

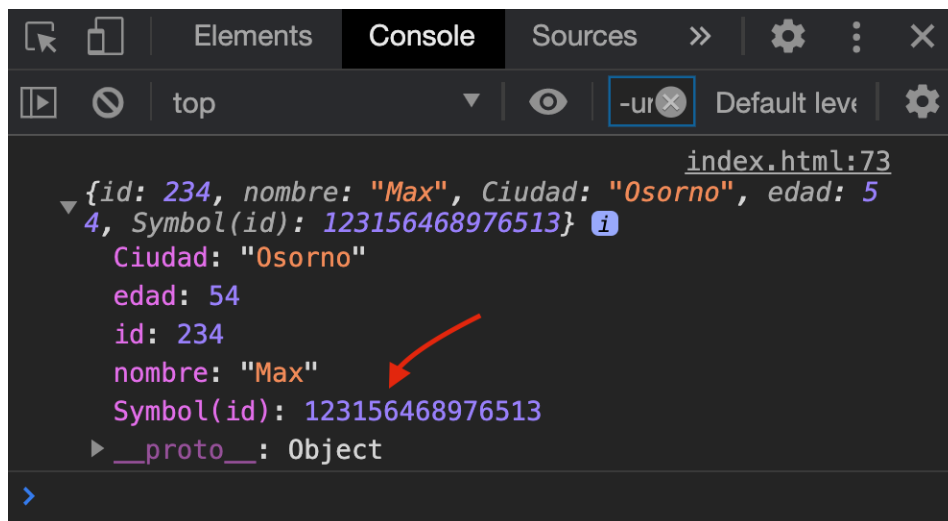
Ahora, vamos a poner en práctica la utilidad de los **Symbols** como identificadores de propiedades de un objeto. Para hacerlo, crearemos el siguiente objeto:

```
1 // creamos el objeto:
2 let usuario = {
3     id: 234,
4     nombre: 'Max',
5     Ciudad: 'Osorno',
6     edad: 54
7 };
```

Si bien nuestro objeto ya tiene un atributo **id**, a continuación, implementaremos un símbolo que nos permite asignarle un **id** completamente único:

```
1 // Creamos el Symbol:
2 const idSimbolo = Symbol('id');
3
4 // Asignamos el Symbol como una propiedad del objeto:
5 usuario[idSimbolo] = 123156468976513;
6 console.log(usuario)
```

Al haber hecho esto, podemos apreciar el símbolo incorporado al objeto en nuestra consola:



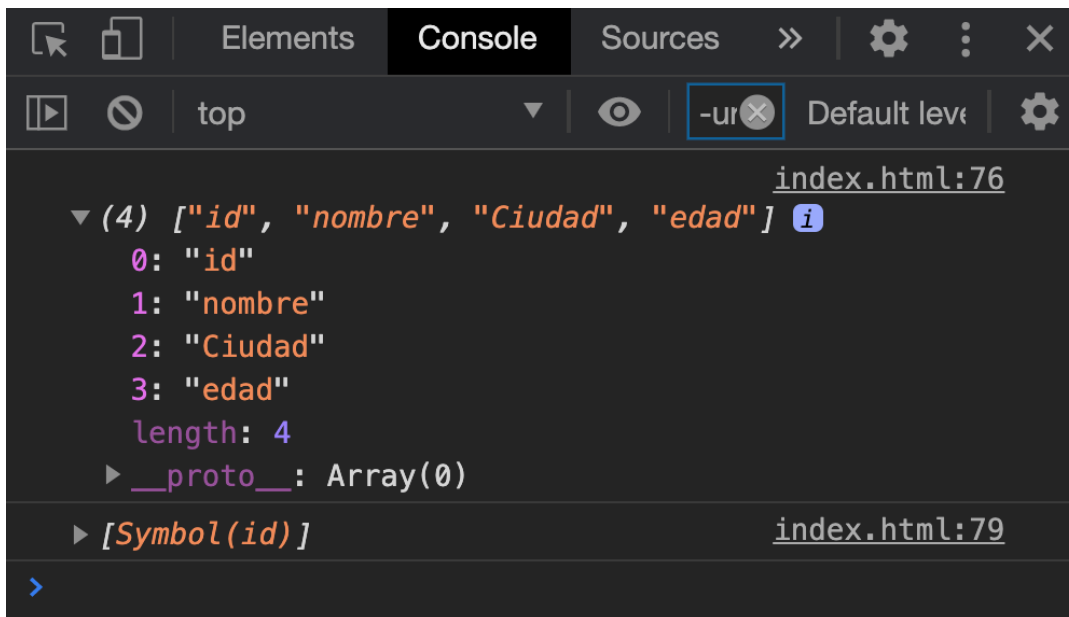
Ahora bien, los símbolos no son incorporados de manera visible en nuestro objeto, esto lo sabemos porque no aparece al ver todas las propiedades del objeto usuario:

```
1 // El símbolo no aparece completamente en nuestro objeto
2 console.log(Object.getOwnPropertyNames(usuario));
```

