamente, pero en paralelo al análisis del resto de la página.

**... y un poco acerca de CSS**

**CSS**, o **Cascading Style Sheets**, es un lenguaje utilizado junto con HTML para describir la apariencia de una página y sus elementos. En pocas palabras, HTML describe la estructura de un documento, mientras que CSS describe su presentación.

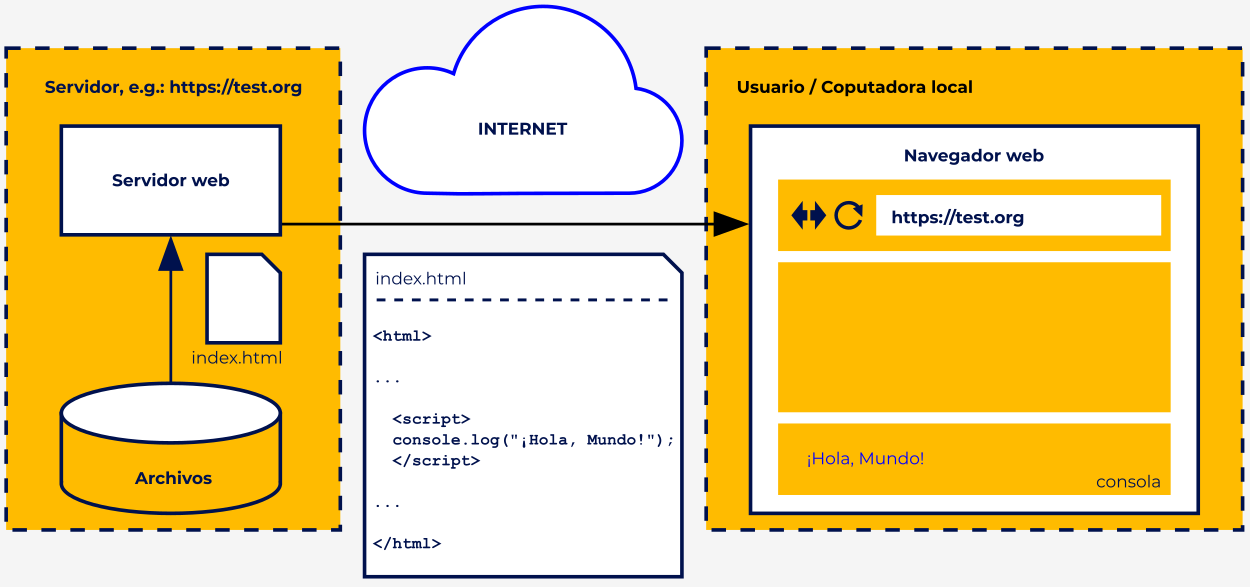
Por ejemplo, en HTML, podemos describir una página que tiene un encabezado, dos párrafos y una tabla de datos.

En CSS, podemos definir qué fuente se usará en toda la página, qué color tendrá el fondo o si el cursor del mouse, cuando se mueve sobre la tabla, debe cambiar de forma.

Entonces podemos tratar CSS como una especie de configuración de la capa visual de la página. Así, la mayoría de las veces el sitio web se construirá sobre la base de un archivo HTML (es decir, una descripción de la estructura), código JavaScript que nos permite agregar, por ejemplo, algunos mecanismos de interacción y un archivo CSS (que describe la presentación de la página). Sin embargo, lo importante es que no habrá página sin un archivo HTML, pero podemos crear fácilmente una página sin usar archivos CSS. La descripción de CSS en sí está fuera del alcance del curso actual y la mencionamos solo por orden.

**Cómo podemos ejecutar nuestra código JavaScript?**

Comencemos con un ejemplo simple, donde el navegador obtiene una página simple (quizás incluso vacía) de *https://test.org*. La dirección es ficticia para este ejemplo, así que no intentes ingresarla. Mira la figura de abajo.



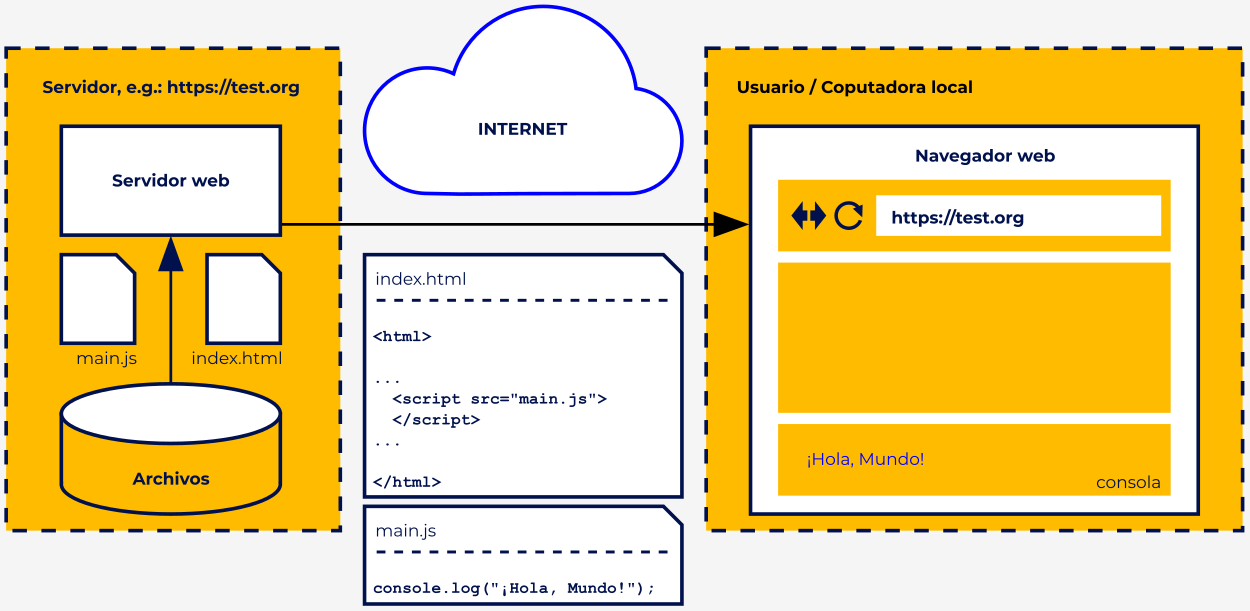
Empecemos por el lado derecho de la figura. El usuario ejecuta un navegador web en su computadora (por ejemplo, Chrome). Usando el atajo de teclado apropiado, se activan las **herramientas de desarrollo** (ver el capítulo anterior) para poder usar la consola. Recuerda que estas herramientas no son necesarias para el uso normal del navegador y, por lo tanto, están ocultas por defecto. Luego, el usuario escribe *https://test.org* (la URL de nuestro sitio falso) en la barra de direcciones.

En el servidor remoto (lado izquierdo del dibujo), asociado a la dirección *https://test.org,* se esta ejecutando un servidor web que, tras recibir una solicitud de nuestro usuario, preparará una respuesta para ello. En el caso más simple, la respuesta solo contendrá un archivo html, que se puede almacenar en el mismo servidor. El archivo html (en este ejemplo, *index.html*) se devuelve al usuario y el navegador lo procesa. Si se define algún contenido (por ejemplo, un párrafo con texto), se mostrará en la ventana del navegador.

Sin embargo, nos interesa más el hecho de que el archivo index.html contiene las etiquetas <script> y </script>, con una pieza de código JavaScript entre ellas. ¿Lo reconoces? Obviamente, esto es un intento de mostrar nuestro "¡Hola, mundo!" en la consola al cargar la página, se debe ejecutar el código colocado dentro de las etiquetas <script> y, si las herramientas para desarrolladores están habilitadas y el panel de la consola está visible, la consola mostrará "¡Hola, Mundo!".

Como dijimos antes, la etiqueta <script> se puede usar de una manera diferente, no solo para restringir el lugar donde escribimos código JavaScript directamente. Si usamos el atributo "src" en esta etiqueta, podemos indicar un archivo JavaScript separado que se adjuntará aquí.

Revisa el código en la pagina siguiente. En el archivo index.html se encuentra de nuevo la etiqueta <script>. Esta vez no hay código JavaScript colocado después, pero al usar el atributo "src", se indica que el código del archivo main.js debe adjuntarse aquí.

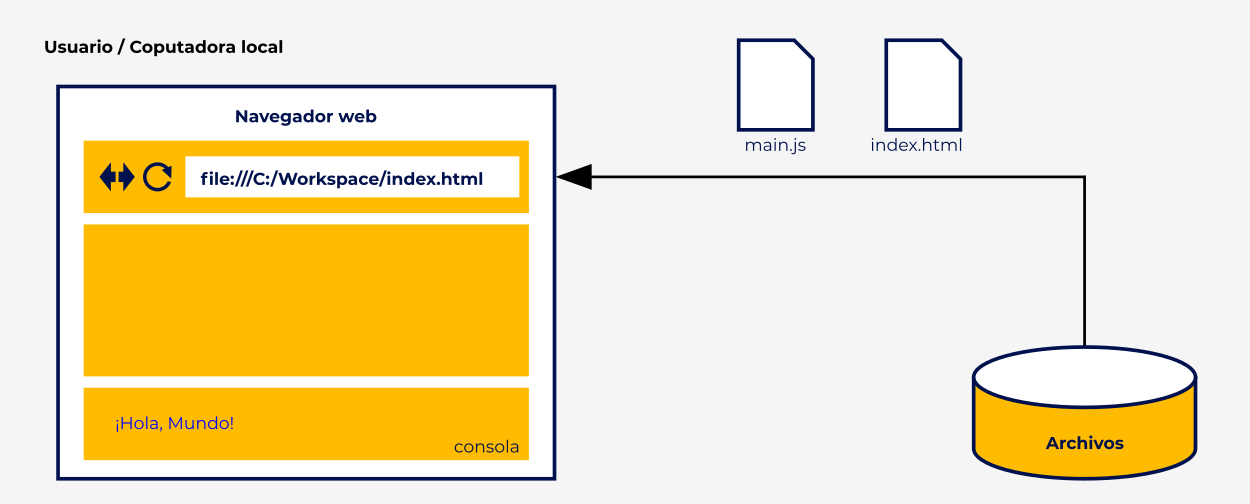


**hacer clic para agrandar**

Todo funcionará exactamente igual que en el escenario anterior, excepto que el servidor web proporcionará el archivo main.js además de index.html. El usuario no notará ninguna diferencia. Por supuesto, colocar nuestro código en un servidor remoto solo para probarlo sería un poco engorroso.

Tenemos otra posibilidad, que es que podemos cargar un archivo html local (es decir, uno que está en nuestra computadora) en el navegador. Si este código contiene una etiqueta <script> que indica algún archivo JavaScript, entonces este archivo también se cargará desde los recursos locales.

La imagen muestra un escenario simple en el que el usuario carga un archivo index.html local en el navegador, en el que hay una referencia a main.js (por lo que este archivo también se cargará automáticamente).



**hacer clic para agrandar**

Puedes cargar un archivo html local escribiendo su ruta local después de archivo:/// en la barra de direcciones, o simplemente abriéndolo en tu navegador usando el comando Abrir del menú. Dado que el menú de los navegadores suele estar oculto, una forma más sencilla puede ser utilizar un acceso directo para abrir documentos existentes en las aplicaciones. El atajo es universal, no solo para navegadores y probablemente ya lo hayas visto:



o en el caso de macOS:



**De acuerdo, tal vez finalmente podamos ejecutar algo....**

Para ejecutar esto localmente, deberás abrir el editor de código de tu elección. Crea un nuevo archivo con la extensión .html (el nombre del archivo no importa, pero es una buena práctica evitar los espacios en el nombre del archivo). Coloca el siguiente código en este archivo y guárdalo.

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Empty Page</title>**

**<script src="main.js"></script>**

**</head>**

**<body>**

**</body>**

**</html>**

Luego, en el mismo editor, crea otro archivo, esta vez llamado main.js (este es el nombre que usamos en nuestro archivo html). Debe contener una línea que has visto antes:

**console.log("¡Hola, Mundo!");**

Guarda los cambios y ve al navegador. Abre una nueva pestaña, habilita las herramientas de desarrollador (se abren para una pestaña en particular) y seleccione la herramienta de consola. Toma un momento para acostumbrarte al diseño de las herramientas de desarrollo (cada herramienta, incluida la consola, debe colocarse en un panel separado, que se puede seleccionar).

Y ahora un pequeño desafío. Intenta modificar el archivo html tu mismo para que no haga referencia al archivo main.js. En su lugar, el mismo código JavaScript que escribimos en main.js debe colocarse directamente después de la etiqueta <script>. Si tienes problemas, vuelva al primer dibujo de esta sección.

Para ejecutar este código en el entorno en línea, colócalo dentro de las pestañas HTML y, si es necesario, presiona el botón ejecutar.

**Ejecutando el código directamente en la consola**

Tenemos otra opción bastante conveniente cuando se trata de ejecutar fragmentos cortos de código JavaScript en el navegador (y nuestro programa, que consta de una instrucción, es definitivamente corto). Como dijimos antes, la consola no solo se usa para mostrar información, sino que también te permite ejecutar instrucciones de JavaScript individuales. Estas instrucciones deben ejecutarse en el contexto de alguna página HTML. Sin embargo, no necesariamente tienes que escribir tu página, como hicimos hace un momento. Intenta abrir una nueva pestaña y escribe about:blank en la barra de direcciones. Esta es una pseudodirección que le dice a su navegador que genere y cargue una página HTML en blanco.

Luego, ejecuta las herramientas de desarrollo. Al principio, podemos comprobar cómo se ve el HTML generado por el navegador. Para hacerlo, seleccione la primera herramienta del panel (en Chrome, será Elementos, en Firefox Inspector). Deberías ver un código html mínimo:

**<html>**

**<head></head>**

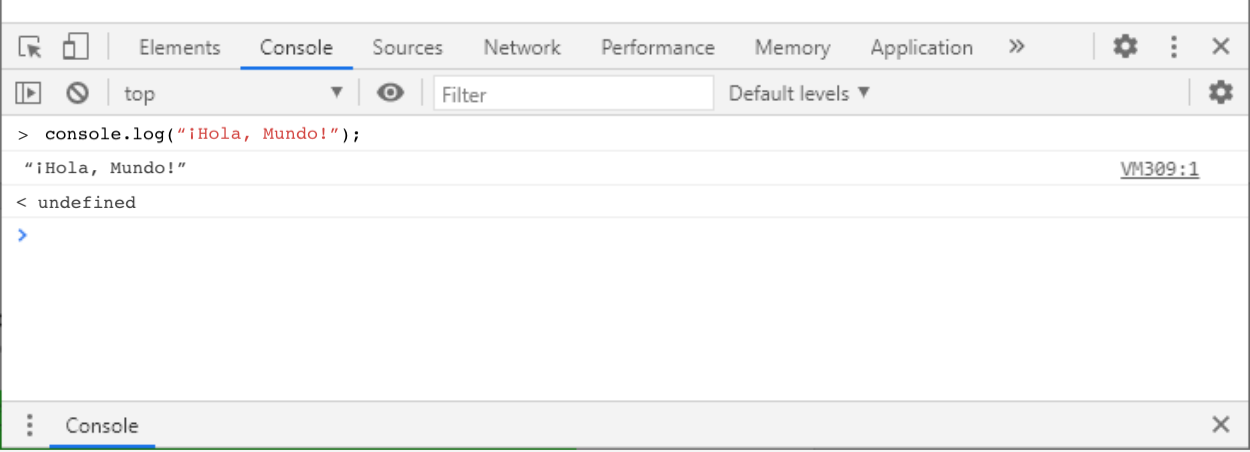
**<body></body>**

**</html>**

Ahora elige la consola de las herramientas para desarrolladores. Deberías ver un aviso, generalmente un signo > o >> seguido de un cursor parpadeante (si no hay cursor, has clic en el aviso). Luego puedes ingresar la instrucción que mostrará "¡Hola, Mundo!" en la consola (usando la función console.log). El escenario se muestra en la siguiente figura.

