Introducción a JavaScript

JavaScript se ha convertido en el lenguaje más popular de programación, y por una buena razón. JavaScript es el único lenguaje que entienden los navegadores como Internet Explorer, Chrome, etc. Y ahora, gracias a una plataforma llamada Node.js, hoy es posible utilizar JavaScript fuera del navegador para crear todo tipo de aplicaciones: aplicaciones de escritorio, aplicaciones móviles, aplicaciones de la línea de comandos y backends de aplicaciones Web, entre otros.

¿Por qué otro libro de JavaScript?

Aunque existe una multitud de recursos en Internet sobre JavaScript, no existía una introducción en Español clara que cualquiera, incluso la persona más principiante, pudiera seguir.

¿A quién está dirigido?

Esta introducción a JavaScript está dirigida a **no programadores**. En vez de ser una referencia completa del lenguaje, nuestro propósito es que puedas adquirir el conocimiento necesario para empezar a solucionar ejercicios y continuar tu aprendizaje.

El editor de texto

Para escribir código vas a necesitar un **editor de texto**. Un **editor de texto** es una aplicación que nos permite crear y editar archivos de texto.

A diferencia de un procesador de palabras (como Microsoft Word), un editor de texto se utiliza para crear archivos de texto **sin formato**.

Cada sistema operativo trae algún editor de texto como **Bloc de Notas** en Windows o **TextEdit** en Mac. Sin embargo, existen editores de texto especializados para programadores que tienen varias ventajas sobre el editor que viene en tu sistema operativo:

- Resalta la sintaxis de acuerdo al lenguaje de programación para facilitar su lectura.
- Ayuda a auto completar el código.
- Es altamente personalizable a través de archivos de configuración o plugins.

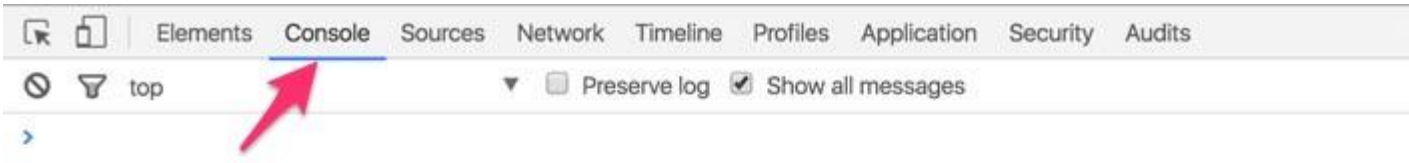
Ejemplos de editores de texto para programadores incluyen Sublime Text, Atom y VSCode, entre muchos otros. Sin embargo, si aún no tienes una preferencia, nuestra recomendación es SublimeText.

A través de las herramientas de desarrollador

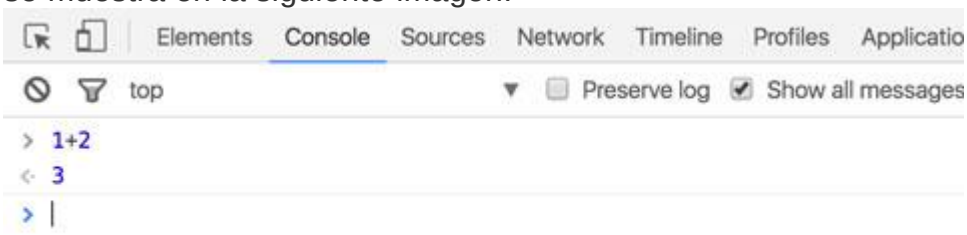
Las herramientas de desarrollador (en inglés developer tools) son un conjunto de herramientas integradas al navegador que utilizan los Desarrolladores Front End para analizar, depurar y mejorar su código.

La forma más fácil de abrir las herramientas de desarrollador en cualquier navegador es hacer click en cualquier parte de la página y seleccionar la opción "Inspeccionar Elemento" en el menú desplegable que aparece.

También existe un atajo del teclado para abrir y cerrar las herramientas de desarrollador. El atajo para la mayoría de navegadores en Mac es `Alt` + `Command` + `I`. Para PC es `Ctrl` + `Shift` + `I`.
Una de las herramientas que incluyen las herramientas de desarrollador es la **Consola**, que la puedes abrir haciendo click en la pestaña "Consola" (o en Inglés "Console") como se muestra en la siguiente imagen.



En la **Consola** podemos escribir una expresión de JavaScript, oprimir Enter, y ver el resultado de esa expresión en la siguiente línea. Por ejemplo, escribe `1+2` y oprime Enter. Deberás ver el número `3` en la siguiente línea como se muestra en la siguiente imagen.



A través de un documento HTML

La otra forma de ejecutar código JavaScript en el navegador es dentro de un documento HTML. Crea un archivo llamado `index.html` y pega el siguiente contenido:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript</title>
  </head>
  <body>
    <script>
      alert("Hola Mundo");
    </script>
  </body>
</html>
```

Ábrelo con tu navegador preferido. Deberías ver un mensaje de alerta con el texto "Hola Mundo".

Aunque insertar el código directamente dentro del HTML funciona, se considera una mala práctica. Crea un nuevo archivo llamado `app.js` en la misma carpeta donde se encuentre `index.html` y pega el siguiente contenido:

```
alert("Hola Amigo");
```

Ahora modifica `index.html` con el siguiente contenido:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript</title>
  </head>
  <body>
    <script src="app.js"></script>
  </body>
</html>
```

Ábrelo nuevamente con un navegador o refresca la página si ya lo tenías abierto. Deberías ver una alerta pero ahora con el texto "Hola Amigo".