Pero, mediante el siguiente código, si podemos visualizar los símbolos que contiene:

```
1 //Pero, si podemos ver el símbolo:
2 console.log(Object.getOwnPropertySymbols(usuario));
```

Como resultado, tenemos lo siguiente en nuestra consola:



Si bien lo podemos notar, los símbolos nos sirven para identificar las propiedades de nuestros objetos con valores únicos.

Ahora, vamos a analizar el uso de symbols en otro contexto de aplicación: la representación de conceptos.

A continuación, plantearemos una función que representa el comportamiento de un semáforo:

```
1 const VERDE = 'verde';
2 const AMARILLO = 'amarillo';
3 const ROJO = 'rojo';
4 const manzana = 'rojo';
```

```
5
6 function semaforo(color) {
7     switch (color) {
8         case ROJO:
9             return 'Frena el auto'
10            break;
11        case AMARILLO:
12            return 'Reduce la velocidad'
13            break;
14        case VERDE:
15            return 'Adelante'
16            break;
17        default:
18            console.log(';Eso no es un color!')
19            break;
20    }
21 }
```

¿Te fijaste que tenemos declarado una variable `manzana`? Ésta nos ayudará a entender la importancia de los **Symbols** para representar conceptos, pues si la utilizamos en nuestra función `semaforo()`, veremos que no arroja ningún error. De hecho, como tiene el mismo valor que la variable `ROJO`, nuestra función no diferencia entre uno y el otro.

```
> semaforo(manzana)
< "Frena el auto"
> semaforo(ROJO)
< "Frena el auto"
>
```

Podemos evitar este tipo de situaciones utilizando símbolos. A continuación, vamos a modificar las constantes para que sean **Symbols**.

```
1 const VERDE = Symbol('verde');
2 const AMARILLO = Symbol('amarillo');
3 const ROJO = Symbol('rojo');
4 const manzana = Symbol('rojo');
```

De esta forma, la constante `ROJO` no es para nada igual con la constante `manzana`, evitando la mala representación de conceptos. Esto aplica también a las demás constantes de colores. Si pasamos `ROJO` y `manzana` a nuestra función, tendremos el comportamiento esperado:

```
> semaforo(ROJO)
< "Frena el auto"
> semaforo(manzana)
  ¡Eso no es un color!                                index.html:76
< undefined
> |
```

De esta forma queda demostrada la utilidad de los símbolos dentro de nuestros desarrollos con ES6.

EXERCISE 2: PROXY Y EL OBJETO REFLECT

Un `proxy` se puede traducir a “representante”, y esto se debe a su naturaleza de proporcionar un objeto que actúa como sustituto de uno de servicio real utilizado por un cliente. Esta es una definición generalizada de lo que es un proxy, pero también se puede describir como un objeto que envuelve a otro, e intercepta las operaciones fundamentales del de destino, a modo de intermediario, permitiéndonos redefinir operaciones fundamentales para ese objeto.