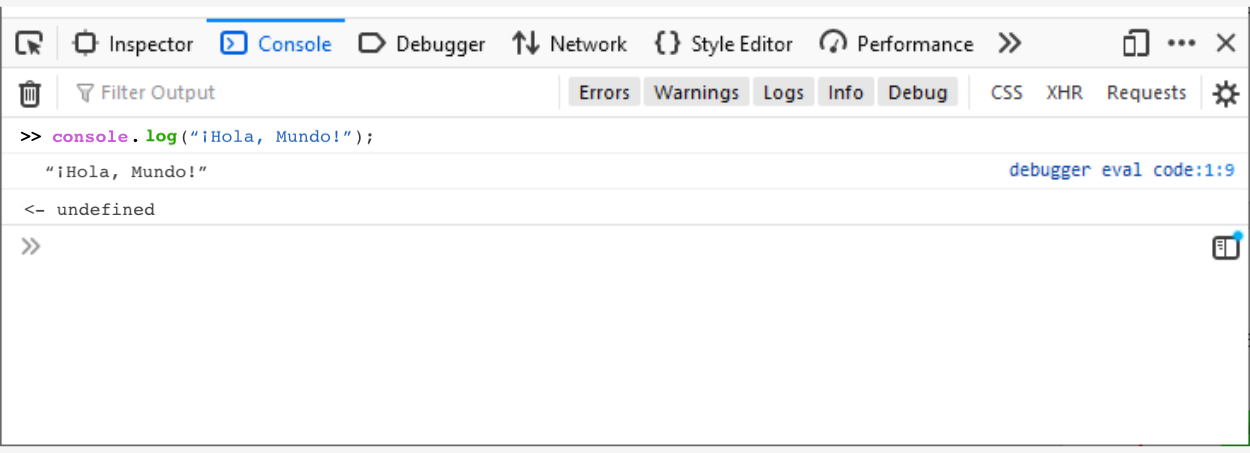


De hecho, independientemente del navegador, deberíamos obtener el mismo efecto: la consola mostrará el texto que especificamos. En el caso de Chrome (que se ejecuta en el sistema operativo Windows), la consola debería verse así después de completar esta tarea:



**hacer clic para agrandar**

En el caso de Firefox (también Windows) de esta forma:



Para ambos navegadores, las ventanas del depurador que contiene la consola pueden variar mínimamente según la versión del software y el sistema operativo que lo ejecute. Las herramientas de desarrollo se pueden mover. Se pueden ubicar en la parte inferior del navegador, como en los ejemplos que se muestran, pero también se pueden colocar en el lado izquierdo o derecho de la ventana (o como una ventana completamente separada). Así que no te sorprendas si el diseño de tu navegador es ligeramente diferente al de las imágenes.

**Resumen**

Nuestro primer programa se lanzó en un entorno en línea al principio. Este entorno nos permite ocultar ciertos detalles que no son importantes para nosotros en esta etapa del curso. Todos los ejercicios y ejemplos que discutiremos deben realizarse en este entorno.

Sin embargo, de vez en cuando, sería bueno que intentaras hacer el ejemplo elegido también en el entorno local. Está mucho más cerca de lo que realmente se usa en el trabajo de un desarrollador web. Ejecutar código JavaScript en el entorno local puede parecer un poco engorroso al principio, pero afortunadamente esta es solo una primera impresión. Recuerda, para probar instrucciones simples, solo necesitas usar la consola con una página vacía (por ejemplo, about:blank). Si deseas probar un código un poco más grande, es mejor crear un archivo html que se refiera al archivo que contiene nuestro código JavaScript usando la etiqueta <script>.

**Tareas**

**Tarea 1**

Emplea console.log para enviar tu nombre completo a la consola.

Ejemplo

console.log("Mary Stuart");

**Tarea 2**

Muestra tu año de nacimiento.

Ejemplo

console.log("1542");

**Tarea 3**

Vuelve a intentar mostrar tu año de nacimiento, esta vez pasando la fecha sin las comillas.

Ejemplo

console.log(1542);

**Tarea 4**

Podemos pasar varios argumentos a console.log separados por comas, por ejemplo:

console.log("abc", "def", "ghi");

Envía información sobre ti a la consola en el formato: Nombre Apellido (Año) por ejemplo *Mary Stuart (1542)*.

* Enviar toda la información como un solo argumento.
* Enviar la información nombre, apellido, año como argumentos separados.

Ejemplo

console.log("Mary Stuart (1542)");

console.log("Mary", " Stuart ", "(1542)");

**Tarea 5**

Envía la misma información (nombre, apellido, año) a la consola, no una al lado de la otra, sino en líneas consecutivas.

Ejemplo

console.log("Mary");

console.log("Stuart ");

console.log("(1542)");

**Tarea 6**

Una cadena se puede concatenar usando el signo +, por ejemplo "abc" + "def" serán tratados como "abcdef". Intenta escribir tu nombre, apellido y año de nacimiento en una línea nuevamente, esta vez no separados por comas, sino por el signo +.

Ejemplo

console.log("Mary" + " Stuart" + "(1542)");

**Tarea 7**

Coloca espacios en los lugares apropiados, de modo que cuando se muestre, se obtenga el mismo efecto que en la **Tarea 4**.

Ejemplo

console.log("Mary " + " Stuart " + "(1542)");

**Fundamentos de JavaScript (JSE)  
Módulo 2**

**Variables, Tipos de Datos, Conversión de Tipos de Datos y Comentarios**

Después de completar el Módulo 2, el estudiante:

* Tendrá los conocimientos y las habilidades para trabajar con variables (es decir: nombrar, declarar, inicializar y modificar sus valores).
* Comprenderá conceptos como el alcance, los bloques de código, el sombreado y el hoising.
* Conocerá las propiedades básicas de los tipos de datos primitivos, como boolean, number, bigint, undefined, null, y será capaz de utilizarlos.
* Estará familiarizado con las propiedades básicas del tipo de dato primitivo string o cadena, incluidos los literales de cadena: comillas simples o dobles, el carácter de escape, la interpolación de cadenas, propiedades y métodos básicos.
* Conocerá las propiedades básicas de tipos de datos complejos como Array y Object (tratados como registros) y será capaz de usarlos en la práctica.

**Fundamentos de JavaScript 1 (JSE):**

**Módulo 2**

**Sección 1**

**Variables**

**Temas en esta sección:**

Nombrar, declarar e inicializar variables.

Declaraciones y modo estricto.

Cambio de valores en variables.

Constantes.

Alcance (bloques, sombreado, y hoisting).

**Variables**

La capacidad de escribir información diversa en la pantalla, como "¡Hola, Mundo!" puede ser divertido por un tiempo, pero no es una forma universal de escribir programas. Es hora de comenzar a aprender más sobre los elementos del rompecabezas que finalmente te permitirán crear programas que resuelvan problemas reales.

Existen bastantes de estos elementos, y los presentaremos gradualmente, aunque no necesariamente en una cronología simple. A menudo volveremos a lo que ya se ha discutido, ampliando la información anterior con algo nuevo. A veces también avanzaremos, utilizando mecanismos que se explicarán con detalle más adelante. Al principio puede parecer un poco abrumador, pero con el tiempo todo debería comenzar a fusionarse en una imagen coherente.

El primer elemento de programación del que hablaremos es la **variable**. Es posible que conozcas el nombre de una variable de las matemáticas, donde significa un símbolo que se usa como identificador para diferentes valores que pueden cambiar. Tienen un papel similar en la programación.

¿Para qué las necesitamos realmente? Como puedes adivinar, la mayoría de los programas son bastante complejos y rara vez podemos resolver el problema con una sola operación. Por lo general, el programa constará de muchas más operaciones, cada una de las cuales puede producir algunos resultados intermedios, que serán necesarios en los siguientes pasos. Las variables nos permiten almacenar dichos resultados, modificarlos o alimentarlos en operaciones posteriores.

**Nombrado de variables**

Imagína a las variables como contenedores en los que puedes almacenar cierta información (dicha información se denominará valores de variables). Cada contenedor debe tener su propio nombre, mediante el cual podremos indicarlo claramente.

Por lo general, tenemos bastante libertad cuando se trata de inventar estos nombres, pero recuerda que deben referirse a lo que almacenaremos en la variable (por ejemplo, altura, color, contador de pasos, etc.). Por supuesto, JavaScript no verificará la correlación entre el nombre y el contenido de la variable; es simplemente una de las muchas buenas prácticas que facilitan que nosotros y otros entendamos el código más adelante.

En la mayoría de los lenguajes de programación, se debe declarar una variable antes de usarla, y JavaScript no es una excepción. Declarar una variable es simplemente "reservar" el nombre de la variable. De esta manera, le informamos al programa que en la parte posterior de la ejecución, usaremos este nombre para referirnos a nuestro contenedor, para recuperar un valor de él o guardar un valor en él.

En JavaScript, los nombres de las variables pueden constar de cualquier secuencia de letras (minúsculas y mayúsculas), dígitos, guiones bajos y signos de dólar, pero no deben comenzar con un dígito. Hay una lista de palabras reservadas que no se pueden usar como nombres de variables (consulta la tabla a continuación).

Lo importante también es que el intérprete de JavaScript distingue entre minúsculas y mayúsculas, también en nombres de variables, por lo que nombres como test, Test, o TEST serán tratados como variables diferentes.

|  |  |  |  |
| --- | --- | --- | --- |
| Los nombres de las variables en JavaScript pueden ser prácticamente cualquier cadena de caracteres. Sin embargo, hay un conjunto de palabras reservadas que no se pueden usar para nombrar variables, funciones o cualquier otra cosa. Son partes integrales del lenguaje y se les asigna un significado que no se puede cambiar. A continuación encontrarás una lista de ellas: | | | |
| abstract | arguments | await | boolean |
| break | byte | case | catch |
| char | class | const | continue |
| debugger | default | delete | do |
| double | else | enum | eval |
| export | extends | false | final |
| finally | float | for | function |
| goto | implements | if | import |
| in | instanceof | int | interface |
| let | long | native | new |
| null | package | private | protected |
| public | return | short | static |
| super | switch | synchronized | this |
| throw | throws | transient | true |
| try | typeof | var | void |
| volatile | while | with | yield |

**Declarando variables**

Como mencionamos anteriormente, **declaramos** la variable para reservarle un nombre. Esto es una simplificación, porque de hecho, el espacio de memoria también está reservado para la variable, pero al programar en JavaScript, prácticamente nunca tenemos que pensar en lo que sucede en la memoria. Por lo general, los valores almacenados en la variable podrán modificarse durante la ejecución del programa (después de todo, son "variables"). ¿Por qué? Porque podemos declarar variables cuyos valores no se pueden cambiar. Para ser honesto, ya ni siquiera las llamamos variables, las llamamos **constantes**. Para las declaraciones, usamos las palabras clave var o let para **variables** y const para **constantes**. Por ahora, sin embargo, sigamos con las variables habituales y volveremos a las constantes en un momento.

Analicemos el siguiente ejemplo de código (también lo encontrarás en la ventana del editor; ejecútalo allí y observa los resultados en la consola):

var height;

console.log(height); // -> undefined

console.log(weight); // -> Uncaught ReferenceError: weight is not defined

La primera línea es la **declaración** de la variable (podemos ver la palabra clave var). Esta declaración significa que la palabra height sera tratada como el nombre del contenedor para ciertos valores.

La declaración, como otras instrucciones de JavaScript, debe terminar con un punto y coma. En la segunda línea, tratamos de escribir el valor de esta variable (es decir, lo que hay en el contenedor) en la consola. Debido a que aún no hemos puesto nada allí, el resultado no está definido (el intérprete conoce esta variable, pero aún no tiene valor, el valor no está definido). En la siguiente línea, tratamos de imprimir el contenido de la variable weight... que olvidamos declarar. Esta vez, veremos ReferenceError. El intérprete de JavaScript, que ejecuta nuestro programa, nos ha informado que no conoce una variable con este nombre (por lo que la variable en sí no está definida).

**Variables - Declarando variables cont.**

En el ejemplo, usamos la palabra clave var. La alternativa es la palabra clave let. Usamos ambas palabras clave de la misma manera. Ambas están destinados para declarar variables, y ambas se pueden encontrar en diferentes ejemplos en Internet o en libros. Sin embargo, no son exactamente iguales y discutiremos las diferencias en su funcionamiento más adelante en este capítulo (incluso en varios lugares).

La palabra clave var proviene de la sintaxis JavaScript original, y la palabra clave let se introdujo mucho más tarde. Por lo tanto, encontrarás var en programas más antiguos. Actualmente, se recomienda enfáticamente usar la palabra let por razones que discutiremos en un momento.

Entonces, echemos un vistazo a nuestro ejemplo reescrito esta vez usando la palabra clave let.

let height;

console.log(height); // -> undefined

Una de las diferencias básicas en el uso de var y let es que let nos impide declarar otra variable con el mismo nombre (se genera un error). El uso de var le permite volver a declarar una variable, lo que puede generar errores en la ejecución del programa.

var height;

var height;

console.log(height); // -> undefined

El ejemplo anterior demuestra la posibilidad de volver a declarar una variable usando la palabra clave var. En esta situación, no causará un error, pero en programas más complejos, una redeclaración, especialmente por accidente, puede tener consecuencias. Al declarar con let, el intérprete verifica si dicha variable ya ha sido declarada, sin importar si se usó let o var en la declaración anterior.

let height;

let height; // -> Uncaught SyntaxError: Identifier 'height' has already been declared

console.log(height);

Así que usa let para declarar variables, aunque solo sea porque no quieres volver a declarar accidentalmente una variable.

**Inicializando variables**

Después de una declaración exitosa, la variable debe ser **inicializada**, en otras palabras, debe recibir su primer valor. La **inicialización** se realiza asignando un determinado valor a una variable (indicado por su nombre). Para asignarlo usamos el operador =.

Puedes asignar a una variable: un valor específico; el contenido de otra variable; o, por ejemplo, el resultado devuelto por una función.

La inicialización se puede realizar junto con la declaración o por separado como un comando independiente. Es importante ingresar el primer valor en la variable antes de intentar leerla, modificarla o mostrarla.

let height = 180;

let anotherHeight = height;

let weight;

console.log(height); // -> 180

console.log(anotherHeight); // -> 180

weight = 70;

console.log(weight); // -> 70

En el ejemplo anterior (verifícalo en el editor), las declaraciones de las variables height y anotherHeight se combinan con su inicialización, mientras que la variable weight se declara e inicializa por separado. Las variables height y weight se inicializan proporcionando valores específicos (más precisamente, un número), mientras que la variable anotherHeight recibe un valor leído de la variable de height. Los valores de todas las variables se muestran en la consola.

Por cierto, presta atención a una cosa. Si especificas un nombre de variable en console.log, el intérprete lo reconoce y muestra su valor. Si pones el mismo nombre entre comillas, se tratará como texto sin formato y se mostrará como tal.

let height = 180;

console.log(height); // -> 180

console.log("height"); // -> height

**Declaraciones y modo estricto**

JavaScript tuvo algunos cambios importantes introducidos en 2009 y 2015. La mayoría de estos cambios ampliaron la sintaxis del lenguaje con nuevos elementos, pero algunos de ellos se referían solo al funcionamiento de los intérpretes de JavaScript. A menudo se trataba de aclarar el comportamiento de los intérpretes en situaciones potencialmente erróneas, como en los casos de inicialización de variables sin ninguna declaración previa.

Veamos un ejemplo:

height = 180;

console.log(height); // -> 180

A primera vista, puedes ver que nos hemos olvidado de declarar la variable height. La sintaxis de JavaScript original permitía tal negligencia, y en el momento de la inicialización hizo esta declaración por nosotros. Parece una solución bastante buena, pero desafortunadamente a veces puede conducir a situaciones ambiguas y potencialmente erróneas (diremos algunas palabras más al respecto mientras discutimos el alcance).