Ejecutando código en Node.js

Existen dos formas de ejecutar código JavaScript en Node.js: desde la consola de Node.js o desde un archivo.

La consola de Node.js

Para abrir la consola de Node.js ejecuta el siguiente comando desde la línea de comandos:

```
$ node
>
```

La consola de Node.js nos permite escribir una **expresión** de JavaScript, oprimir Enter, y ver el resultado de esa expresión en la siguiente línea, muy parecido a cómo lo hicimos sobre la consola del navegador en la sección pasada.

Por ejemplo, si escribimos `1+2` y oprimimos `Enter` debería mostrar `3` en la siguiente línea:

```
$ node
> 1 + 2
3
>
```

Para salir de la consola oprime `Ctrl` + `D`.
Nota: En JavaScript el punto y coma (`;`) al final de cada expresión es opcional. Cuando estemos trabajando en la consola de Node.js los vamos a omitir. Sin embargo, cuando mostremos código que va a ir en un archivo los incluímos porque es una buena práctica.

Desde un archivo

La otra forma de ejecutar código JavaScript en Node.js es crear un archivo con extensión `js` en el que escribimos nuestro código y lo ejecutamos con el comando `node`.
Crea un archivo llamado `app.js`, ábrelo con tu editor favorito y pega el siguiente contenido:

```
console.log("Hola Mundo!");
```

Guárdalo y ejecuta `node app.js` sobre la línea de comandos (asegúrate de estar ubicado sobre la carpeta donde se encuentra el archivo). Deberías ver el texto "Hola Mundo" en la siguiente línea:

```
$ node app.js
Hola Mundo
```

Cambia el texto por cualquier otro y vuelve a ejecutar el archivo.

¡Felicitaciones, has creado tu primer programa en JavaScript con Node.js!

Errores

Veamos ahora qué pasa si cometemos algún error en nuestro código. Por ejemplo, borra el caracter `l`de la palabra `console` y vuelve a ejecutar el archivo. Te debería aparecer un mensaje de error similar al siguiente:

```
/Users/germanescobar/Projects/node/app.js:1
(function (exports, require, module, __filename, __dirname) { consoe.log("Hola
Mundo");
                                                              ^

ReferenceError: consoe is not defined
    at Object.<anonymous> (/Users/germanescobar/Projects/node/app.js:1:63)
    at Module._compile (module.js:571:32)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
```

```
    at Function.Module._load (module.js:439:3)
    at Module.runMain (module.js:605:10)
    at run (bootstrap_node.js:420:7)
    at startup (bootstrap_node.js:139:9)
    at bootstrap_node.js:535:3
```

Toma un tiempo acostumbrarse a leer los mensajes de error de Node.js pero ahí está todo lo que necesitas saber para solucionarlo. El caracter `^` nos muestra dónde ocurrió el error (aunque está mezclado con otro código que genera Node.js) y debajo de esa línea una frase que dice `ReferenceError: consoe is not defined`.

Hay veces en los que es fácil encontrar los errores, otras veces no es tan fácil. Lo que si es cierto es que a medida que vayas trabajando con el lenguaje vas a ir desarrollando una intuición que te va a permitir solucionar los errores más fácilmente, pero al principio es un proceso lento que es parte de ese aprendizaje.

Cometamos otro error intencionalmente para ver un mensaje diferente. Vuelve a escribir `console`correctamente, pero ahora borra la comilla al final de la cadena de texto así:

```
console.log("Hola Mundo);
```

Y vuelve a ejecutar el archivo. Debería salir un mensaje como el siguiente:

```
/Users/germanescobar/Projects/node/app.js:1
(function (exports, require, module, __filename, __dirname) {
console.log("Hola Mundo);

^^^^^^^^^^^^^
SyntaxError: Invalid or unexpected token
    at Object.exports.runInThisContext (vm.js:78:16)
    at Module._compile (module.js:543:28)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
    at Function.Module._load (module.js:439:3)
    at Module.runMain (module.js:605:10)
    at run (bootstrap_node.js:420:7)
    at startup (bootstrap_node.js:139:9)
    at bootstrap_node.js:535:3
```

Esta vez el mensaje `SyntaxError: Invalid or unexpected token` no es tan claro, pero fíjate que nos indica dónde está el problema con varios caracteres `^`.

Escribiendo más líneas

Vuelve a agregar la comilla y verifica que se ejecute normalmente.

Node.js ejecuta el archivo línea por línea, una después de la otra. Así que podemos agregar una segunda línea a nuestro archivo:

```
console.log("Hola Mundo");
console.log("Esto está muy bacano");
```

El ejecutar el archivo deberías ver el siguiente resultado:

```
$ node app.js
Hola Mundo
Esto está muy bacano
```

Comentarios

Los comentarios se utilizan para documentar o aclarar nuestro código y son ignorados al ejecutar el archivo. En JavaScript se utilizan los caracteres `//` para crear un comentario de una línea. Por ejemplo:

```
// este es un comentario de una línea
console.log("Hola Mundo");
console.log("Esto está muy bacano"); // este es otro comentario
```

También puedes crear comentarios de múltiples líneas encerrando el texto entre `/*` y `*/`. Por ejemplo:

```
/*
 este es un comentario
 de varias líneas
 console.log("esto no aparece al ejecutar el archivo")
*/
console.log("Hola Mundo");
console.log("Esto está muy bacano");
```

Fíjate que la última línea del comentario es código JavaScript válido. Sin embargo, **ese código no se ejecuta porque está como comentario**.

Tipos y Operadores

En este capítulo vamos a hablar sobre cadenas de texto, números y booleanos (verdadero o falso), que son tipos de datos básicos en JavaScript, y cómo realizar algunas operaciones con ellos. Empecemos con las cadenas de texto.