Esto nos permite implementar una variada gama de funcionalidades a nuestros desarrollos. Ahora, revisemos como son puestos en práctica los `proxies`.

Un `Proxy` se crea con dos parámetros:

1. `target` (objetivo): el objeto original al que se desea aplicar un proxy.
2. `handler` (manejador): un objeto que define qué operaciones serán interceptadas, y cómo redefinirlas.

La sintaxis es la siguiente, instanciando un nuevo constructor `Proxy`, y pasándole por parámetro los valores `target` y `handler`:

```
1 // sintaxis:  
2 var p = new Proxy (target, handler);
```

A continuación, vamos a plantear un ejemplo en donde primero desarrollaremos el **handler**, y posteriormente el **target**.

```
1 var manejador = {  
2     get(target, key) {  
3         return key in target ? target[key] : 'no existe en el  
4 objeto'  
5     }  
6 }
```

Aquí proporcionamos una implementación del controlador **get()**, que intercepta los intentos de acceder a las propiedades en el destino. Éste recibe por parámetro un **target** (objeto), y un **key** (llave que corresponde a una propiedad de un objeto). Luego, en la sentencia de retorno, establece mediante un operador ternario que: si existe la llave buscada en el objeto especificado, entonces retorna el valor de dicha propiedad; y si no existe, debe retornar el mensaje: “no existe en el objeto”.

Ya que hemos configurado nuestro **handler** o “manejador”, ahora nos queda crear un nuevo **proxy**:

```
1 var p = new Proxy({}, manejador);  
2 p.a = 1;  
3 p.b = 'hola'
```

En este caso, recibirá por parámetro un objeto vacío, y el manejador que recién configuramos. Además, establecemos en **p** los atributos **a** y **b**, con sus valores.

Al tratar de mostrar los valores existentes y los no existentes en nuestra consola, podremos apreciar como el **handler** maneja nuestras interacciones con los atributos del objeto.

```
1 console.log(p.a, p.b);  
2 console.log('c' in p, p.c);
```

En la consola aparece lo siguiente:

```
1 "hola"  
false "no existe en el objeto"  
>
```

De esta forma, podemos comprobar que el **handler** intervino cuando nosotros tratamos de acceder a nuestro objeto, dado que arrojó el mensaje de error como resultado de la evaluación del operador ternario.

Cabe destacar que los manejadores a veces se denominan **trampas**, presumiblemente porque *atrapan* llamadas al objeto de destino. Existen 13 distintas que podemos utilizar con los **Proxy**, como recomendación, puedes investigar más al respecto en el siguiente [enlace](#).

Ya que hemos aprendido cómo funcionan los **proxies** (*plural de proxy*), seguiremos con otro ejemplo para profundizar nuestro conocimiento. Vamos a plantear el siguiente, en donde configuramos un **handler** para validar que la propiedad **"edad"** de nuestro objeto **Persona**, siempre cumple con las condiciones que deseamos.

```
1 let validador = {  
2   set: function(objeto, propiedad, valor) {  
3     // Solo validará la propiedad 'edad':  
4     if(propiedad === 'edad') {  
5       // Aquí validamos de que solo sea un número:  
6       if(typeof valor !== 'number' ||  
7 Number.isNaN(valor)) {  
8         console.log('Edad debe ser un numero')  
9       }  
10      // Aquí validamos que sea un numero positivo:  
11      if(valor < 0) {  
12        console.log('Edad debe ser un número  
13 positivo')  
14      }  
15    }  
16    // Si cumple con el criterio, se asigna el valor:  
17    objeto[propiedad] = valor;  
18    return true;  
19  }  
20 }  
21 // Nuestro Proxy  
22 let persona = new Proxy({}, validador);
```

```
23 //Tratamos de asignar valores a la propiedad "edad"
24 persona.edad = 'joven'; //Error: solo debe ser numeros
25 persona.edad = -36; //Error: solo numeros positivos
26 persona.edad = 62; //¡Perfecto!
27 console.log(persona.edad)
```