Vamos a modificar nuestro ejemplo:

"use strict";

height = 180; // -> Uncaught ReferenceError: height is not defined

console.log(height);

Al principio de nuestro código, agregamos "use strict";. Esta sentencia ha cambiado radicalmente el comportamiento del intérprete. ¿Por qué? Lo usamos cuando queremos obligar al intérprete a comportarse de acuerdo con los estándares modernos de JavaScript. Entonces, siempre que no estés ejecutando un código realmente antiguo, siempre debes usarlo. Y esta vez, usar una variable sin su declaración anterior se trata como un error.

La frase "use strict"; debe colocarse al principio del código. Hará que el intérprete se ocupe del resto del código utilizando el modo estricto, que es el estándar moderno de JavaScript. Todos los demás ejemplos de nuestro curso estarán preparados para funcionar en este modo de forma predeterminada, incluso si "use strict"; no siempre aparezca al principio del código.

**Constantes**

La palabra clave const se usa para declarar contenedores similares a variables. Dichos contenedores se denominan **constantes**. Las constantes también se utilizan para almacenar ciertos valores, pero una vez que se han ingresado valores durante la inicialización, ya no se pueden modificar. Esto significa que este tipo de contenedor se declara e inicializa simultáneamente. Por ejemplo, la siguiente declaración de la constante greeting es correcta:

const greeting = "¡Hola!";

Pero esta próxima definitivamente causa un error:

const greeting; // -> Uncaught SyntaxError: Missing initializer in const declaration

greeting = "¡Hola!";

Como dijimos, un cambio en la constante es imposible. Esta vez la declaración es correcta, pero tratamos de modificar el valor almacenado en la constante.

const greeting = "¡Hola!";

greeting = "Hi!"; // -> Uncaught TypeError: Assignment to constant variable.

El propósito principal de una constante es erradicar la posibilidad de cambiar accidentalmente un valor almacenado en ella. Esto es importante cuando tenemos algunos valores que realmente nunca deberían cambiar. Los ejemplos típicos de constantes son rutas a recursos, tokens y otros datos que nunca cambian durante la vida útil del script.

Pero las constantes también se pueden usar como subresultados en los cálculos o en otros lugares donde la información que se recopiló o calculó no cambiará más. El uso de una const, además de evitar que un valor se cambie por error, permite que el motor de JavaScript optimice el código, lo que puede afectar su rendimiento.

**Alcance**

Hasta ahora, asumimos que después de declarar una variable, su nombre podría usarse en todo el código del programa (es decir, el alcance de la variable es global). Esto no es del todo cierto: el alcance de una variable depende de dónde se declare. Desafortunadamente, para una buena comprensión del alcance de una variable, necesitamos aprender algunos elementos de programación más, como instrucciones o funciones condicionales, que se analizarán con detalle más adelante en el curso. Así que aquí nos limitaremos a la información básica y volveremos a este tema en diferentes partes del curso. Uno de los elementos básicos que influyen en el alcance de las variables es un **bloque del programa**.

**Bloques del Programa**

Podemos separar el código de un programa en bloques. En los bloques que creamos usando llaves {}, hay un conjunto de instrucciones que, por alguna razón, deben tratarse de forma independiente. Los bloques suelen estar asociados a instrucciones condicionales, bucles o funciones, de las que hablaremos más adelante. También podemos separar un bloque de un programa que no tenga nada en especial, simplemente eligiendo un determinado rango de instrucciones (en la práctica, esto no está especialmente justificado, y por ahora solo lo haremos por motivos educativos).

Veamos un ejemplo:

let counter;

console.log(counter); // -> undefined

{

counter = 1;

console.log(counter); // -> 1

}

counter = counter + 1;

console.log(counter); // -> 2

Primero, declaramos la variable counter. Luego abrimos un bloque dentro del cual inicializamos esta variable y mostramos su contenido. Fuera del bloque, aumentamos el valor almacenado en la variable en 1 y lo mostramos nuevamente. En este caso, el intérprete ejecutará el programa como si no hubiera notado el bloque, pasando por las instrucciones antes del bloque, en el bloque y después del bloque. Crear un bloque aquí, sin, por ejemplo, instrucciones condicionales, no tiene una justificación real, es solo un ejemplo del uso de llaves {}.

Los bloques del programa se pueden anidar, es decir, podemos crear un bloque dentro de otro.

let counter;

console.log(counter); // -> undefined

{

counter = 1;

{

console.log(counter); // -> 1

}

}

counter = counter + 1;

console.log(counter); // -> 2

Por cierto, toma en cuenta que el código dentro del bloque se ha movido a la derecha. Esto se denomina **indentación**. Para un intérprete de JavaScript, no importa en absoluto, pero definitivamente aumenta la legibilidad del código, lo que permite a los lectores (incluso a tí) descubrir rápidamente qué partes del código están dentro y cuáles están fuera del bloque. Los editores de código suelen agregar sangrías en los lugares correctos por sí mismos, pero es un buen hábito recordarlo y, si no aparecen automáticamente, da formato al código a mano.

Es hora de seguir adelante para determinar qué está pasando realmente con estos alcances. Desafortunadamente, los alcances de las variables (y constantes) declaradas con let y const son ligeramente diferentes a las declaradas con var. Así que los discutiremos de forma independiente.

**let y const**

La primera regla es simple. Si declaramos alguna variable o constante usando let o const, respectivamente, fuera de los bloques de código, serán **globales**. Con esto queremos decir que sus nombres serán visibles en todo el programa, fuera de los bloques, dentro de los bloques, en las funciones, etc. Podremos referirnos a ellos en cualquier lugar por sus nombres y, por supuesto, tendremos acceso a sus valores.

¿Qué sucede si declaramos algo usando let o const dentro de un bloque? Esto creará una variable o constante local. Será visible solo dentro del bloque en el que se declaró y en los bloques que opcionalmente se pueden anidar en él.

Veamos un ejemplo sencillo:

let height = 180;

{

let weight = 70;

console.log(height); // -> 180

console.log(weight); // -> 70

}

console.log(height); // -> 180

console.log(weight); // -> Uncaught ReferenceError: weight is not defined

La variable height, declarada fuera del bloque, es global. La variable weight es local: su alcance está limitado por el bloque en el que se declaró. Esto es claramente visible cuando se intenta mostrar los valores de ambas variables dentro y fuera del bloque. También podemos probar el caso con bloques anidados

let height = 200;

{

let weight = 100;

{

let info = "tall";

console.log(height); // -> 200

console.log(weight); // -> 100

console.log(info); // -> tall

}

console.log(height); // -> 200

console.log(weight); // -> 100

console.log(info); // -> Uncaught ReferenceError: info is not defined

}

Como puedes ver, la variable info declarada en el bloque más interno solo es visible dentro de él. La variable weight es visible tanto dentro del bloque en el que fue declarada como dentro del bloque anidado en ella. Y la variable global height es visible en todas partes.

Simple, ¿no?

**var**

En el caso de declaraciones de variables usando la palabra clave var, la situación es ligeramente diferente. La variable declarada utilizándola fuera de los bloques será, como en el caso de let, global, es decir, será visible en todas partes. Si la declaras dentro de un bloque, entonces... bueno, por lo general volverá a ser global.

Comencemos con un ejemplo simple:

var height = 180;

{

var weight = 70;

console.log(height); // -> 180

console.log(weight); // -> 70

}

console.log(height); // -> 180

console.log(weight); // -> 70

Como era de esperar, ambas variables, height y weight, resultan ser globales. ¿Las variables declaradas usando var siempre, independientemente del lugar de declaración, serán globales? Definitivamente no. El problema es que var ignora los bloques de programa ordinarios, tratándolos como si no existieran. Entonces, ¿en qué situación podemos declarar una variable local usando var? Sólo dentro de una función. Dedicaremos mucho espacio a discutir la función y luego volveremos al problema del alcance de la variable. Ahora intentaremos presentar y discutir solo un ejemplo simple, que mostrará que las variables var a veces también son locales.

**Unas breves palabras sobre funciones**

Empecemos explicando qué son las **funciones**. A menudo sucede que una determinada pieza de código, que realiza alguna tarea específica, se utilizará muchas veces. Sí, puedes copiar este fragmento de código, todas sus instrucciones, en cualquier lugar donde desees utilizarlo. Sin embargo, esto sería muy ineficiente. En primer lugar, el tamaño de nuestro programa crecería innecesariamente. En segundo lugar, si quisiéramos hacer algunos cambios en este código, por ejemplo, para corregir algún error, tendríamos que hacerlo en todos los lugares donde lo usamos.

Una solución simple a esto problema es una función. Una **función** es solo una pieza de código separada que nombras, de la misma manera que nombras una variable. Si deseas usarla en algún lugar, simplemente se hace referencia a ella con ese nombre (decimos que *llamamos* a la función).

La declaración de una función simple, digamos testFunction, puede verse así:

function testFunction() {

console.log("Hola");

console.log("Mundo");

}

La forma de definir la función que se muestra en el ejemplo es una de varias disponibles en JavaScript. La definición comienza con la palabra clave function, seguida del nombre de la función que inventamos. Después del nombre, verás paréntesis, que opcionalmente podrían contener parámetros pasados a la función (volveremos a esto cuando analicemos funciones con más precisión). Luego abrimos el bloque del programa, que contiene las instrucciones pertenecientes a la función. Al definir una función, las instrucciones contenidas en la función no se ejecutan. Para ejecutar la función, debes llamarla de forma independiente, usando su nombre.

Echa un vistazo al siguiente programa.

console.log("Comencemos:"); // -> Comencemos:

console.log("Hola"); // -> Hola

console.log("Mundo"); // -> Mundo

console.log("y otra vez:"); // -> y otra vez:

console.log("Hola"); // -> Hola

console.log("Mundo"); // -> Mundo

console.log("y una vez más:"); // -> y una vez más:

console.log("Hola"); // -> Hola

console.log("Mundo"); // -> Mundo

Imprimirá una secuencia de texto en la consola:

Comencemos:

Hola

Mundi

y otra vez:

Hola

Mundo

y una vez más:

Hola

Mundo

Podemos reescribir el mismo programa usando nuestra función testFunction. Declarémoslo de nuevo y llamémosla en los lugares correctos:

function testFunction() {

console.log("Hola");

console.log("Mundo");

}

console.log("Comencemos:");

testFunction();

console.log("y otra vez:");

testFunction();

console.log("y una vez más:");

testFunction();

**salida**

El efecto del programa será el mismo que antes (prueba ambos ejemplos).

**La palabra clave var - continuación**

Después de esta breve introducción a las funciones (obviamente, esta no es nuestra última reunión con ellas), volvamos a la palabra clave var y los ámbitos de las variables.

Si declaramos una variable usando la palabra clave var dentro de una función, su alcance se limitará solo al interior de esa función (es un alcance local). Esto significa que el nombre de la variable se reconocerá correctamente solo dentro de esta función.

Consideremos el siguiente ejemplo:

var globalGreeting = "Buenos";

function testFunction() {

var localGreeting = "Días";

console.log("función:");

console.log(globalGreeting);

console.log(localGreeting);

}

testFunction();

console.log("programa principal:");

console.log(globalGreeting);

console.log(localGreeting); // -> Uncaught ReferenceError: localGreeting is not defined

En primer lugar, ejecuta este programa y observa los resultados en la consola. ¿Qué pasó y, sobre todo, por qué pasó?

Echemos un vistazo más de cerca al código. En el ejemplo, declaramos la variable global globalGreeting. Luego definimos la función testFunction, dentro de la cual declaramos la variable local localGreeting. Luego llamamos a la función testFunction, que resultó en escribir los valores de ambas variables (dentro de la función, tenemos acceso tanto a la variable global como a las locales). Intentar acceder a la variable local localGreeting fuera de la función fallará. Así que finalmente logramos demostrar que las declaraciones de variables que usan la palabra var también pueden ser locales.

**Sombreado**

JavaScript permite sombreado de variables. ¿Que significa eso? Significa que podemos declarar una variable global y una variable local con el mismo nombre.

En el alcance local, en el que declaramos una variable local usando su nombre, tendremos acceso al valor local (la variable global está oculta detrás de la local, por lo que no tenemos acceso a ella en este alcance local). Usar este nombre fuera del alcance local significa que nos referiremos a la variable global. **Sin embargo, esta no es la mejor práctica de programación y debemos evitar tales situaciones.** No es difícil adivinar que con un poco de falta de atención, el uso de este mecanismo puede conducir a situaciones no deseadas y probablemente a errores en el funcionamiento del programa.

Si queremos evitar tales situaciones, sería bueno ver exactamente de qué se trata. Comencemos con un ejemplo sin sombreado:

let counter = 100;

console.log(counter); // -> 100

{

counter = 200;

console.log(counter); // -> 200

}

console.log(counter); // -> 200

La variable counter, declarada al principio del programa, es una variable global. A lo largo del programa, también dentro del bloque, operamos sobre esta misma variable. Un pequeño cambio en el código es suficiente para que el programa se comporte de manera completamente diferente.

let counter = 100;

console.log(counter); // -> 100

{

let counter = 200;

console.log(counter); // -> 200

}

console.log(counter); // -> 100

¿Ves la diferencia? Esta vez en el bloque, en lugar de counter = 200; (es decir, cambios en el contenido de la variable counter global), aparece la instrucción let counter = 200; (es decir, declaraciones de la variable local combinada con su inicialización). El intérprete consideraría que tal situación es incorrecta si la redeclaración apareciera en el mismo alcance.

Sin embargo, la declaración es local (tiene un alcance diferente al global) y todas las referencias a la variable con este nombre dentro del bloque se referirán a esta variable local. Fuera del bloque, la variable global se seguirá viendo con el mismo nombre. Presta atención a los valores que muestra la consola.

**Sombreado - continuación**

El sombreado no solo puede estar relacionado con la situación en la que una variable local cubre una variable global. Si aparecen alcances anidados (por ejemplo, bloques anidados en el caso de una declaración let), la variable local declarada en un bloque más anidado eclipsará la variable local del mismo nombre declarada en el bloque externo.

El sombreado también está presente en las declaraciones de variables que usan la palabra var, y esta vez el alcance local no está limitado por el bloque de programa, sino por el bloque de funciones.

var counter = 100;

function testFunction() {

var counter = 200;

console.log(counter);

}

console.log(counter); // -> 100

testFunction(); // -> 200

console.log(counter); // -> 100

En la mayoría de los casos, esto no es deseable, así que trate de evitar dar los mismos nombres de variable a varias variables, independientemente de dónde las declares.

**Hoisting**

¿Recuerdas que dijimos que todas las variables deben declararse antes de su uso? Esto no es del todo cierto, y realmente la palabra "debería" encaja mejor que "debe". Por supuesto, una buena práctica es siempre declarar las variables antes de usarlas. Y apégate a esto. Pero la sintaxis de JavaScript original permite algunas desviaciones de esta regla.

El intérprete de JavaScript escanea el programa antes de ejecutarlo, buscando errores en su sintaxis, entre otras cosas. Hace una cosa más en esta ocasión. Busca todas las declaraciones de variables y las mueve al principio del rango en el que fueron declaradas (al principio del programa si son globales, al principio del bloque si es una declaración let local, o al principio de la función si es una declaración local var). Todo esto sucede, por supuesto, en la memoria del intérprete, y los cambios no son visibles en el código.

**Hoisting**, es un mecanismo bastante complejo y francamente bastante incoherente. Comprenderlo bien requiere la capacidad de usar libremente muchos elementos de JavaScript, que aún no hemos mencionado.

Además, es más una curiosidad que algo práctico que usarás al escribir programas, por lo que veremos solo un pequeño ejemplo que nos permitirá comprender aproximadamente su principio. Esto puede facilitarte la comprensión de algunas situaciones sorprendentes al escribir tu propio código o probar ejemplos que encuentres en varias fuentes.

var height = 180;

console.log(height); // -> 180

console.log(weight); // -> Uncaught ReferenceError: weight is not defined

En el ejemplo anterior, olvidamos declarar la variable weight. El resultado es obvio: nos estamos refiriendo a una variable (es decir, estamos tratando de leer su contenido) que no existe. Algo como esto debe terminar en un error.