Cadenas de texto

Una cadena de texto es un conjunto de caracteres encerrados entre comillas simples (`'`) o dobles (`"`). Por ejemplo:

```
"Texto entre comillas dobles"
'Texto entre comillas simples'
```

Aunque parece fácil, existen tres errores comunes al definir una cadena de texto para que los tengas en cuenta e intentes evitarlos:

1. Olvidarse de la comilla de cierre. Por ejemplo:

```
"Hola mundo
```

Variables

Las variables son uno de los conceptos básicos de la programación y nos permiten almacenar información temporal que podemos usar más adelante en nuestros programas.

Crea un archivo llamado `variables.js` y agrega lo siguiente:

```
var name = "Germán"; // cámbialo por tu nombre
console.log("Hola " + name);
```

En este ejemplo estamos definiendo una variable con nombre `name` y le asignamos el valor "Germán" (o el valor que le hayas asignado). En la siguiente línea estamos utilizando concatenación de cadenas para mostrar la cadena de texto "Hola " seguido del valor que tenga en ese momento la variable `name`.
Las variables se crean con la palabra clave `var` seguido del nombre de la variable. Opcionalmente, le puedes asignar un valor a la variable utilizando el caracter igual y el valor que le quieras dar. El punto y coma (`;`) al final es opcional pero se considera una buena práctica tenerlo.
El nombre de una variable debe comenzar con `$`, `_` o una letra, y después puede contener letras, dígitos, `_` y `$`. Ejemplos de nombres válidos de variables incluyen `name`, `$element` y `_trains`.
Por otro lado, ejemplos de nombres no válidos incluyen `443german`, porque no puede empezar con un número, y `element&123`, porque el caracter `&` no es válido en el nombre.
Las palabras reservadas de JavaScript no se pueden usar como nombres de variables.

Como buena práctica se recomienda empezar las variables con una letra en minúscula y, si el nombre se compone de varias palabras, capitalizar cada palabra después de la primera (más conocido como camel case). Por ejemplo `videoTranscoder` o `firstName`.
Los nombres de las variables diferencian mayúsculas y minúsculas (p.e. `firstname` es diferente a `firstName`).

La utilidad de las variables

Crea un archivo llamado `square.js` y escribe el siguiente código:

```
console.log("El perímetro de un cuadrado de lado 5 es " + (5 * 4));
console.log("El área de un cuadrado de lado 5 es " + (5 * 5));
```

Al ejecutarlo debería aparecer lo siguiente:

```
$ node square.js
El perímetro de un cuadrado de lado 5 es 20
El área de un cuadrado de lado 5 es 25
```

El problema con este código es que si quisiéramos calcular el perímetro y el área de un cuadrado de lado 10, o 20, tendríamos que modificar ese valor en varias partes del código. Podemos mejorarlo utilizando una variable:

```
var side = 5;

console.log("El perímetro de un cuadrado de lado " + side + " es " + (side *
4));
```

```
console.log("El área de un cuadrado de lado " + side + " es " + (side *
side));
```
Si ejecutas el código te debería dar el mismo resultado. La ventaja es que si quieres calcular el perímetro y el área de un cuadrado con otro tamaño solo debes cambiar el valor de la variable. Intenta con 18 (te debería dar 72 de perímetro y 324 de área) y después con 39.

Reasignando el valor de las variables

Puedes reasignar el valor de una variable las veces que lo desees. La forma de hacerlo es similar a la forma en que se declara la una variable con un valor inicial, pero omitiendo la palabra `var`. Por ejemplo:

```
// asumiendo que name fue ya declarada
name = "Nuevo valor";
```
Inténtalo. Abre la consola de Node.js e ingresa lo siguiente:

```
$ node
> var name = "Pedro" // declaramos name y le asignamos el valor "Pedro"
undefined
> name               // verificamos el valor actual
"Pedro"
> name = 123         // reasignamos el valor de name con el número 123
123
> name               // verificamos cuál es el valor actual de name
123
```
Fíjate que, como en este ejemplo, el nuevo valor que se le asigne a la variable no tiene que ser del mismo tipo del valor anterior.

También es posible reasignar el valor de una variable utilizando su valor anterior. Por ejemplo, intenta lo siguiente:

```
$ node
> var count = 1      // declaramos la variable con un valor inicial
undefined
> count              // verificamos cuál es el valor actual de count
1
> count = count + 1  // incrementamos en uno el valor actual de count
2
> count              // verificamos el valor actual de count
2
```
De hecho, incrementar el valor de una variable es tan común que existe un atajo para eso. Asumiendo que sigues en la consola de Node.js intenta lo siguiente:

```
> count ++
3
> count ++
4
```

Variables sin valor

En programación es muy común declarar una variable sin un valor, quizá porque más adelante vamos a pedirle el valor al usuario, o simplemente porque el valor se lo vamos a asignar después.

Una variable declarada sin un valor va a tener el valor de `undefined`.
Abre la consola de Node.js intenta lo siguiente:

```
$ node
> var name
undefined
> name
undefined
```

¿Dónde y cuánto vive una variable?

Las variables se almacenan en en una memoria especial del computador llamada **memoria RAM** y viven durante la ejecución del programa, es decir, desde el momento en que las defines hasta que tu programa termina de ejecutarse. Si abres la consola de Node.js y defines una variable, esta vive hasta que cierres esa sesión.

La **memoria RAM** es una memoría de rápido acceso que está disponible mientras tu computador está encendido. El sistema operativo se encarga de administrar la memoria RAM y asignarle una porción a cada programa que se está ejecutando. Cuando el programa termina, el sistema operativo reclama esa memoria y "destruye" todas las variables que ese programa haya creado.