Por consola, podemos apreciar las siguientes intervenciones del **handler** de nuestro **Proxy**:

Edad debe ser un numero	index.html:63
Edad debe ser un número positivo	index.html:67
62	index.html:82
>	

Como se puede observar, los **Proxies** son un valioso elemento nuevo para validar el ingreso correcto de datos en nuestros objetos. Ahora, analizaremos un componente de ES6 que va de la mano con ellos, este es el objeto **Reflect**. En castellano, reflect es *reflejar*, y se deriva del concepto de la reflexión: la capacidad de un programa para manipular variables, propiedades y métodos de objetos en tiempo de ejecución. **Reflect** es el término utilizado para especificar un objeto integrado, lo que simplifica la creación del proxy.

A diferencia de la mayoría de los objetos globales, **Reflect** no es un constructor. Esto significa que no podemos usarlo con el operador `new`, o invocarlo como función, al igual que con los objetos **Math** y **JSON**. Por esta misma razón, todos los métodos del objeto **Reflect** son estáticos.

Para cada método interno como **"set()"** o **"get()"**, que está atrapado por proxy, existe un método que tiene exactamente la misma funcionalidad en **reflect**. Tiene el mismo nombre y argumentos que las trampas de proxy.

Entonces, mientras se utiliza un proxy para **intervenir** en operaciones sobre un objeto, se usa un **reflect** para **reenviar** una operación al objeto original. En nuestro IDE podremos ver los métodos disponibles:


```
Reflect.  
  construct  
  defineProperty  
  deleteProperty  
  get  
  getOwnPropertyDesc... function Reflect.getOwnProperty...  
  getPrototypeOf  
  has  
  isExtensible  
  ownKeys  
  preventExtensions  
  set  
  setPrototypeOf
```

A continuación, se detallan todos los métodos que podemos usar con **Reflect**:

Reflect.apply(): llama a una función con argumentos especificados.

Reflect.construct(): actúa como el operador **new**, pero como una función. Es equivalente a llamar a **new target(... args)**.

Reflect.defineProperty(): es similar a **Object.defineProperty()**, pero devuelve un valor booleano que indica si la propiedad se definió correctamente en el objeto.

Reflect.deleteProperty(): se comporta como el operador **delete**, pero como una función. Es equivalente a llamar a **delete objectName [propertyName]**.

Reflect.get(): devuelve el valor de una propiedad.

Reflect.getOwnPropertyDescriptor(): es similar a **Object.getOwnPropertyDescriptor()**. Devuelve un descriptor de propiedad si esta existe en el objeto, o de lo contrario devuelve **undefined**.

Reflect.getPrototypeOf(): es lo mismo que **Object.getPrototypeOf()**.

Reflect.has(): funciona como el operador **in**, pero como una función. Devuelve un valor booleano que indica si existe o contiene una propiedad (sea propia o heredada).

Reflect.isExtensible(): es lo mismo que **Object.isExtensible()**.

Reflect.ownKeys(): devuelve una matriz de las llaves de las propiedades de un objeto (no heredadas).

Reflect.preventExtensions(): es similar a **Object.preventExtensions()**. Devuelve un booleano.

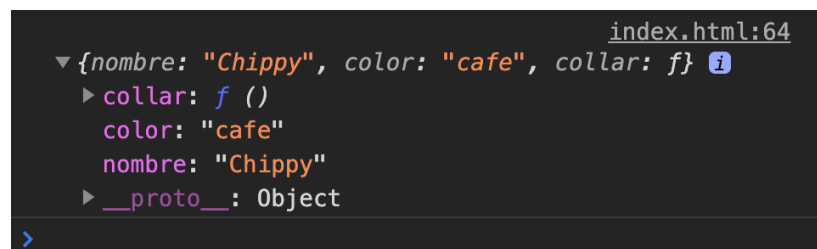
Reflect.set(): asigna un valor a una propiedad, y devuelve un valor booleano que es verdadero si la propiedad se asigna correctamente.