Hagamos un pequeño cambio:

var height = 180;

console.log(height); // -> 180

console.log(weight); // -> undefined

var weight = 70;

console.log(weight); // -> 70

Esta vez declaramos nuestra variable, pero en un lugar bastante extraño. Junto con la declaración, también realizamos la inicialización. El resultado del programa puede ser un poco sorprendente. Esta vez no hay errores. El Hoisting ha funcionado y el intérprete ha movido la declaración al comienzo del rango (en este caso, el programa).

Sin embargo, el intento de mostrar el contenido de la variable weight da dos resultados diferentes. ¿Por qué? El hositing solo se refiere a la declaración, no a la inicialización. Entonces el valor 70, que le asignamos a la variable weight, permanece en la línea donde está la declaración original. El motor de JavaScript interpreta el ejemplo anterior más o menos de la siguiente manera:

var weight;

var height = 180;

console.log(height); // -> 180

console.log(weight); // -> undefined

weight = 70;

console.log(weight); // -> 70

Desafortunadamente, el hoisting funciona un poco diferente con las declaraciones let y const.

Sin embargo, no entraremos en eso. Basta con que seas consciente del fenómeno. Y sobre todo, recordar SIEMPRE declarar las variables antes de usarlas.

**Resumen**

Usar variables, es decir, declarar, inicializar, cambiar o leer sus valores es una parte elemental de prácticamente todos los lenguajes de programación. JavaScript no es la excepción, ya que necesitas usar variables para programar en él. Recuerda declarar las variables antes de usarlas. Presta atención en dónde las declaras, ya sean locales o globales. Trata de usar las palabras clave let y const, no la palabra var. Saber esto último será útil no solo para comprender los ejemplos que se encuentran en varias fuentes, sino para que puedas evitar hacer lo mismo. Recuerda no usar los mismos nombres para diferentes variables, incluso si las declaras en diferentes rangos. Y, por supuesto, asigna nombres a las variables que estarán relacionadas con lo que deseas almacenar en ellas: el código debe ser legible no solo para el intérprete, sino también para las personas.

**Tareas**

**Tarea 1**

Juguemos a la floristería. Declara seis variables, recordando nombrarlas según su propósito:

* el precio de una sola rosa (8) y el número de rosas que tienes (70)
* el precio de un solo lirio (10) y el número de lirios que tienes (50)
* el precio de un solo tulipán (2) y la cantidad de tulipanes que tienes (120)

Ahora declara tres variables, una para cada una de las rosas, lirios y tulipanes que tienes, en las que colocas su precio total. Inserta los valores correspondientes en las variables utilizando las variables declaradas en el paso anterior. Finalmente, declara una variable en la que almacenes el precio de todas tus flores (nuevamente, usa las variables anteriores para la inicialización). Muestra toda la información del inventario en la consola de la siguiente forma:

Rosa: precio unitario: 8 , cantidad: 70 , valor: 560

Lirio: precio unitario: 10 , cantidad: 50 , valor: 500

Tulipán: precio unitario: 2 , cantidad: 120 , valor: 240

Total: 1300

Solución

// existen muchos nombres de variables posibles y correctos

let rosePrice = 8;

let lilyPrice = 10;

let tulipPrice = 2;

let numberOfRoses = 70;

let numberOfLilies = 50;

let numberOfTulips = 120;

let rosesValue = rosePrice \* numberOfRoses;

let liliesValue = lilyPrice \* numberOfLilies;

let tulipsValue = tulipPrice \* numberOfTulips;

let total = rosesValue + liliesValue + tulipsValue;

console.log("Rosa: precio unitario:", rosePrice, ", cantidad:", numberOfRoses, ", valor:", rosesValue);

console.log("Lirio: precio unitario:", lilyPrice, ", cantidad:", numberOfLilies, ", valor:", liliesValue);

console.log("Tulipán: precio unitario:", tulipPrice, ", cantidad:", numberOfTulips, ", valor:", tulipsValue);

console.log("Total: ", total);

**Tarea 2**

Modifica el código del ejemplo anterior. Supón que los precios de las flores serán constantes (no cambiarán). Declara e inicializa las variables restantes de la misma manera que en el ejemplo anterior. Muestra toda la información recopilada en la consola. Ahora disminuye el número de rosas en 20 y el de lirios en 30. Vuelve a mostrar toda la información recopilada en la consola.

Solución

const rosePrice = 8;

const lilyPrice = 10;

const tulipPrice = 2;

let numberOfRoses = 70;

let numberOfLilies = 50;

let numberOfTulips = 120;

let rosesValue = rosePrice \* numberOfRoses;

let liliesValue = lilyPrice \* numberOfLilies;

let tulipsValue = tulipPrice \* numberOfTulips;

let total = rosesValue + liliesValue + tulipsValue;

console.log("Rosa: precio unitario:", rosePrice, ", cantidad:", numberOfRoses, ", valor:", rosesValue);

console.log("Lirio: precio unitario:", lilyPrice, ", cantidad:", numberOfLilies, ", valor:", liliesValue);

console.log("Tulipán: precio unitario:", tulipPrice, ", cantidad:", numberOfTulips, ", valor:", tulipsValue);

console.log("Total: ", total);

numberOfRoses = numberOfRoses - 20;

numberOfLilies = numberOfLilies - 30;

rosesValue = rosePrice \* numberOfRoses;

liliesValue = lilyPrice \* numberOfLilies;

tulipsValue = tulipPrice \* numberOfTulips;

total = rosesValue + liliesValue + tulipsValue;

console.log("Rosa: precio unitario:", rosePrice, ", cantidad:", numberOfRoses, ", valor:", rosesValue);

console.log("Lirio: precio unitario:", lilyPrice, ", cantidad:", numberOfLilies, ", valor:", liliesValue);

console.log("Tulipán: precio unitario:", tulipPrice, ", cantidad:", numberOfTulips, ", valor:", tulipsValue);

console.log("Total: ", total);

**LABORATORIO**  
**Tiempo Estimado**

15-30 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

* variables (es decir, nombrar, declarar, inicializar y modificar su valor)

**Escenario**

Nuestra tarea será crear una lista de contactos. Inicialmente, la lista será bastante simple: solo escribiremos tres personas utilizando los datos que se muestran en la tabla a continuación. En el resto del curso, volverá a este script y lo ampliará sistemáticamente con nuevas funciones, utilizando los elementos de JavaScript recién aprendidos.

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Teléfono** | **Correo** |
| Maxwell Wright | (0191) 719 6495 | Curabitur.egestas.nunc@nonummyac.co.uk |
| Raja Villarreal | 0866 398 2895 | posuere.vulputate@sed.com |
| Helen Richards | 0800 1111 | libero@convallis.edu |

Declara e inicializa las variables donde almacenarás toda la información (nueve variables en total). Muestra en la consola información sobre el primer y último contacto en el formulario: nombre/teléfono/correo.

**Fundamentos de JavaScript 1 (JSE):**

**Modulo 2 - Sección 2**

**Tipos de datos y sus conversiones: Parte 1**

Temas en esta sección:

* Tipos de datos en JS
* Tipos de datos primitivos: Boolean (booleano)
* Tipos de datos primitivos: Number (numérico)
* Tipos de datos primitivos: BigInt (entero largo)
* Tipos de datos primitivos: String (cadena)
* Tipos de datos primitivos: undefined (indefinido)
* Tipos de datos primitivos: Symbol (símbolo)
* Tipos de datos primitivos: null (nulo)
* Conversión de tipos de datos: Función de construcción primitiva
* Conversión de tipos de datos: Conversiones primitivas
* Conversión a String (cadena)
* Conversión a Number (numérico)
* Conversión a Boolean (booleano)
* Conversión a BigInt (entero largo)
* Conversiones implicitas

**Tipos de Datos y sus Conversiones**

Los programas están destinados a procesar datos. No importa si se trata de una aplicación de tienda web, un sistema de gestión de recursos humanos o un juego de computadora, cada uno de estos programas lee, procesa y almacena grandes cantidades de datos. En el capítulo anterior, ya aprendimos cómo declarar, iniciar y modificar las variables que permiten almacenar estos datos en el contexto de un programa en ejecución. Mientras discutíamos las variables, apareció el concepto de tipos de datos y resultó que el lenguaje JavaScript, entre otras cosas, te permite cambiar el tipo de dato almacenado en una variable.

Podemos dividir los datos según sus propiedades. Por ejemplo, seguramente distinguirás intuitivamente entre datos numéricos y datos de texto. Tal clasificación es, por supuesto, arbitraria. Los números se pueden dividir en, por ejemplo, números enteros y números reales.

Distinguir los datos por sus tipos es uno de los rasgos característicos de cualquier lenguaje de programación. Cada tipo de dato está conectado con ciertas operaciones que podemos realizar sobre él. Por lo general, también existen métodos para convertir datos entre tipos seleccionados (por ejemplo, un número se puede convertir para que se guarde como una cadena).

En JavaScript, los tipos de datos se dividen en primitivos (o simples) y complejos (o compuestos). Entre los tipos primitivos podemos encontrar números y cadenas de caracteres, mientras que los tipos complejos incluyen, por ejemplo, arreglos y objetos.

La diferencia entre estos tipos de datos se encuentra precisamente en sus nombres. Los tipos primitivos, bueno, simplemente no son complejos. Si escribes datos de un tipo primitivo en una variable, se almacenará allí un valor particular. Este valor será atómico, es decir, no será posible extraer componentes de él. Los datos de tipos complejos, como un arreglo, constarán de muchos elementos de tipos primitivos (no complejos).

Por lo tanto, lógicamente nos ocuparemos primero de los tipos primitivos.

Antes de pasar a discutir los tipos de datos, debemos presentar un nuevo concepto más: literales.

Los literales son una forma de anotar valores específicos (datos) en el código del programa. Los literales se encuentran en prácticamente todos los lenguajes de programación, incluido JavaScript. Usamos literales en el capítulo anterior al inicializar variables.

Veamos un ejemplo:

let year = 1990;

console.log(year); // -> 1990

console.log(1991); // -> 1991

console.log("Alice"); // -> Alice

En este ejemplo, declaramos la variable year e inmediatamente la iniciamos con el valor 1990. Los dígitos 1990, escritos en el código en el lugar de inicialización de la variable, son un literal que representa un número . El valor 1990 se muestra en la consola utilizando la variable year. Luego mostramos en la consola el valor 1991 y "Alice", en ambos casos usando literales (que representan un número y una cadena respectivamente). En JavaScript, casi cada tipo de datos tiene su propio literal.

**El operador typeof**

Mientras aprendes sobre los tipos de datos de JavaScript, el operador typeof puede ser útil. De hecho, también es útil para el trabajo normal con este lenguaje, por lo que sería bueno que lo recordaras para más adelante. Uno de los capítulos posteriores lo dedicaremos a los **operadores**, pero en este punto es suficiente saber que un operador es un símbolo o nombre que representa alguna acción a realizar sobre los argumentos indicados. Por ejemplo, el símbolo + es un operador de dos argumentos que representa la suma.

El operador typeof que acabamos de mencionar es unario (solo toma un argumento) y nos informa del tipo de datos indicados como un argumento dado. El argumento puede ser un literal o una variable; en este último caso, se nos informará sobre el tipo de datos almacenados en él. El operador typeof devuelve una cadena con uno de los valores fijos asignados a cada uno de los tipos.

Todos los posibles valores de retorno del operador **typeof** son:

"undefined"

"object"

"boolean"

"number"

"bigint"

"string"

"symbol"

"function"

**salida**

Esta lista nos muestra aproximadamente qué tipos de datos trataremos en JavaScript.

Probemos el operador typeof usando un ejemplo simple:

let year = 1990;

console.log(typeof year); // -> number

console.log(typeof 1991); // -> number

let name = "Alice";

console.log(typeof name); // -> string

console.log(typeof "Bob"); // -> string

let typeOfYear = typeof year;

console.log(typeOfYear); // -> number

console.log(typeof typeOfYear); // -> string

Nuevamente declaramos e iniciamos la variable year. Como puedes ver, typeof tanto para el literal 1991 como para la variable que contiene un número (la inicializamos con el literal 1990) devolverá la palabra "number". Realizamos una prueba similar en las cadenas "Alice" y "Bob", usando el nombre de la variable. Además, hacemos un pequeño experimento. El resultado de typeof year se almacena en la variable denominada typeOfYear. Como puede ver, almacena el valor como un "number". Si comprobamos el tipo de esta variable, obtenemos "string". Comprueba el ejemplo tu mismo en el editor.

**Tipos de Datos Primitivos**

En JavaScript, hay seis tipos de datos primitivos (o simples): **Boolean, Number, BigInt, String, Symbol** y **undefined**. Además, el valor primitivo **null** también se trata como un tipo separado. Un dato primitivo, como ya hemos dicho, es un tipo de dato cuyos valores son atómicos. Esto significa que el valor es un elemento indivisible.

Intentemos echar un vistazo más de cerca a los datos primitivos.

**Boolean**

El Boolean (booleano) es un tipo de dato lógico. Solo puede tomar uno de dos valores: true o false. Se usa principalmente como una expresión condicional necesaria para decidir qué parte del código debe ejecutarse o cuánto tiempo debe repetirse algo (esto se denomina declaración de flujo de control y lo veremos más de cerca en el Módulo 4).

Los valores booleanos también se utilizan como lo que comúnmente se conoce como una **bandera**, una variable que señala algo que puede estar presente o ausente, habilitado o deshabilitado, etc. Como cualquier otra variable, los valores booleanos deben tener nombres claros e informativos. No es obligatorio, pero a menudo podemos ver que los nombres de las banderas booleanas tienen el prefijo "es", para mostrar la intención de comprobar si hay valores verdaderos o falsos.

let isDataValid = true;

let isStringTooLong = false;

let isGameOver = false;

continueLoop = true;

console.log(false); // -> false

console.log(typeof false); // -> boolean

console.log(isDataValid); // -> true

console.log(typeof isDataValid); // -> boolean

Podemos realizar, sin conversión (es decir, cambiar a otro tipo) operaciones lógicas en valores booleanos, algunos quizás los conozcas de las matemáticas, como NOT, AND y OR (los símbolos !, & & y || correspondientemente). Sabremos más sobre ellos en el capítulo de operadores.

**Number (numérico)**

Este es el tipo numérico principal en JavaScript que representa tanto números reales (por ejemplo, fracciones) como enteros. El formato en el que se almacenan los datos de este tipo en la memoria hace que los valores de este tipo sean a veces aproximados (especialmente, pero no únicamente, valores muy grandes o algunas fracciones). Se supone, entre otras cosas, que para garantizar la exactitud de los cálculos, los valores enteros deben limitarse en JavaScript al rango de -(253 â 1) a (253 â 1).

**Los números** permiten todas las operaciones aritméticas típicas, como suma, resta, multiplicación y división.

const year = 1991;

let delayInSeconds = 0.00016;

let area = (16 \* 3.14);

let halfArea = area / 2;

console.log(year); // -> 1991;

console.log(typeof year); // -> number;

Los números en JavaScript generalmente se presentan en forma decimal, a la que estamos acostumbrados en la vida cotidiana. Sin embargo, si un literal que describe un número está precedido por un prefijo apropiado, podemos presentarlo en hexadecimal (0xâ¦), octal (0o...) o binario (0b...). También podemos escribir números en forma exponencial, por ejemplo, en lugar de 9000, podemos escribir 9e3, y en lugar de 0.00123, podemos escribe 123e-5. Probablemente ya estés familiarizado con los términos que usamos hace un momento, como representación decimal, hexadecimal o exponencial.

let a = 10; // decimal - default

let b = 0x10; // hexadecimal

let c = 0o10; // octal

let d = 0b10; // binary

console.log(a); // -> 10

console.log(b); // -> 16

console.log(c); // -> 8

console.log(d); // -> 2

let x = 9e3;

let y = 123e-5;

console.log(x); // -> 9000

console.log(y); // -> 0.00123

Además de los números regulares en JavaScript, usamos tres valores especiales adicionales, que son: Infinity, -Infinity y NaN (no un número). Los dos primeros no requieren ninguna explicación adicional: son exactamente lo que sabemos de las matemáticas. El último, NaN, no es tanto un valor numérico como una notificación de que alguna acción aritmética (o función matemática) no se pudo realizar porque el argumento no es un número o no se puede convertir a un número.

let a = 1 / 0;

let b = -Infinity;

console.log(a); // -> Infinity

console.log(b); // -> -Infinity

console.log(typeof a); // -> number

console.log(typeof b); // -> number

let s = "definitivamente no es un numero";

let n = s \* 10;

console.log(n); // -> NaN

console.log(typeof n); // -> number

Prueba estos ejemplos e intenta cambiar los valores que aparecen en ellos tu mismo.