Nota: Más adelante, cuando veamos funciones, aprenderemos que las variables tienen un **alcance** y no todas las variables sobreviven hasta que termina el programa.

Condicionales

Hasta ahora hemos visto código que se ejecuta línea a línea, una detrás de otra. Pero a veces se hace necesario romper esa secuencia y crear ramas que nos permitan tomar diferentes caminos en el código dependiendo de ciertas condiciones.

Por ejemplo, imagina cómo podríamos hacer un programa que nos diga si un número es mayor o menor a diez. Si es mayor a 10 debería imprimir una cosa, pero si es menor debería imprimir otra.

A este concepto se le conoce como condicionales y su sintaxis es la siguiente:

```
if (<condición>) {
  // código que se ejecuta si se cumple la condición
}
```

La condición puede ser cualquier expresión que evalúa a verdadero (`true`) o falso (`false`). Crea un archivo llamado `conditionals.js` y agrega el siguiente contenido:

```
if (true) {
  console.log("Hola Mundo");
}
```

Nota: Las líneas que terminan con un corchete (`{` o `}`) no se les agrega punto y coma (;).
Ejecúta el archivo desde la consola, deberías ver lo siguiente:

```
$ node conditionals.js
Hola Mundo
```

No importa cuántas veces ejecutes este archivo, el resultado siempre será el mismo.

Ahora probemos con falso (`false`) en vez de verdadero (`true`):

```
if (false) {
  console.log("Hola Mundo");
}
```

Ejecútalo. Esta vez **nunca** debería imprimir "Hola mundo", no importa cuantas veces lo ejecutes.

En vez de utilizar `true` o `false` como condición, podemos utilizar una expresión **que evalúe a `true`o `false`**.

```
if (1 == 1) {
  console.log("Hola Mundo");
}
```

El resultado al ejecutarlo debería ser:

```
$ node conditionals.js
Hola mundo
```

Prueba ahora con `1 == 2`, `1 < 6` y `8 < 6` en la condición y fíjate que tenga sentido.
Ahora que ya sabes cómo funciona los condicionales (muchos los llamamos los ifs) crea un programa en un archivo llamado `number.js` que imprima "El número es menor a 10" solo si el número que está almacenado en la variable `num` es menor a 10:

```
var num = 8;

if (num < 10) {
  console.log("El número es menor a 10");
}
```

Si lo ejecutas te debería aparecer lo siguiente:

```
$ node number.js
El número es menor a 10
```

Ahora vamos a modificar el programa para que, además de imprimir "El número es menor a 10" si el número es menor a 10, también imprima "El número es igual o mayor a 10" si el número es igual o mayor a 10. Podemos utilizar otro condicional debajo del que ya teníamos:

```
var num = 8;

if (num < 10) {
  console.log("El número es menor a 10");
}

if (num >= 10) {
```

```
  console.log("El número es igual o mayor a 10");
}
```
Ejecuta el programa e ingresa un número menor a 10, después un número mayor a 10, y por último 10. Verifica que el resultado sea el esperado.

De lo contrario (else)

Lo único que necesitas para hacer condicionales es el `if`. Pero existen dos atajos que te van a permitir escribir código más corto.
El primer atajo es el `else`, que significa "de lo contrario" en Inglés. El `else` nos permite definir el código que se debe ejecutar si el `if` no se cumple, es decir si la condición evalúa a falso. La sintaxis es la siguiente:

```
if (<condición>) {
  // código que se ejecuta si se cumple la condición
} else {
  // código que se ejecuta si NO se cumple la condición
}
```
Podemos modificar el programa anterior, que nos dice si el número almacenado en la variable `num` es menor a 10, o si es mayor o igual, con un `else`.

```
var num = 8;

if (num < 10) {
  console.log("El número es menor a 10");
} else {
  console.log("El número es igual o mayor a 10");
}
```
Más corto y si lo ejecutas debería funcionar igual.

Condiciones anidadas

Ahora imagina que queremos modificar este programa para que en vez de imprimir "El número es igual o mayor a 10", imprima "El número es igual a 10" o "El número es mayor a 10" dependiendo si el número es igual 10 o mayor a 10 respectivamente.

En JavaScript (y en la mayoría de lenguajes de programación) es posible anidar condicionales, así que una posible solución sería la siguiente:

```
var num = 8;

if (num < 10) {
  console.log("El número es menor a 10");
} else {
  if (num > 10) {
    console.log("El número es mayor a 10");
  } else {
    console.log("El número es igual a 10");
  }
}
```

Prúebalo con un número menor a 10, otro mayor a 10 y con 10. Te debería aparecer "El número es menor a 10", "El número es mayor a 10" y "El número es igual a 10" respectivamente.

De lo contrario, si (else if)

En general, es preferible no tener que anidar condicionales porque son difíciles de leer y entender. Otro atajo que nos ofrece JavaScript para los condicionales es el `else if`, que significa "De lo contrario, si ..." en Inglés. La sintaxis es la siguiente:

```
if (<primera condición>) {
  // código que se ejecuta si <primera condición> se cumple
} else if (<segunda condición>) {
  // código si <primera condición> NO se cumple, pero <segunda condición> se cumple
} else if (<tercera condición>) {
  // código si <primera condición> y <segunda condición> NO se cumplen, pero <tercera condición> sí se cumple
} else {
  // código si ninguna de las condiciones se cumple
}
```

Puedes definir tantos `else if` como desees. El `else` es opcional.
Modifiquemos nuestro ejemplo anterior y en vez de utilizar condiciones anidadas, utilicemos `else if`:

```
var num = 8;

if (num < 10) {
  console.log("El número es menor a 10");
} else if (num > 10) {
  console.log("El número es mayor a 10");
} else {
  console.log("El número es igual a 10");
}
```

Lo más importante de entender en este código es que el programa sólo va a entrar a **una** de estas ramas. Por ningún motivo va a entrar a dos de ellas. Si la condición del primer `if` se cumple, el programa ejecuta el código que esté en ese bloque y después **salta** hasta después del `else` para continuar con el resto del programa o terminar.
Si la condición del primer `if` no se cumple, pero la del `else if` sí se cumple, el programa ejecuta el código de ese bloque y **salta** hasta después del `else` para continuar con el resto del programa o terminar.

