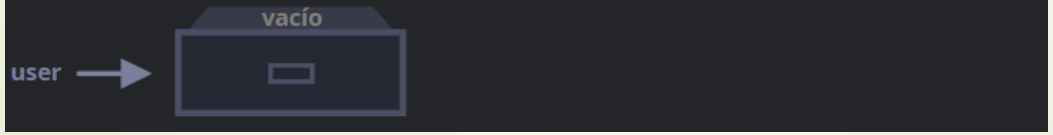
INTRODUCCION A OBJETOS EN JAVASCRIPT

2º DAW - Desarrollo Web en Entorno Cliente

¿Qué son objetos básicos?

- 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.
- Se puede crear un objeto vacío (“gabinete vacío”) utilizando una de estas dos sintaxis (Normalmente se utilizan las llaves {...}. Esa declaración se llama objeto literal):

```
1  let user = new Object(); // sintaxis de "constructor de objetos"
2  let user = {};  // sintaxis de "objeto literal"
```



Literales y propiedades

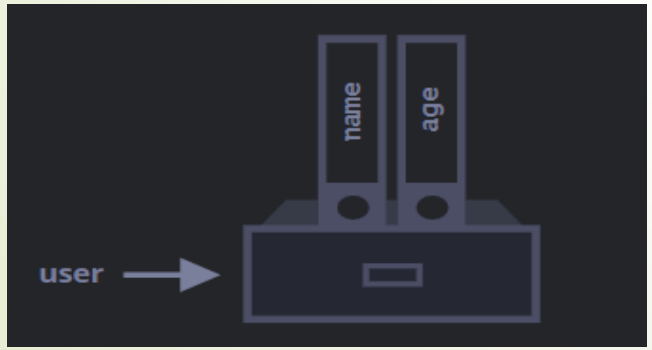
Podemos poner inmediatamente algunas propiedades dentro de {...} como pares “clave:valor”:

```
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:

- La primera propiedad tiene la clave "name" y el valor "John".
- 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”.



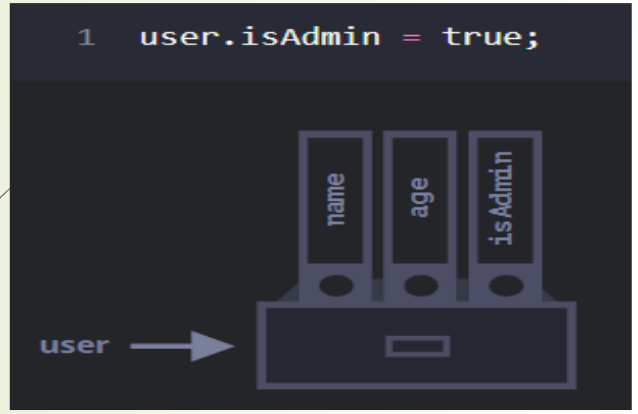
Podemos agregar, eliminar y leer archivos de él en cualquier momento.

Se puede acceder a los valores de las propiedades utilizando la notación de punto:

```
1  // Obteniendo los valores de las propiedades del objeto:
2  alert( user.name ); // John
3  alert( user.age ); // 30
```

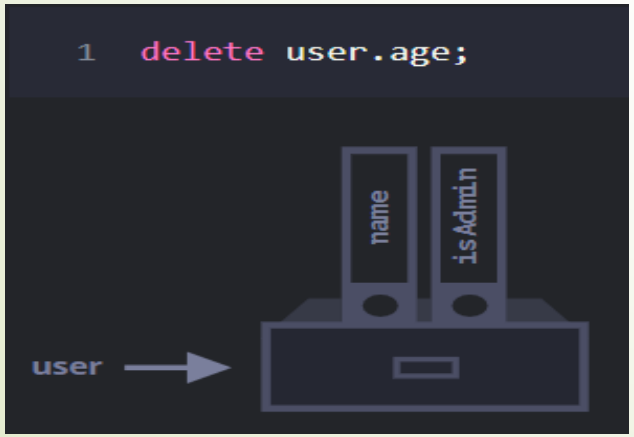
El valor puede ser de cualquier tipo. Agreguemos uno booleano:

```
1  user.isAdmin = true;
```



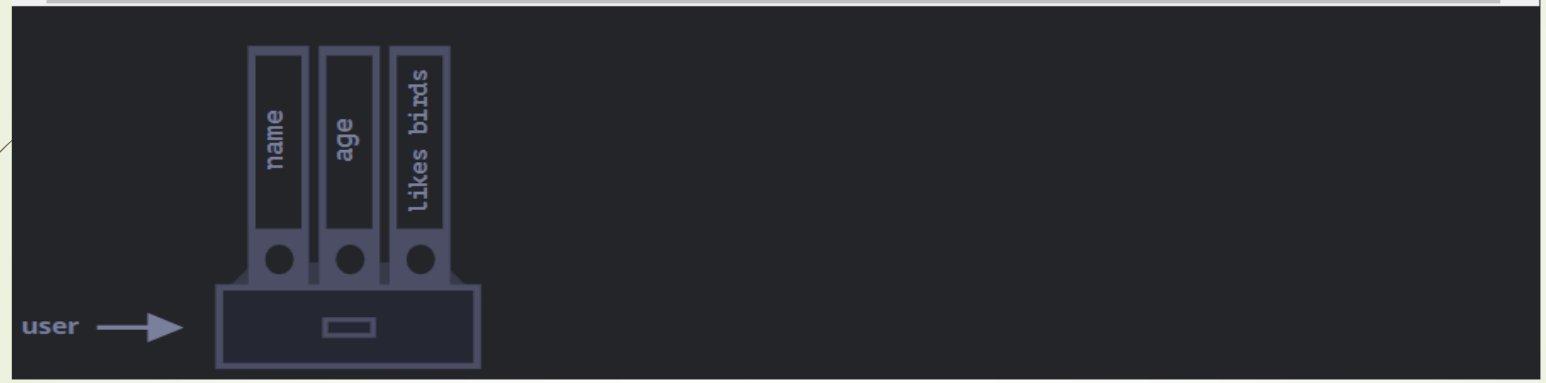
Para eliminar una propiedad podemos usar el operador delete:

```
1  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:

```
1  // Esto nos daría un error de sintaxis
2  user.likes birds = true
```

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:

```
1  let user = {};
2
3  // asignando
4  user["likes birds"] = true;
5
6  // obteniendo
7  alert(user["likes birds"]); // true
8
9  // eliminando
10 delete user["likes birds"];
```

Ahora todo está bien. Nota que el string dentro de los corchetes está adecuadamente entre comillas (cualquier tipo de comillas servirían).

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;
```

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 tán 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
```

Como podemos ver en el código, se ignora la asignación de un valor primitivo 5.

La prueba de propiedad existente, el operador "in"

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 propie
```

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
```

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 prop

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”.

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.

...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
}
```

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.
- 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

Escribe el código, una línea para cada acción:

Crea un objeto `user` vacío.

Agrega la propiedad `name` con el valor `John`.

Agrega la propiedad `surname` con el valor `Smith`.

Cambia el valor de `name` a `Pete`.

Remueve la propiedad `name` del objeto.

Verificar los vacíos

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.

Multiplicar propiedades numéricas por 2

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);

// Después 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.

Referencias de objetos y copias

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:

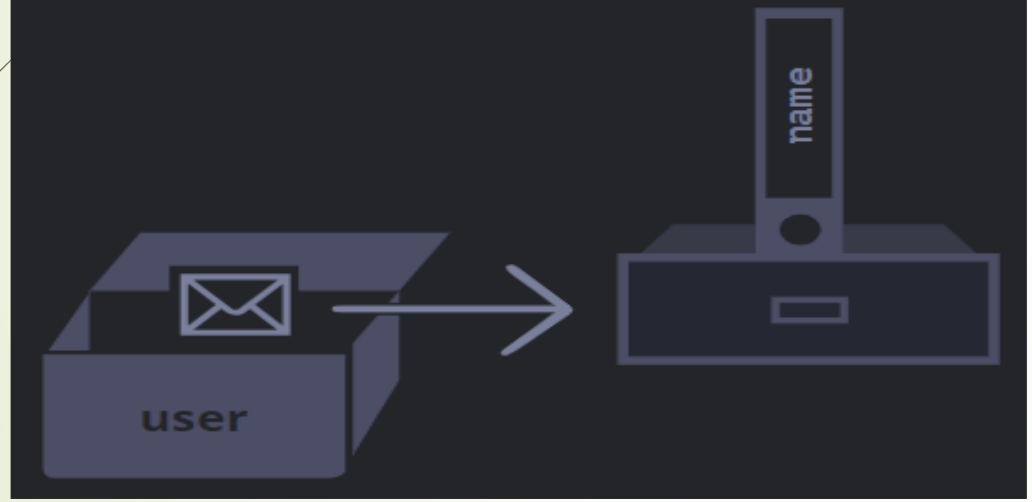
```
1  let message = "Hello!";
2  let phrase = message;
```

Como resultado tenemos dos variables independientes, cada una almacenando la cadena `"Hello!"`.

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:

```
1  let a = {};
2  let b = a; // copia la referencia
3
4  alert( a == b ); // true, verdadero. Ambas variables hacen referencia al mismo objeto
5  alert( a === b ); // true
```

Y aquí dos objetos independientes no son iguales, aunque se vean iguales (ambos están vacíos):

```
1  let a = {};
2  let b = {}; // dos objetos independientes
3
4  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.

NOTA: Un efecto importante de almacenar objetos como referencias es que un objeto declarado como const puede ser modificado.

Clonación y mezcla, Object.assign

Entonces copiar una variable de objeto crea una referencia adicional al mismo 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

// copiemos 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 mimo 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 propiedadas 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 tiene 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 pasarla 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).

Método del objeto “this”

Los objetos son creados usualmente para representar entidades del mundo real, como usuarios, órdenes, etc.:

```
let user = {
  name: "John",
  age: 30
};
```

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().

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!
```

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 lineas:

```
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.

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.
- Tened 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

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?
```

Crea una calculadora

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.

Encadenamiento

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.

Constructor, operador “new”

La 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:

- Son nombradas con la primera letra mayúscula.
- Sólo deben ejecutarse con el operador "new".

Por ejemplo:

```
1  function User(name) {
2    this.name = name;
3    this.isAdmin = false;
4  }
5
6  let user = new User("Jack");
7
8  alert(user.name); // Jack
9  alert(user.isAdmin); // false
```

Cuando una función es ejecutada con new, realiza los siguientes pasos:

- Se crea un nuevo objeto vacío y se asigna a this.
- Se ejecuta el cuerpo de la función. Normalmente se modifica this y se le agrega nuevas propiedades.
- 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 re-utilizables.

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.

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:

```
1  function User() {
2    alert(new.target);
3  }
4
5  // sin  "new":
6  User(); // undefined
7
8  // con  "new":
9  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); // ...Agregaré 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 la 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.

Métodos en constructor

Utilizar constructor de funciones para crear objetos nos da mucha flexibilidad. La función constructora 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.

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.

TAREAS

Crear nueva Calculadora

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.

Crear nuevo Acumulador

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.

Encadenamiento opcional '?.'

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.streeten un usuario sin dirección obtendremos un error:

```
1  let user = {}; // usuario sin propiedad "address"
2
3  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.

```
1  // Error si el resultado de querySelector (...) es null
2  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:

```
1  let user = {};
2
3  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. 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
```

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.

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 ?.:

```
1  let user = {}; // El usuario no tiene dirección
2
3  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:

```
1  let html = document.querySelector('.elem')?.innerHTML; // será undefined si no existe e
```

Leer 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 ?..

NOTA: Si por ejemplo no hay una variable user declarada previamente, entonces user?.anything provocará un error.

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:

```
1  let user = null;
2  let x = 0;
3
4  user?.sayHi(x++); // no hay "user", por lo que la ejecución no alcanza a sayHi ni a x++
5
6  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
```

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.
- Aun 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.