**BigInt**

El tipo de dato **BigInt** no se usa con demasiada frecuencia. Nos permite escribir números enteros de prácticamente cualquier longitud. Para casi todas las operaciones numéricas normales, el tipo **Number** es suficiente, pero de vez en cuando necesitamos un tipo que pueda manejar números enteros mucho más grandes.

Podemos usar operaciones matemáticas en BigInts de la misma manera que en Numbers, pero hay una diferencia al dividir. Como BigInt es un tipo entero, el resultado de la división siempre se **redondeará hacia abajo** al número entero más cercano.

Los literales BigInt son números con el sufijo **â¦n**.

let big = 1234567890000000000000n;

let big2 = 1n;

console.log(big); // -> 1234567890000000000000n

console.log(typeof big); // -> bigint

console.log(big2); // -> 1n

console.log(7n / 4n); // -> 1n

No puedes usar otros tipos de datos en operaciones aritméticas con BigInts, es decir, no puedes agregar un BigInt y un Number entre sí (esto generará un error).

let big3 = 1000n + 20;

// -> Uncaught TypeError: Cannot mix BigInt and other types, use explicit conversions

BigInt no tiene su propio equivalente de los valores Infinity o NaN. En el caso del tipo Number, dichos valores aparecen al dividir entre 0 (resultado Infinito) o al intentar realizar una acción aritmética sobre un valor que no es un número (resultado NaN). En el caso del tipo BigInt, dichas acciones generarán un error.

let big4 = 1000n / 0n; // -> Uncaught RangeError: Division by zero

**String (cadenas)**

El tipo String representa una secuencia de caracteres que forman un fragmento de texto. Las operaciones comunes en los textos incluyen la concatenación, la extracción de la subcadena y la verificación de la longitud de la cadena. Las cadenas se utilizan mucho en la programación y más aún en el desarrollo web, ya que tanto HTML como gran parte del contenido de Internet es texto.

El uso más común de texto en el desarrollo web incluye:

* Enlaces y rutas a recursos.
* Tokens.
* Comprobación de formularios y entradas llenadas por el usuario.
* Generación de contenido dinámico.

Las **Cadenas**, como otros datos primitivos, son inmutables, por lo que cuando queremos cambiar incluso una letra en una cadena, en realidad, creamos una nueva cadena.

En ejemplos anteriores, ya usamos cadenas de caracteres. Usamos comillas para indicar que un texto determinado debe tratarse como una cadena (es decir, tipo String). Los literales de cadena se pueden crear utilizando comillas simples o dobles, siempre que coincidan los caracteres de comillas inicial y final.

let country = "Malawi";

let continent = 'Africa';

console.log(country); // -> Malawi

console.log(typeof country); // -> string

console.log(continent); // -> Africa

console.log(typeof continent); // -> string

Si usas comillas dobles para marcar una cadena, puedes colocar comillas simples dentro de la cadena y se tratarán como caracteres normales. Esto también funcionará en la situación opuesta (es decir, colocar comillas dobles entre comillas simples).

let message1 = "El buque 'Mars' hizo escala en el puerto.";

let message2 = 'El ciclón "Cilida" pasará cerca de Mauritius.';

console.log(message1); // -> El buque 'Mars' hizo escala en el puerto.

console.log(message2); // -> El ciclón "Cilida" pasará cerca de Mauritius.

Si deseas poner comillas simples o dobles dentro de la cadena, también puedes usar el carácter de escape: barra invertida. Una comilla precedida por \ (barra invertida) se interpretará como caracteres ordinarios que son parte de nuestra cadena, no partes de una construcción literal. La barra invertida en sí misma, si se va a tratar como un carácter ordinario (no como un carácter de control), también debe estar precedida por... un carácter de escape (es decir, una barra invertida).

let message1 = 'El buque \'Mars\' hizo escala en el puerto.';

let message2 = "El ciclón \"Cilida\" pasará cerca de Mauritius.";

console.log(message1); // -> El buque 'Mars' hizo escala en el puerto.

console.log(message2); // -> El ciclón "Cilida" pasará cerca de Mauritius.

let path = "C:\\Windows";

console.log(path); // -> C:\Windows

Intentar realizar operaciones aritméticas en valores de tipo String, como resta, multiplicación o división, por lo general terminará en un error. Más precisamente, el valor de NaN se devolverá como resultado de la acción.

¿Por qué sucede esto? Al ver los operadores aritméticos -, \* o \, el intérprete de JavaScript intenta interpretar los valores dados como números, o convertirlos en números. Entonces, si las cadenas de caracteres consisten en dígitos, la conversión automática será exitosa y obtendremos el resultado de la acción aritmética como un valor de tipo Number. Si la cadena de caracteres no puede interpretarse como un número (y convertirse), obtendremos el resultado NaN. Hablaremos más sobre las conversiones en un momento.

let path = "C:\\Windows" - "Windows";

console.log(path); // -> NaN

let test = "100" - "10";

console.log(test); // -> 90

console.log(typeof test); // -> number

La excepción es la operación de suma, que no se tratará como una operación aritmética, sino como un intento de crear una nueva cadena mediante la combinación de dos cadenas o más.

let path = "C:\\" + "Windows";

console.log(path); // -> C:\Windows

let test = "100" + "10";

console.log(test); // -> 10010

console.log(typeof test); // -> string

Un mecanismo muy conveniente que se introdujo en JavaScript en 2015 es la **interpolación de cadenas**. Te permite tratar una cadena de caracteres como una plantilla, en la que puedes colocar valores en lugares seleccionados, como los que se toman de las variables. Tal literal se crea usando acentos graves en lugar de comillas. Los lugares donde se insertan los valores están marcados con corchetes precedidos por un signo de $.

let country = "Malawi";

let continent = "Africa";

let sentence = ` ${country} se ubica en ${continent}.`;

console.log(sentence); // -> Malawi se ubica en Africa.

Intentar realizar operaciones aritméticas en valores de tipo String, como resta, multiplicación o división, por lo general terminará en un error. Más precisamente, el valor de NaN se devolverá como resultado de la acción.

¿Por qué sucede esto? Al ver los operadores aritméticos -, \* o \, el intérprete de JavaScript intenta interpretar los valores dados como números, o convertirlos en números. Entonces, si las cadenas de caracteres consisten en dígitos, la conversión automática será exitosa y obtendremos el resultado de la acción aritmética como un valor de tipo Number. Si la cadena de caracteres no puede interpretarse como un número (y convertirse), obtendremos el resultado NaN. Hablaremos más sobre las conversiones en un momento.

let path = "C:\\Windows" - "Windows";

console.log(path); // -> NaN

let test = "100" - "10";

console.log(test); // -> 90

console.log(typeof test); // -> number

La excepción es la operación de suma, que no se tratará como una operación aritmética, sino como un intento de crear una nueva cadena mediante la combinación de dos cadenas o más.

let path = "C:\\" + "Windows";

console.log(path); // -> C:\Windows

let test = "100" + "10";

console.log(test); // -> 10010

console.log(typeof test); // -> string

Un mecanismo muy conveniente que se introdujo en JavaScript en 2015 es la **interpolación de cadenas**. Te permite tratar una cadena de caracteres como una plantilla, en la que puedes colocar valores en lugares seleccionados, como los que se toman de las variables. Tal literal se crea usando acentos graves (o acentos graves) en lugar de comillas. Los lugares donde se insertan los valores están marcados con corchetes precedidos por un signo de $.

let country = "Malawi";

let continent = "Africa";

let sentence = ` ${country} se ubica en ${continent}.`;

console.log(sentence); // -> Malawi se ubica en Africa.

Puedes hacer mucho trabajo útil con datos del tipo String. Desafortunadamente, requieren dos nuevos conceptos: **métodos** (e indirectamente, objetos) y **autoboxing**. La explicación exacta de ambos conceptos va más allá del alcance de este curso, por lo que intentaremos simplificarlos un poco.

En uno de los capítulos anteriores, presentamos el concepto de función, también en una forma algo simplificada. Ahora hablemos de métodos.

Un **método** es un tipo especial de función que pertenece a un objeto. Los **objetos** son tipos de datos complejos, que pueden constar de muchos valores (almacenados en propiedades) y métodos. Si deseas llamar al método de un objeto, escribe el nombre del método después de un punto. ¿Esto te recuerda a algo? Esta es exactamente la notación que usas cuando llamas a console.log. El objeto consola tiene muchos otros métodos además del método log, como time y timeEnd (que se pueden usar para medir el tiempo).

console.time();

console.log("probar consola"); // -> probar consola

console.timeEnd(); // -> default: 0.108154296875 ms

Todos los datos de tipos primitivos como Number, BigInt, Boolean o String tienen objetos correspondientes a los que se pueden convertir. Cada uno de estos objetos tendrá métodos diseñados para un tipo de dato específico. Llegados a este punto, llegamos a otro concepto, el **autoboxing**. Si aparece un punto después de un literal que representa un tipo primitivo, o después de una variable que contiene este tipo de datos, el intérprete de JavaScript intenta tratar este valor como un objeto y no como un primitivo. Para este propósito, convierte al dato primitivo en el objeto correspondiente sobre la marcha, que tiene los métodos apropiados (es decir, realiza autoboxing). Un poco confuso, ¿no? Afortunadamente, para usar métodos, no tenemos que entenderlo exactamente, es suficiente seguir la convención dada.

Echemos un vistazo a un ejemplo:

let river = "Mekong";

let character = river.charAt(2);

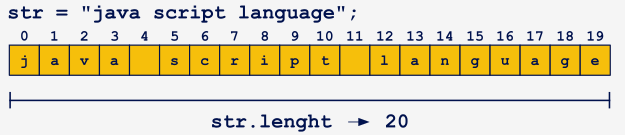
console.log(character); // -> k

En la variable river, almacenamos el dato primitivo de tipo String. En la siguiente línea, nos referimos a esta variable, escribiendo un punto después de su nombre y el nombre de uno de los métodos: charAt (un método del objeto de la clase String). Aunque el dato primitivo no contiene métodos a los que se pueda llamar, el intérprete convierte temporalmente este valor en un objeto adecuado que ya tiene dichos métodos. Uno de estos métodos es charAt, al que ahora llamamos. El método opera en una cadena colocada originalmente en la variable river y devuelve una sola letra desde la posición especificada (las letras se cuentan a partir de 0).

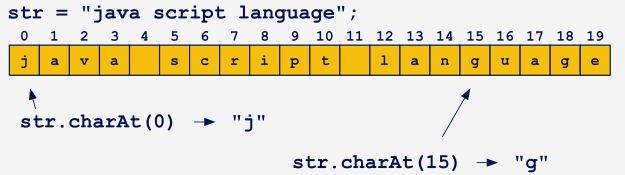
Una vez completada la operación, el intérprete elimina el objeto temporal. Entonces, desde nuestro punto de vista, parece que acabamos de llamar a un método en un tipo de dato primitivo.

Los métodos de cadena y las propiedades comúnmente utilizados (es decir, valores con nombre relacionados con el objeto) son:

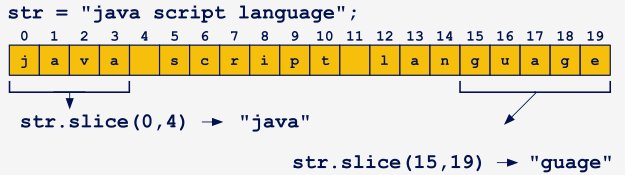
* length: propiedad que devuelve el número de caracteres en una cadena.



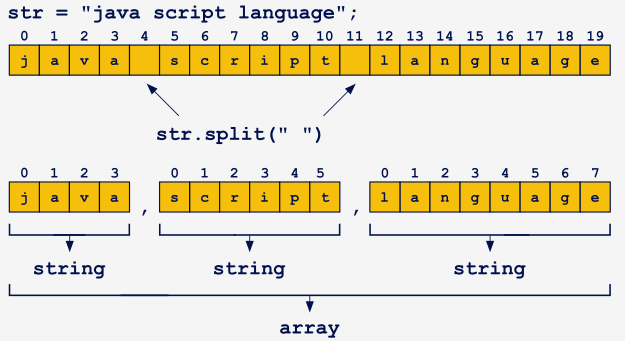
* charAt(Index): método que devuelve el carácter en la posición "Index" en las cadenas (los índices comienzan desde 0).



* slice(beginIndex, [opcional] endIndex): método que devuelve una nueva cadena que se crea a partir de los caracteres entre beginIndex (incluido) y endIndex (excluido); si se omite endIndex, entonces la nueva cadena es desde beginIndex hasta el final de la cadena.



* split(separator, [opcional] limit): método que divide la cadena en subcadenas cada vez que se encuentra un separador en esa cadena y devuelve una arreglo de esas subcadenas (diremos algunas palabras sobre arreglos en un momento), mientras que un límite opcional limit limita el número de subcadenas añadidas a la lista.



let str = "java script language";

console.log(str.length); // -> 20

console.log('test'.length); // -> 4

console.log(str.charAt(0)); // -> 'j'

console.log('abc'.charAt(1)); // -> 'b'

console.log(str.slice(0, 4)); // -> 'java'

console.log('test'.slice(1, 3)); // -> 'es'

console.log(str.split(' ')); // -> ['java', 'script', 'language']

console.log('192.168.1.1'.split('.')); // -> ['192', '168', '1', '1']

Para entenderlo correctamente, es necesario ejecutar ejemplos de datos del tipo String. No tengas miedo de experimentar cambiando los datos en los ejemplos, agregando nuevas variables o mostrando información adicional en la consola.

**Undefined**

El tipo undefined (indefinido) tiene un solo valor: undefined. Es el valor predeterminado que tienen todas las variables después de una declaración si no se les asigna ningún valor. También puedes asignar el valor undefined a cualquier variable, pero en general, esto debe evitarse, porque si necesitamos marcar una variable para que no tenga ningún valor significativo, debemos usar null.

let declaredVar;

console.log(typeof declaredVar); // -> undefined

declaredVar = 5;

console.log(typeof declaredVar); // -> number

declaredVar = undefined;

console.log(typeof declaredVar); // -> undefined

//El operador typeof también puede devolver el valor indefinido cuando se usa una variable inexistente como argumento.

console.log(typeof notDeclaredVar); // -> undefined

console.log(notDeclaredVar); // -> Uncaught ReferenceError: notDeclared is not defined

**Symbol**

El tipo de dato **Symbol** es, bueno... complicado por decir lo menos. Y, afortunadamente, no nos resulta especialmente útil.

Es un nuevo tipo primitivo que se agregó a JavaScript en 2015. No tiene ningún valor literal y solo se puede crear mediante una función de constructor especial. Los símbolos son una forma de identificador que garantiza que sea único.

Los símbolos son un tema avanzado, y para comprender su poder y utilidad, tendremos que cubrir muchos otros temas primero, así que por ahora, recuerda que el tipo de dato Symbol existe.

**null**

El valor null es bastante específico. El valor en sí es primitivo, mientras que el tipo al que pertenece no es un tipo primitivo, como Number o undefined. Esta es una categoría separada, asociada con tipos complejos, como objetos. El valor null se usa para indicar que la variable no contiene nada y, en la mayoría de los casos, es una variable que pretende contener valores de tipos complejos.