Condiciones compuestas

Imagina que queremos escribir un programa que imprima "El número está entre 10 y 20" si el valor de una variable está efectivamente entre 10 y 20. ¿Cómo te imaginas que lo podríamos solucionar?

Una opción es usar condiciones anidadas, de esta forma:

```
var num = 15;

if (num >= 10) {
  if (num <= 20) {
    console.log("El número está entre 10 y 20");
  }
}
```

Sin embargo, cómo decíamos antes, leer condiciones anidadas es difícil y, en lo posible, es mejor evitarlas. En cambio, podemos utilizar los operadores lógicos **y** (`&&`) y **ó** (`||`) para crear condiciones compuestas. El ejemplo anterior lo podemos mejorar con **y**:

```
var num = 15;

if (num >= 10 && num <= 20) {
  console.log("El número está entre 10 y 20");
}
```

Lo que estamos diciendo con este código es: si el número es **mayor o igual a 10 y menor o igual 20**entonces imprima "El número está entre 10 y 20". Fíjate que a cada lado del `&&` hay una expresión que evalúa a verdadero o falso: `num >= 10` y `num <= 20`.

Imagina ahora que necesitamos escribir un programa que imprima "Excelente elección" cuando el valor de una variable sea "rojo" **o** "negro" únicamente:

```
var color = "negro";

if (color === "rojo" || color === "negro") {
  console.log("Excelente elección");
}
```

Operador condicional (ternario)

Los operadores ternarios se utilizan con frecuencia como atajos para los condicionales `if`. Este está compuesto de la siguiente forma `<condición> ? <expr1> : <expr2>`. Ahora desglosémoslo paso a paso para entender un poco mejor como funciona.

Lo primero que se está haciendo y lo que esta antes de `?` es la condición que queremos validar; si esto es verdadero se ejecutara la `expr1` de lo contrario se ejecutara la `expr2`.

```
var num = 10;

num >= 15 ? console.log('Es mayor o igual que 15') : console.log('Es menor que 15');
```

Pensando como un programador

Vamos a jugar un juego llamado **Verdadero o Falso**. Yo digo una afirmación y tu debes responder si es verdadera o falsa. Trata de no mirar las respuestas debajo. Después comparas:

1. La Tierra gira alrededor del sol. (¿Verdadero o falso?)
2. Paris es la capital de Estados Unidos.
3. La Tierra gira alrededor del sol **y** los leones son animales.
4. Paris es la capital de Estados Unidos **y** los leones son animales.
5. La Tierra gira alrededor de Marte **y** los perros hablan Español.
6. Los leones son animales **o** la Tierra gira alrededor del sol.
7. Paris es la capital de Estados Unidos **o** los leones son animales.
8. El planeta tierra gira alrededor de Marte **o** los perros hablan Español.

Las respuestas son las siguientes:

1. Verdadero.
2. Falso.
3. Verdadero.
4. Falso.
5. Falso.
6. Verdadero.
7. Verdadero.
8. Falso.

Cuando utilizamos **y** las dos expresiones deben ser verdaderas para que el resultado sea verdadero. Cuando utilizamos **o** cualquiera de las dos expresiones puede ser verdadera para que el resultado sea verdadero.

Evaluando expresiones booleanas

Volvamos a jugar el juego, pero en vez de utilizar frases, utilicemos expresiones booleanas. Debes decidir si cada una de las siguientes expresiones es verdadera o falsa (`true` o `false`):

1. `true`
2. `false`
3. `1 < 1`
4. `2 != 3`
5. `1 < 1 && 2 != 3`

Copia y pega cada expresión en la consola de Node.js para conocer las respuestas.

Analicemos la última expresión: `1 < 1 && 2 != 3`. ¿Cómo podemos saber si es verdadera o falsa?
El primer paso es reemplazar cada lado de la expresión. `1 < 1` es `false` y `2 != 3` es `true`. Quedaría:
`false && true`
Recuerda que para que una expresión con **y** (`&&`) sea verdadera, cada lado **tiene** que ser verdadero. Sin embargo, podemos hacer una tabla con todas las combinaciones entre verdadero y falso que podamos usar como referencia más adelante:

Expresión	Resultado
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Fíjate que el resultado solo es `true` cuando los dos lados del `&&` son `true`.
Hagamos lo mismo para el **ó** (`||`):

Expresión	Resultado
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Con el **ó** cualquiera de los lados puede ser `true` para que el resultado sea `true`.
A estas tablas se les conoce como **Tablas de Verdad**.

Hagamos algunos ejercicios. Decide si las siguientes expresiones evalúan a `true` o `false`. Primero reemplaza cada lado del `&&` o el `||` y luego utiliza las tablas de verdad:

- `"hola" == "hola" && 1 < 2`
- `true && 5 != 5`
- `1 == 1 || 2 != 1`

Revisa tu respuesta evaluando cada expresión en la consola de Node.js.