Reflect.setPrototypeOf(): establece el prototipo de un objeto.

A continuación, vamos a realizar un ejemplo en donde usaremos 3 métodos: **has()**, **ownKeys()**, y **set()**.

```
1 const perrito = {  
2     nombre: 'Chippy',  
3     color: 'cafe',  
4     collar: function() {  
5         console.log(`Mi nombre es ${this.nombre}`);  
6     }  
7 }  
8 console.log(perrito)
```

En nuestra consola, nuestro objeto **perrito** se ve así:



```
index.html:64  
▼ {nombre: "Chippy", color: "cafe", collar: f} ⓘ  
  ► collar: f ()  
    color: "cafe"  
    nombre: "Chippy"  
  ► __proto__: Object  
>
```

A continuación, emplearemos la función `has` del objeto `Reflect`, para comprobar si el objeto `perrito` contiene ciertas propiedades. Por parámetro, debemos pasarle el objeto en donde va a buscar, y la propiedad que buscamos:

```
1 //Usamos has para ver si el objeto tiene las propiedades:
2 console.log(Reflect.has(perrito, 'color')); // si la tiene
3 console.log(Reflect.has(perrito, 'edad')); // no la tiene
```

Cuando este método no encuentra la propiedad que buscamos retorna un `false`:

```
true                                index.html:66
false                              index.html:68
>
```

También podemos utilizar el método `ownKeys()` para obtener las llaves de propiedad de nuestro objeto:

```
1 console.log(Reflect.ownKeys(perrito));
```

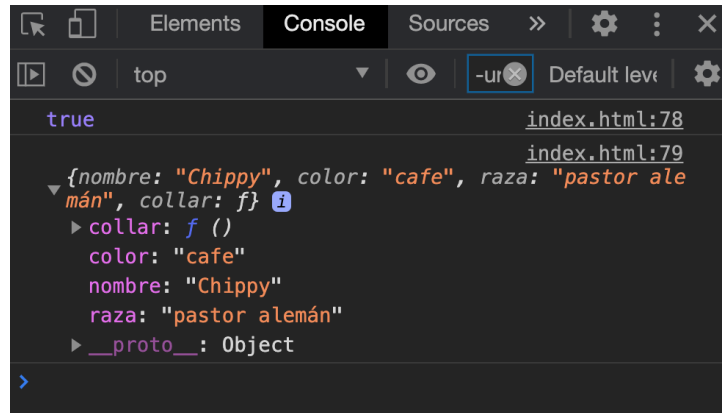
Este método nos retorna lo siguiente:

```
▶ (3) ["nombre", "color", "collar"]  index.html:74
>
```

Cómo podemos ver, este método nos retorna todas las propiedades de nuestro objeto. El último método que revisaremos en el ejemplo nos permitirá asignar una nueva propiedad a dicho objeto, y este es `set()`:

```
1 console.log(Reflect.set(perrito, 'raza', 'pastor alemán'));
2 console.log(perrito)
```

Por parámetro debemos introducir el objeto al cual queremos agregarle una propiedad, luego el nombre de ésta, y posteriormente su valor. A continuación, veremos el resultado de este método por consola:



```
true index.html:78
{nombre: "Chippy", color: "cafe", raza: "pastor alemán", collar: f()} index.html:79
  ▶ collar: f ()
    color: "cafe"
    nombre: "Chippy"
    raza: "pastor alemán"
  ▶ __proto__: Object
```

Cómo podemos apreciar, ahora nuestro objeto **perrito** contiene la propiedad **raza**, con el valor **“pastor alemán”**. De esta forma hemos logrado, de manera exitosa, aprender los detalles acerca del objeto **Proxy** y el objeto **Reflect**.