En pocas palabras, podemos suponer que el valor undefined se asigna automáticamente a las variables no inicializadas, pero si queremos indicar explícitamente que la variable no contiene nada, le asignamos un valor null. Una advertencia importante para null es que cuando se verifica con el operador typeof, devolverá "object", un resultado sorprendente. Esto es parte de un sistema de objetos mucho más complicado, pero por ahora, solo necesita saber que typeof null es igual a "object".

let someResource;

console.log(someResource); // -> undefined

console.log(typeof someResource); // -> undefined

someResource = null;

console.log(someResource); // -> null

console.log(typeof someResource); // -> object

En este curso, sin embargo, aparte de menciones menores, no estaremos aprendiendo un concepto conocido como programación orientada a objetos, y por lo tanto, usar el valor null no será tan importante para nosotros por el momento.

**Conversiones de Tipo de Datos**

**Funciones de construcción primitivas**

Usar literales no es la única manera de crear variables de los tipos primitivos dados. La segunda opción es crearlos usando funciones del tipo **constructor**. Este tipo de funciones se utilizan principalmente en JavaScript para la programación orientada a objetos, que está fuera del alcance de nuestro curso. Sin embargo, estas pocas funciones constructoras enumeradas también se pueden usar para crear datos primitivos, no solo objetos (esta no es una característica general, sino solo para las funciones enumeradas). Las siguientes funciones devolverán datos primitivos de un tipo determinado: Boolean, Number, BigInt y String.

La mayoría de estas funciones se pueden llamar sin argumentos. En tal situación:

* La función String por defecto creará y devolverá una cadena vacía primitiva: "".
* La función Number por defecto creará y devolverá el valor 0;
* La función Boolean por defecto creará y devolverá el valor de false.

La función BigInt, a diferencia de otras funciones constructoras, requiere que le pases algún valor inicial. Puede ser un número entero que se convertirá en BigInt (ver ejemplos).

const str = String();

const num = Number();

const bool = Boolean();

console.log(str); // ->

console.log(num); // -> 0

console.log(bool); // -> false

const big1 = BigInt(42);

console.log(big1); // -> 42n

const big2 = BigInt(); // -> Uncaught TypeError: Cannot convert undefined to a BigInt

Pero crear valores predeterminados no es nada impresionante. Podemos lograr esto usando literales. Entonces, ¿para qué usamos estas funciones? Bueno, los usamos en conversiones de tipos de datos

**Conversiones**

Es una situación bastante común tener un valor de un tipo pero necesitar un valor de otro tipo. El ejemplo más simple es cuando tenemos un número, pero necesitamos agregarlo a algún texto. Las conversiones en JavaScript ocurren automáticamente en situaciones específicas, pero también se pueden usar explícitamente a través de funciones como String() o Number(). Anteriormente vimos cómo esas funciones podrían usarse para crear valores predeterminados de esos tipos, pero eso no es todo lo que pueden hacer. Esas funciones también aceptan argumentos entre paréntesis y (si es posible) los convertirán a un tipo dado.

const num = 42;

const strFromNum1 = String(num);

const strFromNum2 = String(8);

const strFromBool = String(true);

const numFromStr = Number("312");

const boolFromNumber = Boolean(0);

La mayoría de estas conversiones son sencillas, pero algunas pueden ser un poco confusas, así que analicemos cada caso de conversión primitiva. Prueba todos los ejemplos que se muestran para la conversión de tipo de dato. Intenta experimentar con tus propios valores.

**Conversión a String**

Las conversiones son las más fáciles de entender, ya que intentan cambiar directamente el valor a una cadena, y esto se puede hacer para todos los tipos de datos primitivos. Así que no hay sorpresas allí. Toma en cuenta que en el ejemplo, utilizamos la técnica descrita recientemente de la **interpolación de cadenas de caracteres**.

let str = "text";

let strStr = String(str);

console.log(`${typeof str} : ${str}`); // -> string : text

console.log(`${typeof strStr} : ${strStr}`); // -> string : text

let nr = 42;

let strNr = String(nr);

console.log(`${typeof nr} : ${nr}`); // -> number : 42

console.log(`${typeof strNr} : ${strNr}`); // -> string : 42

let bl = true;

let strBl = String(bl);

console.log(`${typeof bl} : ${bl}`); // -> boolean : true

console.log(`${typeof strBl} : ${strBl}`); // -> string : true

let bnr = 123n;

let strBnr = String(bnr);

console.log(`${typeof bnr} : ${bnr}`); // -> bigint : 123

console.log(`${typeof strBnr} : ${strBnr}`); // -> string : 123

let un = undefined;

let strUn = String(un);

console.log(`${typeof un} : ${un}`); // -> undefined : undefined

console.log(`${typeof strUn} : ${strUn}`); // -> string : undefined

let n = null;

let strN = String(n);

console.log(`${typeof n} : ${n}`); // -> object : null

console.log(`${typeof strN} : ${strN}`); // -> string : null

**Conversión a Number**

La conversión a un número no es tan obvia como la conversión a una cadena. Funciona como se esperaba para cadenas que representan números reales, como "14", "-72.134", o cadenas que representan números en notación científica, como "2e3 ", o valores numéricos especiales como "NaN" o "Infinity".

Sin embargo, la cadena también puede contener números en formato hexadecimal, octal y binario. Deben estar precedidos por 0x, 0o o 0b respectivamente. Para cualquier cadena que no se pueda convertir a un valor especial, se devuelve NaN (no un número). Un BigInt también se puede convertir en un Number, pero debemos recordar que un BigInt puede almacenar valores mucho más grandes que un Number, por lo que para valores grandes, parte de ellos puede ser truncado o terminar siendo impreciso. El Boolean true se convierte en 1 y false en 0; esto es común para muchos lenguajes de programación. Un intento de convertir un valor undefined dará como resultado NaN, mientras que null se convertirá en 0.

console.log(Number(42)); // -> 42

console.log(Number("11")); // -> 11

console.log(Number("0x11")); // -> 17

console.log(Number("0o11")); // -> 9

console.log(Number("0b11")); // -> 3

console.log(Number("12e3")); // -> 12000

console.log(Number("Infinity"));// -> Infinity

console.log(Number("text")); // -> NaN

console.log(Number(14n)); // -> 14

console.log(Number(123456789123456789123n)); // - > 123456789123

456800000

console.log(Number(true)); // -> 1

console.log(Number(false)); // -> 0

console.log(Number(undefined)); // -> NaN

console.log(Number(null));// -> 0

**Conversión a Boolean**

Las conversiones a Boolean siguen reglas simples, ya que solo podemos tener uno de dos valores: true o false. El valor false siempre se devuelve para:

* 0,
* NaN,
* cadena vacía,
* undefined,
* null

Cualquier otro valor dará como resultado que se devuelva true.

console.log(Boolean(true)); // -> true

console.log(Boolean(42)); // -> true

console.log(Boolean(0)); // -> false

console.log(Boolean(NaN)); // -> false

console.log(Boolean("texto")); // -> true

console.log(Boolean("")); // -> false

console.log(Boolean(undefined)); // -> false

console.log(Boolean(null)); // -> false

**Conversión a BigInt**

Para que las conversiones a BigInt tengan éxito, necesitamos un número o una cadena que represente un número como un valor a convertir. Los valores para la conversión se pueden proporcionar en forma decimal predeterminada, así como en forma hexadecimal, octal o binaria. Esto se aplica tanto a la situación en la que damos los literales Number y String como argumentos (o variables que contienen datos de esos tipos). También podemos usar la notación exponencial, pero solo para argumentos Number. A diferencia de otras conversiones, la conversión a BigInt generará un error y detendrá el programa cuando no pueda convertir un valor dado.

**Nota:** Al probar el siguiente ejemplo, presta atención al hecho de que el primer error impide la ejecución de más código. Así que ejecuta el ejemplo varias veces seguidas, eliminando las llamadas incorrectas una por una.

console.log(BigInt(11)); // -> 11n

console.log(BigInt(0x11)); // -> 17n

console.log(BigInt(11e2)); // -> 1100n

console.log(BigInt(true)); // -> 1n

console.log(BigInt("11")); // -> 11n

console.log(BigInt("0x11")); // -> 17n

console.log(BigInt(null)); // -> Uncaught TypeError: Cannot convert null to a BigInt

console.log(BigInt(undefined)); // -> Uncaught TypeError: Cannot convert undefined to a BigInt

console.log(BigInt(NaN)); // -> Uncaught RangeError: The number NaN cannot be converted to a BigInt because it is not an integer

**Conversiones Implícitas**

Las conversiones también pueden ocurrir automáticamente y ocurren todo el tiempo. Este ejemplo simple lo demostrará (probamos un ejemplo similar cuando discutimos el tipo de dato String):

const str1 = 42 + "1";

console.log(str1); // -> 421

console.log(typeof str1); // -> string

const str2 = 42 - "1";

console.log(str2); // -> 41

console.log(typeof str2); // -> number

Entonces, ¿qué está pasando? Los detalles se mostrarán en el capítulo sobre operadores, pero la respuesta corta es que cuando intentamos realizar una suma cuando uno de los argumentos es una cadena, JavaScript también convertirá el resto de los argumentos en una cadena. Esto es lo que sucede con str1 en el ejemplo. Sin embargo, la resta con una cadena no tiene mucho sentido, por lo que, en ese caso, JavaScript convierte todo a Number.

**Tareas**

**Datos Primitivos**

* Escribe un fragmento de código que creará variables y las inicializará con valores Boolean, Number, BigInt, String y tipos undefined utilizando (siempre que sea posible) literales y funciones constructoras.
* Imprime todos los valores y todos los tipos de esos valores usando console.log. Intenta usar la interpolación de cadenas para mostrar el valor y el tipo al mismo tiempo con una sola llamada a console.log, por ejemplo, en el siguiente formato: 1000 [número].
* Realiza una cadena de conversiones: crea un Boolean a partir de un BigInt creado a partir de un Number que se creó a partir de un String. Comienza con el valor "1234". ¿Es posible?
* Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos de datos primitivos.
* Intenta sumar dos valores de diferentes tipos y verifica los resultados.
* Intenta modificar la línea const str1 = 42 + "1"; para obtener el resultado 43 (sin eliminar las comillas alrededor del 1)

**Tareas**

**Tarea 1**

Escribe un código que cree variables y las inicialice con valores Boolean, Number, BigInt, String y tipos undefined usando (cuando sea posible) literales y funciones constructoras.

Solución

let b1 = true;

let b2 = Boolean(true);

let n1 = 100;

let n2 = Number(200);

let bi1 = 100n;

let bi2 = BigInt(200);

let s1 = "Hello";

let s2 = String("Hello");

let u1 = undefined;

**Tarea 2**

Imprime todos los valores y todos los tipos de esos valores usando console.log. Intenta usar la interpolación de cadenas para mostrar el valor y el tipo al mismo tiempo con una sola llamada a console.log.

Solución

console.log(`${b1} [${typeof b1}]`);

console.log(`${b2} [${typeof b2}]`);

console.log(`${n1} [${typeof n1}]`);

console.log(`${n2} [${typeof n2}]`);

console.log(`${bi1} [${typeof bi1}]`);

console.log(`${bi2} [${typeof bi2}]`);

console.log(`${s1} [${typeof s1}]`);

console.log(`${s2} [${typeof s2}]`);

console.log(`${u1} [${typeof u1}]`);

**Tarea 3**

Realizar una cadena de conversiones: crear un Boolean a partir de un BigInt creado a partir de un Number que se creó a partir de un String . Comienza con el valor "1234". ¿Es posible?

Ejemplo

let b = Boolean( BigInt(Number("1234")));

console.log(`${b} [${typeof b}]`);

// or

let s = "1234";

let n = Number(s);

let bi = BigInt(n);

let b = Boolean(bi);

console.log(`${b} [${typeof b}]`);

**Tarea 4**

Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos primitivos.

Ejemplo

let b = true + false;

let n = 100 + 200;

let bi = 100n + 200n;

let s = "He" + "llo";

let u = undefined + undefined;

console.log(`${b} [${typeof b}]`); // !!! number

console.log(`${n} [${typeof n}]`);

console.log(`${bi} [${typeof bi}]`);

console.log(`${s} [${typeof s}]`);

console.log(`${u} [${typeof u}]`); // !!! number

**Tarea 5**

Prueba sumar dos valores de diferentes tipos y verifica los resultados.

Ejemplos

let b1 = true + 100;

// let b2 = true + 100n; // -> error!

let b3 = true + "100";

// let n1 = 100 + 200n; // -> error!

let n2 = 100 + true;

let n3 = 100 + "200";

// let bi1 = 100n + 200; // -> error!

// let bi2 = 100n + true // -> error!

let bi3 = 100n + "200";

let s1 = "100" + 200;

let s2 = "100" + 200n;

let s3 = "100" + true;

let s4 = "abc" + 200;

let s5 = "abc" + 200n;

let s6 = "abc" + true;

console.log(`${b1} [${typeof b1}]`); // -> 101 [number]

// console.log(`${b2} [${typeof b2}]`);

console.log(`${b3} [${typeof b3}]`); // -> true100 [string]

// console.log(`${n1} [${typeof n1}]`);

console.log(`${n2} [${typeof n2}]`); // -> 101 [number]

console.log(`${n3} [${typeof n3}]`); // -> 100200 [string]

// console.log(`${bi1} [${typeof bi1}]`);

// console.log(`${bi2} [${typeof bi2}]`);

console.log(`${bi3} [${typeof bi3}]`); // -> 100200 [string]

console.log(`${s1} [${typeof s1}]`); // -> 100200 [string]

console.log(`${s2} [${typeof s2}]`); // -> 100200 [string]

console.log(`${s3} [${typeof s3}]`); // -> 100true [string]

console.log(`${s4} [${typeof s4}]`); // -> abc200 [string]

console.log(`${s5} [${typeof s5}]`); // -> abc200 [string]

console.log(`${s6} [${typeof s6}]`); // -> abctrue [string]

**Tarea 6**

Intenta modificar la línea const str1 = 42 + "1"; para obtener el resultado 43 (sin eliminar las comillas alrededor del 1 ).

Ejemplo

const str1 = 42 + +"1";

**Fundamentos de JavaScript 1 (JSE):  
Módulo 2**

**Sección 3**

**Tipos de datos y su conversión: Parte 2**

Temas en esta sección:

* Tipos de datos complejos: objetos
* Tipos de datos complejos: arreglos
* Arreglos: la propiedad length
* Arreglos: el método indexOf
* Arreglos: el método push
* Arreglos: el método unshift
* Arreglos: el método pop
* Arreglos: el método shift
* Arreglos: el método reverse
* Arreglos: el método slice
* Arreglos: el método concat

**Tipos de datos complejos**

Limitaremos la discusión de tipos complejos a solo dos de ellos: objetos y arreglos. Desafortunadamente, incluso estos tipos deberán presentarse de manera simplificada. Esto debería ser suficiente para usarlos en su ámbito básico, pero las técnicas más avanzadas relacionadas con ellos, así como otros tipos complejos, se presentarán en las siguientes partes del curso.

**Objeto**

Los objetos tienen muchas aplicaciones en JavaScript. Una de las más básicas, y la única que usaremos ahora, es utilizarlo como una estructura conocida en informática como registro. Un registro es una colección de campos con nombre. Cada campo tiene su propio nombre (o clave) y un valor asignado. En el caso de los objetos de JavaScript, estos campos suelen denominarse propiedades. Los registros, o en nuestro caso, los objetos, te permiten almacenar múltiples valores de diferentes tipos en un solo lugar. En JavaScript, hay algunas formas de crear objetos, pero la más fácil y rápida es usar el literal de llaves {}.

let testObj = {};

console.log(typeof testObj); // -> object

El objeto que creamos y almacenamos en la variable testObj no es particularmente útil porque está... vacío. No hemos definido ningún campo en él, es decir, ningún par clave-valor. Intentémoslo de nuevo, esta vez definiendo un objeto que contenga dos campos con las claves nr y str.

let testObj = {

nr: 600,

str: "texto"

};

Toma en cuenta que hemos creado objetos usando el mismo literal, pero al mismo tiempo hemos creado propiedades que son pares clave-valor. Las propiedades están separadas por comas. Posteriormente, se puede hacer referencia a una propiedad (campo) específica de un objeto con notación de puntos. Esta notación requiere que el nombre del objeto (un literal o el nombre de una variable que contiene el objeto) sea seguido por un punto, seguido por el nombre del campo (clave) nuevamente.

console.log(testObj.nr); // -> 600

console.log(testObj.str); // -> texto

¿Para qué necesitamos objetos? La razón más simple para usarlos puede ser el deseo de almacenar varios valores en un solo lugar, que están vinculados entre sí por alguna razón.