Podemos negar cualquier expresión booleana anteponiendo un signo de exclamación (`!`). Por ejemplo:

- `!true` es `false`
- `!false` es `true`

De hecho, esa es la tabla de verdad de la negación. Intenta los siguientes ejercicios. Primero reemplaza lo que está entre paréntesis y luego aplica la tabla de verdad de la negación:

- `!(1 === 1)`
- `!(2 <= 3)`
- `!(true && 5 !== 5)`
- `!(1 < 1 && 2 !== 3)`

El proceso para solucionar cualquier expresión booleana, sin importar qué tan compleja sea, es el siguiente:

1. Evalúa los operadores de igualdad (`<`, `>`, `===`, `!==` etc).
2. Evalúa los `&&` y `||` que esten dentro de paréntesis.
3. Evalúa las negaciones (`!`).
4. Evalúa cualquier `&&` y `||` que falte.

Hagamoslo juntos. Intentemos evaluar la siguiente expresión booleana:

```
3 != 4 && !("pedro" === "juan" || 26 > 10)
```

1. Evaluar los operadores de igualdad:

```
2. true && !(false || true)
```

3. Evaluar los `&&` y `||` que estén dentro de paréntesis:

```
4. true && !true
```

5. Evaluar las negaciones:

```
6. true && false
```

7. Evaluar cualquier `&&` y `||` que falte:

```
8. false
```

Inténtalo tu. Decide si las siguientes expresiones evalúan a `true` o `false`:

- `!(5 === 5) && 8 !== 8`
- `("gut" === "ikk" && 26 > 30) || ("gut" === "gut" && 26 > 10)`
- `!("testing" == "testing" && !(5 > 8))`

Iteraciones o ciclos

Los ciclos nos permiten repetir la ejecución de un código varias veces. Imagina que quisiéramos repetir la frase "Hola mundo" 5 veces. Podríamos hacerlo manualmente. Crea un archivo llamado `loops.js` y escribe el siguiente código:

```
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
```

Ejecútalo y deberías ver la frase "Hola mundo" 5 veces en tu pantalla:

```
$ node loops.js
Hola mundo
Hola mundo
Hola mundo
Hola mundo
Hola mundo
```

Ahora imagina que quisieramos repetirlo 850 veces. Ya no sería tan divertido copiar todo ese número de líneas en el archivo. Podemos entonces utilizar un ciclo. Un ciclo se crea utilizando la palabra clave `while` seguido de una condición, que va a definir el número de veces que se va a repetir ese ciclo.
Reemplaza el contenido del archivo `loops.rb` por el siguiente:

```
var i = 0;
while (i < 850) {
  console.log("Hola mundo");
  i = i + 1;
}
```

Ejecútalo y revisa que la frase "Hola mundo" aparezca 850 veces. Como ejercicio modifícalo para que aparezca el valor de `i` antes de cada frase.
Debería salir algo así (omitimos algunas líneas para no gastar tanto pap...ehhh...espacio en disco):

```
$ node loops.js
0 Hola mundo
1 Hola mundo
2 Hola mundo
...
345 Hola mundo
...
849 Hola mundo
```

Un ciclo en JavaScript tiene la siguiente sintaxis:

```
while (<condicion>) {
  // acá va el cuerpo del ciclo, el código que se va a repetir mientras la
condición se cumpla
}
```

La condición puede ser cualquier valor o expresión booleana. El cuerpo del ciclo se va a ejecutar mientras que la condición se cumpla. Por ejemplo, crea un archivo llamado `inifiniteLoop.js` que contenga lo siguiente:

```
while (true) {
  console.log("Hola Mundo");
}
```

¿Qué crees que va a ocurrir? Antes de ejecutarlo debes saber que puedes interrumpir cualquier programa oprimiendo `Ctrl` + `C` :)
El código anterior crea lo que en programación llamamos un **ciclo infinito**. Intenta evitarlos.

En el momento en el que la condición deja de cumplirse el ciclo se detiene y continúa con el resto del programa. Podemos crear un ciclo que nunca va a ejecutar el cuerpo del ciclo:

```
while (false) {
  console.log("Hola mundo");
}
```

Si ejecutas ese código no deberías ver ninguna frase `"Hola mundo"`.
En vez de `true` o `false` puedes utilizar cualquier otra condición como lo hicimos en el ciclo que muestra `"Hola mundo"` 850 veces:

```
var i = 0;
while (i < 850) {
```

```
  console.log("Hola mundo");
  i++;
}
```

Primero declaramos una variable `i` que inicia en 0. Cada vez que ingresa en el ciclo la vamos a incrementar en 1 hasta que lleguemos a 850. En ese momento la condición va a dejar de ser verdadero y el ciclo se detendrá.

for

El `while` es todo lo que necesitas para hacer ciclos en JavaScript. Sin embargo, ese patrón que vimos en el ejemplo anterior en el que tenemos una **inicialización** (`var i = 0`), una **condición** (`i < 850`) y un **incrementador** (`i++`) es tan común, que JavaScript tiene un atajo para esto, el `for`.

El `for` tiene la siguiente sintaxis:

```
for (<inicialización>; <condición>; <incrementador>) {
  // el cuerpo del ciclo, el código que se repite mientras que la condición sea verdadera
}
```

El ejemplo anterior lo podemos reescribir de la siguiente forma:

```
for (var i = 0; i < 850; i ++) {
  console.log("Hola mundo");
}
```

Son equivalentes, la única diferencia es que el inicializador, la condición y el incrementador están definidos en la misma línea, pero se ejecuta de la misma forma que el `while`:

- La **inicialización** se ejecuta antes de evaluar la **condición** por primera vez.
- La **condición** se ejecuta cada vez que se itera.
- El **cuerpo** se ejecuta cada vez que la **condición** se cumple.
- El **incrementador** se ejecuta cada vez que el **cuerpo** se ejecuta, antes de volver a evaluar la **condición**.

Nota: Tanto el inicializador, la condición y el incrementador son opcionales. Si quisieras podrías hacer un ciclo infinito con un `for` de la siguiente forma:

```
for(;;) {
  // el cuerpo del ciclo también es opcional
}
```

Algunos ejemplos

Imagina que queremos hacer un programa que imprima los números del 10 a 20 pero saltando cada otro número, es decir, que imprima 10, 12, 14, 16, 18 y 20.

El primer paso antes de escribir el ciclo es identificar las diferentes partes del ciclo: la **inicialización**, la **condición**, el **incrementador** y el **cuerpo**.

Para este ejemplo serían:

1. **Inicializa** una variable en 10.
2. La **condición** es que la variable sea menor o igual a 20.
3. **Incrementa** la variable en dos en cada iteración.
4. El **cuerpo** debe imprimir la variable.

Con esta información ya puedes implementar el ciclo con `while` o `for`, recuerda que son equivalentes. La solución utilizando un `while` sería:

```
var i = 10;       // el inicializador
while (i <= 20) { // la condición
  console.log(i);
  i = i + 2;      // el incrementador
}
```

Podemos hacer lo mismo con un `for`:

```
for (var i=10; i <= 20; i = i + 2) {
  console.log(i);
}
```

Hagamos otro ejemplo. Imagina que queremos imprimir los números del 1 a al 100 pero deforma descendente, es decir, 100, 99, 98 ... 1. Empecemos identificando las partes del ciclo:

1. **Inicializa** una variable en 100.
2. La **condición** es que la variable sea mayor que 0.
3. El **incrementador** va a ser un decrementador en este caso, va a decrementar la variable en 1 cada iteración.
4. El **cuerpo** debe imprimir la variable.

```
var i = 100;
while (i > 0) {
  console.log(i);
  i--;
}
```

Arreglos

Hasta ahora hemos trabajado con cadenas de texto, números y booleanos. En este capítulo vamos a hablar de un nuevo tipo de datos: los arreglos.

Un arreglo es una lista ordenada de elementos de cualquier tipo. Para crear tu primer arreglo abre la consola de Node.js y escribe lo siguiente:

```
var array = [1, "Pedro", true, false, "Juan"]
```

La sintaxis de un arreglo es muy simple. Los elementos del arreglo se envuelven entre corchetes y se separan con coma. Fíjate que el arreglo que creamos contiene números, cadenas de texto y booleanos. Cada elemento del arreglo puede ser de cualquier tipo (incluso otros arreglos!).

Obteniendo elementos del arreglo

Para obtener la primera posición del arreglo que acabamos de crear utilizas `array[0]`:

```
$ node
> array = [1, "Pedro", true, false, "Juan"]
[1, "Pedro", true, false, "Juan"]
> array[0]
1
```

La sintaxis para obtener un elemento del arreglo es `[n]` donde `n` es la posición empezando en 0. Imprime ls demás elementos del arreglo:

```
> array[1]
"Pedro"
> array[2]
true
> array[3]
false
> array[4]
"Juan"
```

Puedes obtener el ultimo elemento del arreglo obteniendo la longitud de este y restandole 1, lo cual seria [array.length - 1]. Hay que tener en cuenta que la longitud del arreglo no comienza desde 0 sino desde 1.

```
> array[array.length - 1]
"Juan"
```

Recorriendo un arreglo

En el ejemplo anterior pudimos imprimir cada una de las posiciones porque era un arreglo de pocos elementos. Sin embargo esto no siempre es práctico. Primero, el arreglo puede ser muy grande o puede que no sepamos el tamaño del arreglo. Crea un archivo llamado `arrays.js` y escribe el siguiente código:

```
var array = [1, "Pedro", true, false, "Juan"];

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

Reemplazando un elemento

Es posible reemplazar el valor de cualquier elemento del arreglo. Por ejemplo:

```
var array = [1, "Pedro", true, false, "Juan"];
array[1] = "Germán"; // reemplazamos el elemento en la posición 1

// [1, "Germán", true, false, "Juan"]
```

En este ejemplo estamos reemplazando la posición 1 del arreglo (que realmente es la segunda porque recuerda que empieza en 0) con el valor "Germán". La línea más importante es la siguiente:

```
array[1] = "Germán";
```
Como ejercicio intenta reemplazar el último elemento ("Juan") por otro valor.

Insertando nuevos elementos

Es posible insertar nuevos elementos en un arreglo (puede estar vacío o tener elementos). Por ejemplo:

```
var array = ["Pedro"];
array.push("Germán"); // ["Pedro", "Germán"]
array.push("Diana"); // ["Pedro", "Germán", "Diana"]
```
El método `push` te permite agregar un elemento al final de la lista. ¿Qué pasa si queremos agregar un elemento en otra posición? Para eso sirve el método `splice`:
```
var array = ["Pedro", "Germán", "Diana"];
array.splice(0, 0, "Juan") // ["Juan", "Pedro", "Germán", "Diana"]
```
El método `splice` se utiliza tanto para insertar como eliminar elementos. Para insertar debes pasarle 3 o más argumentos. El primer argumento es la posición en la que quieres insertar el elemento. La segunda posición debe estar en `0`. Los demás argumentos son los elementos que deseas insertar en el arreglo. Fíjate que en el ejemplo todos los elementos desde esa posición se mueven a la derecha.

Eliminando elementos

Para eliminar elementos de un arreglo utilizas el método `splice`. Por ejemplo:
```
var array = ["Pedro", "Germán", "Diana"];
array.splice(1, 1); // ["Pedro", "Diana"]
```
El método `splice` recibe uno o dos argumentos cuando quieres eliminar elementos: el índice del elemento que quieres eliminar y la cantidad de elementos a eliminar. Si omites el segundo argumento se eliminarán todos los elementos después del índice que hayas especificado en el primer argumento. Por ejemplo:
```
var array = ["Pedro", "Germán", "Diana"];
array.splice(0); // []
```

Funciones

Eventualmente vas tener algunas líneas de código que necesitan ser ejecutadas varias veces y desde diferentes partes de tu programa. En vez de repetir el mismo código una y otra vez puedes crear una función (también se les conoce como procedimientos o métodos) e **invocarla** cada vez que necesites ejecutar ese trozo de código.

Crea un archivo llamado `functions.js` y escribe lo siguiente:
```
function hello() {
```