Supongamos que recopilamos información sobre los usuarios de nuestro sistema. La información sobre un solo usuario consistirá en su nombre, apellido, edad y dirección de correo electrónico. Intentemos escribir un fragmento de código apropiado para dos usuarios, sin usar objetos por ahora.

let name1 = "Calvin";

let surname1 = "Hart";

let age1 = 66;

let email1 = "CalvinMHart@teleworm.us";

let name2 = "Mateus";

let surname2 = "Pinto";

let age2 = 21;

let email2 = "MateusPinto@dayrep.com";

Parece que todo funciona correctamente, pero si lo pensamos bien, notaremos dos inconvenientes. En primer lugar, para cada usuario, deberás crear nombres de variables separados para el apellido, el correo electrónico, etc. ¿Qué sucede si describimos a cada usuario con un poco más de precisión? ¿O si no fueran solo dos usuarios, sino, digamos, mil? Entonces sería menos inconveniente. Hasta cierto punto, podemos arreglarlo con objetos. El segundo problema es que ya en la etapa de escritura, necesitamos saber el número exacto de usuarios que se describirán en el sistema. Esto sería extremadamente limitante en aplicaciones reales y sería mejor poder agregarlas dinámicamente. También podremos mejorar esto, no con objetos, sino con arreglos (más sobre esto en un momento).

Así que mejoremos nuestro de código con objetos:

let user1 = {

name: "Calvin",

surname: "Hart",

age: 66,

email: "CalvinMHart@teleworm.us"

};

let user2 = {

name: "Mateus",

surname: "Pinto",

age: 21,

email: "MateusPinto@dayrep.com"

};

Todavía tenemos que dar nombres diferentes a las variables que almacenan información (en forma de objetos) sobre usuarios individuales, pero esta vez las propiedades pueden tener los mismos nombres. Esto hace que el código no solo sea más claro y coherente, sino que también facilita la realización de acciones en las propiedades de diferentes usuarios.

Las propiedades de un objeto, como hemos indicado anteriormente, se ponen a disposición con un punto y un nombre clave. Podemos tanto leer como modificar el valor asociado a una clave en particular. Además, también podemos modificar todo el objeto añadiendo una nueva propiedad que antes no existía. También hacemos esto usando la notación de puntos: si durante un intento de modificar la propiedad, el intérprete no encuentra la clave que especificamos, la creará.

console.log(user1.name); // -> Calvin

console.log(user2.name); // -> Mateus

console.log(user1.age); // -> 66

user1.age = 67;

console.log(user1.age); // -> 67

console.log(user2.phone); // -> undefined

user2.phone = "904-399-7557";

console.log(user2.phone); // -> 904-399-7557

Si puedes agregar nuevos campos a un objeto existente, ¿también puedes eliminarlos? Por supuesto que puedes: el operador de eliminación se usa para esto.

console.log(user2.phone); // -> 904-399-7557

delete user2.phone;

console.log(user2.phone); // -> undefined

La usabilidad de los objetos va mucho más allá de usarlos como estructuras de almacenamiento de datos. Sin embargo, es un tema aparte, en gran medida relacionado con la programación orientada a objetos, que forma parte de técnicas de programación más avanzadas. En nuestro caso, los objetos serán estructuras simples, formadas por pares de clave-valor.

**Arreglos**

Un **arreglo**, como un objeto, es un tipo de datos complejo que se puede usar para almacenar una colección de datos. Similar a un objeto, los datos almacenados (los valores) pueden ser de cualquier tipo. La diferencia entre estas estructuras es que en un arreglo solo almacenamos valores, sin los nombres asociados (es decir, las claves).

Entonces, ¿cómo sabemos a qué elemento del arreglo nos referimos si no podemos señalarlo por su nombre? Lo sabemos porque los elementos del arreglo están ordenados (pero no necesariamente clasificados) y ocupan posiciones consecutivas y numeradas dentro de el. El número del campo donde se encuentra un valor particular en el arreglo se denomina índice o posición. El índice comienza desde 0.

La forma más sencilla de crear arreglos en JavaScript es usar corchetes (es un literal de arreglo). De esta forma, podemos crear tanto un arreglo vacío, en el que se insertarán los elementos más tarde, como un arreglo que contenga algunos elementos iniciales (que estarán separados por comas). Al referirnos a un elemento del arreglo en particular, usamos la notación de corchetes: después del nombre de la variable del arreglo, escribimos un corchete, en el que ponemos el índice del elemento que nos interesa.

Veamos un ejemplo sencillo:

let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

console.log(days[0]); // -> Sun

console.log(days[2]); // -> Tue

console.log(days[5]); // -> Fri

days[0] = "Sunday";

console.log(days[0]); // -> Sunday

let emptyArray = [];

console.log(emptyArray[0]); // -> undefined

Para comenzar, hemos declarado e iniciado el arreglo days, que contiene siete nombres abreviados de los días de la semana. Los elementos de este arreglo son datos del tipo String. Teniendo en cuenta que los índices (las posiciones de los elementos) en el arreglo comienzan desde 0, mostramos tres días de la semana seleccionados en la consola. Luego cambiamos el elemento en el índice 0, y el valor "Sun" se reemplaza por "Sunday". El último fragmento de código es una declaración de un arreglo vacío y un intento de leer un elemento inexistente en el arreglo.

¿Cómo podemos agregar un nuevo elemento a un arreglo existente, por ejemplo, uno vacío?

La forma más sencilla sería asignar un nuevo valor a una posición específica utilizando la notación de corchetes. Para el intérprete, no importa si ya hay algo en este índice o no. Simplemente coloca un nuevo valor allí. Lo interesante es que no tenemos que llenar el arreglo con elementos uno por uno; puedes dejar espacios vacíos en el arreglo.

let animals = [];

console.log(animals[0]); // -> undefined

animals[0] = "dog";

animals[2] = "cat";

console.log(animals[0]); // -> dog

console.log(animals[1]); // -> undefined

console.log(animals[2]); // -> cat

En el ejemplo, declaramos un arreglo vacío llamado animals. Luego colocamos dos elementos, "dog" y "cat", en las posiciones 0 y 2, dejando la posición 1 vacía. Sin embargo, esta no es la única forma de agregar nuevos elementos en el arreglo, y presentaremos otros en un momento, así como formas de eliminarlos.

Por lo general, almacenamos el mismo tipo de dato en un arreglo, pero como mencionamos anteriormente, JavaScript no lo requiere. Entonces podemos crear fácilmente un arreglo que contenga elementos de diferentes tipos.

let values = ["Test", 7, 12.3, false];

Como ya hemos dicho, un elemento del arreglo puede ser de cualquier tipo. Lo que es interesante es el hecho de que también podemos almacenar arreglos como elementos de un arreglo, y podemos acceder a los elementos de este arreglo anidado usando múltiples corchetes.

let names = [["Olivia", "Emma", "Mia", "Sofia"], ["William", "James", "Daniel"]];

console.log(names[0]); // -> ["Olivia", "Emma", "Mia", "Sofia"]

console.log(names[0][1]); // -> Emma

console.log(names[1][1]); // -> James

let femaleNames = names[0];

console.log(femaleNames[0]); // -> Olivia

console.log(femaleNames[2]); // -> Mia

El ejemplo muestra una declaración de un arreglo que contiene otros dos arreglos como sus componentes. Toma en cuenta que los arreglos internos no tienen que tener la misma longitud (en muchos otros lenguajes de programación, esto es obligatorio).

**¿Para qué pueden ser útiles los arreglos en la práctica?**

Son principalmente una forma conveniente de almacenar una colección de elementos bajo un nombre. Además, es muy importante que podamos agregar nuevos elementos a un arreglo mientras el programa se está ejecutando.

¿Recuerdas el ejemplo con los usuarios del sistema que probamos mientras discutíamos los objetos? Una de las desventajas de la solución presentada fue la necesidad de declarar variables para todos los usuarios, por lo que en la etapa de escritura del programa teníamos que saber el número de usuarios. Usando un arreglo, podemos agregar nuevos usuarios mientras se ejecuta el programa. Mencionamos varias veces que los elementos de un arreglo pueden ser cualquier dato, incluidos los objetos. Como recordatorio, repitamos el ejemplo en el que declaramos dos variables del tipo objeto, user1 y user2, que contienen información sobre dos usuarios del sistema:

let user1 = {

name: "Calvin",

surname: "Hart",

age: 66,

email: "CalvinMHart@teleworm.us"

};

let user2 = {

name: "Mateus",

surname: "Pinto",

age: 21,

email: "MateusPinto@dayrep.com"

};

Pongamos información sobre estos dos usuarios en el arreglo de usuarios e intentemos mostrar alguna información como parte de la prueba:

let users =[

{

name: "Calvin",

surname: "Hart",

age: 66,

email: "CalvinMHart@teleworm.us"

},

{

name: "Mateus",

surname: "Pinto",

age: 21,

email: "MateusPinto@dayrep.com"

}

];

console.log(users[0].name); // -> Calvin

console.log(users[1].age); // -> 21

Intentemos agregar un nuevo usuario en el arreglo. Lo haremos de la única forma que conocemos hasta ahora, que es asignando un nuevo elemento a un índice libre (esto es una continuación del ejemplo anterior).

users[2] = {

name: "Irene",

surname: "Purnell",

age: 32,

email: "IreneHPurnell@rhyta.com"

}

console.log(users[0].name); // -> Calvin

console.log(users[1].name); // -> Mateus

console.log(users[2].name); // -> Irene

Durante la operación del programa, es posible interactuar con el usuario, por ejemplo, para recuperar el arreglo con elementos que no conocíamos mientras escribíamos el programa.

Ahora hagamos un pequeño experimento y apliquemos el operador typeof a la variable que contiene el arreglo. El resultado puede ser algo sorprendente:

let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

console.log(typeof days); // -> object

Para hablar en general, en JavaScript, todo excepto los datos primitivos son objetos. Los **arreglos** también se tratan como un tipo especial de objeto. El operador typeof no distingue entre tipos de objetos (o más precisamente, clases), por lo que nos informa que la variable days contiene un objeto. Si queremos asegurarnos de que la variable contiene un arreglo, podemos hacerlo usando el operador instanceof, entre otros. Está estrechamente relacionado con la programación orientada a objetos, de la que no hablaremos en este curso, pero por el momento solo necesitamos saber cómo usarlo en esta situación específica.

let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

let day = "Sunday";

console.log(typeof days); // -> object

console.log(typeof day); // -> string

console.log(days instanceof Array); // -> true

console.log(day instanceof Array); // -> false

El operador instanceof es un operador de dos argumentos, que requiere que se especifique la variable probada (o literal) y la clase de objeto. En nuestro caso, la clase Array. El operador devuelve verdadero (true) o falso (false), según el resultado de la prueba.

Recientemente presentamos los conceptos de **método** y **propiedad**. Aparecieron cuando hablábamos del tipo String. Estas eran funciones y valores relacionados con un objeto específico. Ahora resulta que un arreglo se implementa como un objeto en JavaScript, por lo que probablemente también tenga sus métodos y propiedades. Y de hecho los tiene. Hay muchos métodos muy útiles que nos ayudan a trabajar con arreglos, como combinar arreglos, cortar elementos, ordenar o filtrar.

Solo veremos algunos de ellos ahora, porque muchos otros requieren que podamos crear nuestras propias funciones. Volveremos sobre algunos de ellos en el apartado dedicado a las funciones.

**length**

La propiedad length se utiliza para obtener información sobre la longitud (la cantidad de elementos) del arreglo (incluidas las posiciones vacías entre los elementos existentes).

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names.length); // -> 4

names[5] = "Amelia";

console.log(names.length); // -> 6

console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel", undefined, "Amelia"]

console.log(names[3]); // -> Samuel

console.log(names[4]); // -> undefined

console.log(names[5]); // -> Amelia

**indexOf**

El método indexOf se utiliza para buscar en el arreglo y localizar un valor dado. Si se encuentra el valor (el elemento está en el arreglo), se devolverá su índice (posición). El método devuelve -1 si no se encuentra el elemento. Si hay más de un elemento con el mismo valor en el arreglo, se devuelve el índice del primer elemento encontrado.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names.indexOf("Mateo")); // -> 2

console.log(names.indexOf("Victor")); // -> -1

**indexOf**

El método indexOf se utiliza para buscar en el arreglo y localizar un valor dado. Si se encuentra el valor (el elemento está en el arreglo), se devolverá su índice (posición). El método devuelve -1 si no se encuentra el elemento. Si hay más de un elemento con el mismo valor en el arreglo, se devuelve el índice del primer elemento encontrado.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names.indexOf("Mateo")); // -> 2

console.log(names.indexOf("Victor")); // -> -1

**unshift**

El método unshift funciona de manera similar a push, con la diferencia de que se agrega un nuevo elemento al inicio del arreglo. La longitud de arreglo aumenta en 1, todos los elementos antiguos se mueven hacia la derecha y el nuevo elemento se coloca en el espacio vacío que se ha creado al inicio del arreglo. El índice del nuevo elemento es 0.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names);

console.log(names.indexOf("Mateo")); // -> 2

console.log(names.indexOf("Victor")); // -> -1 -> No existe

names.unshift("Amelia");

console.log(names.indexOf("Amelia")); // -> 0

console.log(names);

**pop**

El método pop te permite eliminar el último elemento del arreglo. Como resultado de su ejecución, se devuelve el elemento con mayor índice, mientras que al mismo tiempo se elimina del arreglo original. La longitud del arreglo obviamente se reduce en 1.

let names= ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names.length); // -> 4

let name = names.pop();

console.log(names.length); // -> 3

console.log(name); // -> Samuel

console.log(names); // -> ["Olivia", "Emma", "Mateo"]

**shift**

El método shift funciona de manera similar a pop, solo que esta vez eliminamos el elemento del inicio del arreglo (con el índice 0). El método devuelve el elemento eliminado y todos los demás elementos se desplazan hacia la izquierda, llenando el espacio vacío. La longitud del arreglo original se reduce en 1.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

console.log(names.length); // -> 4

let name = names.shift();

console.log(names.length); // -> 3

console.log(name); // -> Olivia

console.log(names); // -> ["Emma", "Mateo", "Samuel"]

**reverse**

El método reverse invierte el orden de los elementos de un arreglo. El método devuelve un arreglo con los elementos en orden inverso.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

names.reverse();

console.log(names); // -> ["Samuel", "Mateo", "Emma", "Olivia"]

**slice**

El método slice te permite crear un nuevo arreglo a partir de elementos seleccionados del arreglo original. Llamar al método no afecta el arreglo original. El método toma uno o dos valores enteros como argumentos.

Las combinaciones básicas son:

* Un argumento mayor que cero: se copian todos los elementos del índice dado como argumento hasta el final del arreglo.
* Dos argumentos mayores que cero: se copia el elemento del índice especificado como primer argumento al elemento especificado como segundo argumento.
* Dos argumentos, el primero positivo, el segundo negativo: se copian todos los elementos desde el índice especificado hasta el final del arreglo, excepto el número especificado de los últimos elementos (por ejemplo, el argumento -3 significa que no copiamos los últimos tres elementos).
* Un argumento negativo: el número especificado de los últimos elementos se copian al final del arreglo (por ejemplo, -2 significa que se copian los dos últimos elementos).

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

let n1 = names.slice(2);

console.log(n1); // -> ["Mateo", "Samuel"]

let n2 = names.slice(1,3);

console.log(n2); // -> ["Emma", "Mateo"]

let n3 = names.slice(0, -1);

console.log(n3); // -> ["Olivia", "Emma", "Mateo"]

let n4 = names.slice(-1);

console.log(n4); // -> ["Samuel"]

console.log(names); // -> ["Olivia", "Emma", "Mateo","Samuel"]

**concat**

El método concat crea un nuevo arreglo adjuntando elementos del arreglo dado como argumento a los elementos originales del arreglo. El método no cambia ni el arreglo original ni el arreglo especificado como argumento.

let names = ["Olivia", "Emma", "Mateo", "Samuel"];

let otherNames = ["William", "Jane"];

let allNames = names.concat( otherNames);

console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel"]

console.log(otherNames); // -> ["William", "Jane"]

console.log(allNames); // -> ["Olivia", "Emma", "Mateo","Samuel", "William", "Jane"]

Hasta ahora, los arreglos son el elemento de programación más complicado que hemos aprendido. Así que no sería de extrañar que tuvieras que pasar por esta parte del curso una vez más. Los arreglos son muy importantes, por lo que vale la pena tomarse un poco más de tiempo para entenderlos bien. Volveremos a los arreglos más de una vez, entre otras cosas cuando hablemos de bucles y funciones.

**Resumen**

Este capítulo contiene bastante información. Comenzamos el capítulo con una discusión sobre **tipos de datos simples**, que aparte del tipo String, en realidad no debería causar ningún problema. Los tipos Number, BigInt o Boolean no se llaman por casualidad **primitivos**.

Mientras discutíamos el tipo String, aprendimos lo qué es **autoboxing** (conversión automática de un primitivo a un objeto relacionado con ese primitivo) y cómo podemos usar **métodos** relacionados con el objeto String.

Algún espacio se dedicó a discutir la **conversión de tipos de datos**. Al final vimos información básica sobre **tipos complejos**, limitándonos a objetos y arreglos. Recuerda que introdujimos los objetos en una forma muy simplificada, lo que nos permitirá usarlos como **registros** (es decir, tipos de datos que consisten en campos clave-valor). Lo que discutimos está relacionado con la programación orientada a objetos, de la que puedes haber oído hablar, pero que no forma parte del curso actual.

Los **arreglos** se han discutido con un poco más de detalle. Volveremos sobre ellos más de una vez, porque son uno de los elementos básicos utilizados en la práctica. Los ampliaremos en la parte de bucles y funciones del curso.

**Tareas**

**Objetos**

**Tarea 1**

Crea un objeto que describa un boleto de tren y guárdalo en la variable ticket. El objeto debe tener tres campos:

* estación inicial (el nombre clave es from, y como valor, proporciona el nombre de la estación más cercana en tu área)
* estación final (el nombre clave es to, y como valor, dar cualquier otra estación dentro de 100 km)
* el precio del boleto (el nombre clave es price, y como valor, proporciona la cantidad que te gustaría pagar por este boleto)

El objeto debe crearse usando llaves {}, en los que todos los campos se enumerarán inmediatamente. Luego muestra los valores de todos los campos del ticket en la consola.

Ejemplo

let ticket = {

from: "Berlin",

to: "Potsdam",

price: 11

};

console.log(`Ticket from: ${ticket.from}`);

console.log(`Ticket to: ${ticket.to}`);

console.log(`Ticket price: ${ticket.price}`);

**Task 2**

Declara un objeto vacío y guárdalo en la variable person. Usando la notación de punto, agrega los campos name y surname al objeto ingresando tus datos como valores. Intenta mostrar los campos individuales en la consola.

Ejemplo

let person = {};

person.name = "Mary";

person.surname = "Stuart";

console.log(`${person.name} ${person.surname}`);

**Arreglos**

**Tarea 3**

Estamos creando una pequeña biblioteca de libros sobre programación en JavaScript. Tenemos tres libros y queremos preparar una lista de ellos. Almacenaremos tres datos de cada libro: el título, el autor y el número de páginas:

* + *Speaking JavaScript*, Axel Rauschmayer, 460;
  + *Programming JavaScript Applications*, Eric Elliott, 254;
  + *Understanding ECMAScript 6*, Nicholas C. Zakas, 352.
* Crea un arreglo de tres objetos que representen los libros. Cada objeto debe tener las siguientes propiedades: title, author, pages

Ejemplo

let books = [{

title: "Speaking JavaScript",

author: "Axel Rauschmayer",

pages: 460

},

{

title: "Programming JavaScript Applications",

author: "Eric Elliot",

pages: 254

},

{

title: "Understanding ECMAScript 6",

author: "Nicholas C. Zakas",

pages: 352

}

];

**Tarea 4**

Agregar un nuevo libro a la colección: *Learning JavaScript Design Patterns*, por Addy Osmani, 254 páginas. Usa el método apropiado para adjunta el libro al final del arreglo. Muestra la longitud del arreglo y, a su vez, todos los nombres de los libros en la colección.

Ejemplo

let newBook = {

title: "Learning JavaScript Design Patterns",

author: "Addy Osmani",

pages: 254

};

books.push(newBook);

console.log(books.length);

console.log(books[0].title);

console.log(books[1].title);

console.log(books[2].title);

console.log(books[3].title);

**Task 5**

Utiliza el comando slice para copiar los dos últimos libros al nuevo arreglo.

Ejemplo

let selectedBooks = books.slice(-2);

**Tarea 6**

El primer libro de la colección se pierde en circunstancias inexplicables. Ya has aceptado la pérdida, así que ahora elimínalo del arreglo. ¿Cuál método usarás para este propósito? Muestra la longitud del arreglo y todos los nombres de los libros de la colección a su vez.

Ejemplo

books.shift();

console.log(books.length);

console.log(books[0].title);

console.log(books[1].title);

console.log(books[2].title);

**Tarea 7**

Muestra la suma de las páginas de todos los libros de la colección.

Ejemplo

let sum = books[0].pages + books[1].pages + books[2].pages;

console.log(`pages: ${sum}`);

**LABORATORIO**

**Tiempo estimado**

15-30 minutos

**Nivel de dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

* Las propiedades básicas de los tipos de datos complejos Array y Object (usados como un registro) y ser capaz de usarlos en la práctica.

**Escenario**

¿Recuerdas la lista de contactos que se creó mientras realizabas la tarea del laboratorio anterior? Tienes que admitir que a primera vista parecía bastante extraño. Tuvimos que usar nueve variables para almacenar información sobre solo tres usuarios. Lo que es peor, agregar cada nuevo usuario requeriría la creación de tres variables más. Esto no es conveniente ni práctico. Afortunadamente, desde entonces hemos aprendido algo sobre arreglos y objetos, lo que nos permitirá guardar nuestra lista de una manera un poco más conveniente. Con los mismos datos que en el laboratorio anterior, crea la lista de contactos como un arreglo, cada elemento del cual será un objeto que describa a un solo usuario. Entonces, deberíamos tener un arreglo de tres elementos, y cada objeto colocado en el contendrá tres piezas de información (nombre, teléfono y correo electrónico).

Al final de la lista declarada de esta manera, agrega un nuevo contacto usando el método del arreglo apropiado. El nuevo contacto es: **Maisie Haley / 0913 531 3030 / risus.Quisque@urna.ca**.

Muestra el primer y último contacto, de nuevo en el formato: **nombre / teléfono / correo electrónico**. Utiliza la propiedad length del arreglo para determinar el índice del último elemento. Recuerde que los elementos del arreglo se indexan a partir de 0.

let contacts = [{

name: "Maxwell Wright",

phone: "(0191) 719 6495",

email: "Curabitur.egestas.nunc@nonummyac.co.uk"

}, {

name: "Raja Villarreal",

phone: "0866 398 2895",

email: "posuere.vulputate@sed.com"

}, {

name: "Helen Richards",

phone: "0800 1111",

email: "libero@convallis.edu"

}];

**Fundamentos de JavaScript 1 (JSE):  
Módulo 2**

**Sección 4**

**Comentarios**

Temas en la sección:

* Comentarios de una sola línea
* Comentarios de varias líneas
* Documentación
* Alternar código

**Comentarios**

Los comentarios son algo común en la programación. "Comentar" puede no ser una técnica de programación clave (si se le puede llamar técnica), pero te permite mejorar tu trabajo con el código, entre otras cosas, haciéndolo más legible. Entonces, ¿qué son los comentarios y por qué los necesitamos?

Los comentarios son solo texto sin formato, totalmente ignorados por el intérprete de JavaScript, que generalmente sirven para explicar una determinada pieza de código, que por algunas razones puede no ser completamente legible. Sin embargo, no podemos escribirlos con total libertad, ya que el intérprete intentará tratarlos como comandos, nombres de variables o palabras clave. Entonces JavaScript necesita distinguir los comentarios del resto del código. En JavaScript, tenemos dos tipos de comentarios, y ambos se usan comúnmente en muchos lenguajes de programación, desde la familia de lenguajes C hasta Python. Se denominan comentarios de una sola línea y de varias líneas.

**Comentarios de una sola línea**

Esta es la forma principal de comentar el código. Utiliza un carácter de doble diagonal al comienzo del comentario que se extiende hasta el final de la línea. El intérprete ignorará cualquier código colocado a la derecha de la doble diagonal. Si queremos crear comentarios de esta forma en varias líneas, tenemos que poner // en cada línea por separado. Probablemente hayas notado que hemos usado este tipo en ejemplos anteriores.

// Este es un comentario de una sola línea

let x = 42; // Este también es un comentario de una sola línea, aunque la parte anterior a la doble diagonal es un código que se ejecutará.

// Esta línea y la siguiente serán ignoradas

// x = 8;

console.log(x); // -> 42

En la mayoría de los editores de código modernos, es posible "comentar" fragmentos de código usando métodos abreviados del teclado. Esto significa que podemos colocar el cursor en la línea de código seleccionada y presionando una combinación específica de teclas podemos colocar el signo de comentario al inicio de la línea. Presionar la misma combinación nuevamente "descomentará" el carácter del comentario. Por lo general, también es posible marcar varias líneas (con un mouse o usando las teclas de flecha con la tecla Shift presionada) y colocar un carácter de comentario al comienzo de todas las líneas marcadas usando la misma combinación de teclas. Este mecanismo es compatible con aplicaciones como Sublime Text o Visual Studio Code, dos editores que recomendamos anteriormente. Pruébalo en el editor de la plataforma OpenEDG que estás utilizando actualmente. Esta combinación de teclas de la que hablamos es la más habitual en Windows:

ctrl + /

mientras que en macOS:

command + /

Este método se usa con mayor frecuencia para "comentar" (es decir, deshabilitar temporalmente) un fragmento de código seleccionado, por ejemplo, para probar una versión alternativa del mismo.

**Comentarios de varias líneas**

Los comentarios de varias líneas, también conocidos como comentarios de bloque, permiten que los comentarios abarquen varias líneas, pero también te permiten colocar comentarios dentro de una línea, lo que no es posible con los comentarios de una sola línea. Se crean con una diagonal y un asterisco al principio del comentario y un asterisco y una diagonal al final.

/\*

Este es un bloque

de comentarios y puede

abarcar varias líneas

Entonces este código no se ejecutará

console.log("¡Hola, Mundo!");

\*/

let x /\* porque no hay mejor nombre \*/ = 42;

console.log(x);

**Pero, ¿por qué comentar en primer lugar?**

Como ya se dijo, es muy importante dar nombres apropiados e informativos a las variables (y también a las funciones y cualquier otra cosa que podamos nombrar), pero incluso los mejores nombres no ayudarán si el código es complejo, y el código se vuelve complejo con bastante facilidad. Entonces, para ayudar a los desarrolladores a expresar su intención para una pieza de código dada, usan comentarios para explicarse más verbalmente.

Por supuesto, estos son ejemplos exagerados de comentarios solo para mostrarte cómo usarlos. En las aplicaciones y secuencias de comandos de la vida real, los comentarios deben usarse con prudencia, ya que demasiados comentarios obvios harán que el código sea aún más difícil de leer.

Como regla general, los comentarios deben usarse cuando la lectura del código no es suficiente para comprender lo que hace, o en situaciones en las que el código se comporta de manera diferente a lo esperado y necesita demostrar que es intencional. También debes recordar que en la mayoría de los proyectos comerciales, no eres la única persona que leerá este código. E incluso si lo fueras, leer tu propio código después de unos meses es un desafío, y leer el código de otra persona puede llevar ese desafío al siguiente nivel.

// inicializar greetingText con Hola

const greetingText = "Hola";

angle = angle + 90; // girar 90 grados para compensar la pantalla vertical

// comprobar que 0 no sea el divisor

let result = a / b;

// no hay necesidad de comprobar el divisor

let result = a / b;

// dividir a entre b

let result = a / b;

**Documentación**

Podemos usar comentarios para documentar el código y escribir exactamente qué funciones hace y qué parámetros requiere. En muchos proyectos, los archivos tienen un encabezado con información sobre el autor, la licencia o el historial de cambios. Hay herramientas que pueden generar documentación automáticamente a partir de comentarios, siempre que se utilicen de acuerdo con la referencia de las herramientas. Un ejemplo de tal herramienta podría ser JSDoc. Colocar comentarios en el código de acuerdo con el formato de esta herramienta permitirá generar un sitio web que contenga información detallada sobre un proyecto (por ejemplo, descripciones de funciones, sus parámetros de llamada, valores devueltos, etc.).

**Alternar código**

A veces tenemos algún fragmento de código que creemos que está causando algunos problemas o queremos comprobar algunas opciones rápidamente. Comentar el código es una gran herramienta para ayudar con eso.

// const greetingText = "Hola";

const greetingText = "Hola";

// const greetingText = "Bienvenido";

Podemos deshabilitar cada línea de código usando comentarios de una sola línea como en el ejemplo. Usando comentarios de bloque, podemos comentar grandes fragmentos de código o funciones completas. Esto es muy útil cuando se identifican algunos problemas de código.

Resumen

Por el momento, el uso de comentarios puede parecerte una simple curiosidad. Recuerda que con el tiempo, comenzarás a escribir programas que se volverán cada vez más complejos. Consistirán en muchos archivos, con cientos o miles de líneas de código. Los comentarios te brindan la oportunidad de aumentar la claridad del código agregando información que ayudará a otros a comprender partes seleccionadas del mismo. El siguiente paso será la capacidad de generar documentación de proyectos a partir de ellos, aunque para ello deberás ajustar tus comentarios a la convención que impone la herramienta seleccionada. Los comentarios también son muy útiles para activar y desactivar piezas seleccionadas de código, que usamos con mayor frecuencia cuando probamos versiones alternativas o cuando buscamos errores en ellas.

**Tarea**

Hay un código que actualmente no funciona. Intenta arreglarlo usando solo comentarios. Intenta, si es posible, usar los atajos de teclado en tu editor para este propósito.

Revisar

"use strict";

const prefix = "username\_";

let userName = "Jack";

// const userName = "Adam";

let prefixedUserName;

// const prefixedUserName;

userName = "John";

prefixedUserName = prefix + userName;

console.log(prefixedUserName /\*+ prefixedUserName2\*/);

// console.log(prefixedUserName2);

**Tareas**

**Tarea 1**

Existe un código que actualmente no funciona. Intenta arreglarlo usando solo comentarios. Intenta, si es posible, usar los atajos de teclado en tu editor para este propósito.

"use strict";

const prefix = "username\_";

let userName = "Jack";

const userName = "Adam";

let prefixedUserName;

const prefixedUserName;

userName = "John";

prefixedUserName = prefix + userName;

console.log(prefixedUserName + prefixedUserName2);

console.log(prefixedUserName2);  
Ejemplo

"use strict";

const prefix = "username\_";

let userName = "Jack";

// const userName = "Adam";

let prefixedUserName;

// const prefixedUserName;

userName = "John";

prefixedUserName = prefix + userName;

console.log(prefixedUserName /\*+ prefixedUserName2\*/);

// console.log(prefixedUserName2);