```
  console.log("Hola Mundo");
}
```
Para definir una función usamos la palabra reservada `function`, le damos un nombre (en este caso `hello`), abrimos y cerramos paréntesis (`()`). Después abrimos corchetes (`{`), escribimos el cuerpo de la función (el código que queremos ejecutar cuando sea invocada), y por último cerramos los corchetes `}`. Si ejecutamos este código no aparece nada en la pantalla:

```
$ node functions.js
```
Una característica de las funciones es que no se ejecutan hasta que alguien las **invoque**. Modifiquemos nuestro programa para invocarla:

```
function hello() {
  console.log("Hola Mundo");
}

hello(); // acá la estamos invocamos
```
En la última línea la estamos invocando. Si lo ejecutas ahora si debería aparecer "Hola mundo":

```
$ node methods.js
Hola mundo
```

Argumentos

Las funciones pueden recibir cero o más argumentos (o parámetros). Piensa en los argumentos como **variables** que puedes utilizar dentro de la función. Utilizando argumentos podemos hacer una función reutilizable que salude a cualquier persona:

```
function hello(name) {
  console.log("Hola " + name);
}

hello("Germán");
hello("David");
```
Si lo ejecutamos deberías ver lo siguiente:

```
$ node methods.js
Hola Germán
Hola David
```
Los argumentos se definen dentro de los paréntesis al declarar la función y se separan con coma.

Retornando un valor

Opcionalmente puedes retornar un valor desde la función utilizando la palabra clave `return`. Podemos modificar la función `hello` para que en vez de imprimir con `console.log` retorne una cadena de texto:

```
function hello(name) {
```

```
  return "Hola " + name;
}

var g1 = hello("Germán"); // podemos asignar el valor de retorno a una
variable
console.log(g1);

// podemos llamar la función directamente en el parámetro de otra función.
console.log(hello("David"));
```

¿Notas la diferencia? En vez de hacer el `console.log` dentro de la función lo hacemos cuando la invocamos (de lo contrario no aparecería nada en pantalla). En lo posible se recomienda retornar valores en vez de utilizar `console.log` dentro de las funciones. La razón es que retornar un valor hace la función más reutilizable. Ahora podemos utilizar esta función en otros contextos en donde no se utilice `console.log` para imprimir en la línea de comandos, como en una aplicación Web.

El `return` es la última línea que se ejecuta de una función, cualquier código que se encuentre después de esa línea será ignorado. Por ejemplo:

```
function hello(name) {
  return "Hola " + name;
  console.log("Esto nunca se va a imprimir");
}

console.log(hello("Pedro"));
```

Si ejecutas este código deberás ver lo siguiente:

```
$ node functions.js
Hola Pedro
```

La última línea de la función nunca va a ser ejecutada porque la función siempre retorna antes de llegar a ella.

Las partes de una función

Recapitulemos lo que hemos visto hasta ahora. La sintaxis de una función es la siguiente:

```
function <name>([arg1], [arg2], ...) {
  // cuerpo de la función
  return <valor de retorno>;
}
```

Lo que debes tener en cuenta:

- La función se crea con la palabra clave `function`.
- El nombre de la función tiene las mismas reglas de nombramiento que las variables: debe comenzar con `$`, `_` o una letra, y después puede contener letras, dígitos, `_` y `$`.
- La función puede tener cero o más argumentos dentro de los paréntesis que van después del nombre.

- Piensa en los argumentos como variables que puedes utilizar en la función.

- Los valores de esos argumentos se definen cuando invocan la función.
- Cada argumento debe tener un nombre de una variable válido. Recuerda que el nombre de una variable debe comenzar con `$`, `_` o una letra, y después puede contener letras, dígitos, `_` y `$`.
- Puedes retornar un valor desde la función utilizando la palabra clave `return`.
- El valor de retorno debe ser un tipo válido de JavaScript: un número, una cadena de texto, un booleano, un arreglo, etc.
- Puedes almacenar el valor de retorno de una función en una variable o puedes invocar la función como parámetro de otra función.

Cajas negras

En muchas ocasiones es bueno pensar en funciones como cajas negras que reciben unos parámetros de entrada y genera un valor de salida (el valor de retorno).

Ejemplo

Vamos a hacer una función que calcule el indice de masa corporal (IMC). El IMC es una medida que relaciona el peso de una persona con su altura. La formula para calcular el IMC es peso dividido altura al cuadrado:

```
IMC = peso / (altura^2)
```

Traduzcamos eso a código JavaScript. Crea un archivo llamado `bmi.js` (BMI por Body Mass Index) y escribe lo siguiente:

```
function bmi(weight, height) {
  return weight / height ** 2
}

console.log("Tu IMC es: " + bmi(80, 1.8));
```

Si ejecutas el archivo debería mostrar algo así:

```
$ node bmi.js
Tu IMC es: 24.691358024691358
```