educación IT
CODE YOUR FUTURE

ReactJS Developer

Módulo 8

Generadores

Generadores en JavaScript

Hasta ahora, hemos aprendido que las funciones en JavaScript y en casi cualquier lenguaje de programación son **bloques de código que tienen un nombre y pueden ser llamadas reiteradas veces a lo largo de nuestro script**.

Estos bloques son un conjunto de instrucciones que se ejecutan **una detrás de la otra** hasta que no haya más instrucciones o hasta que encontremos un "`return`" que devuelve un valor y detiene la ejecución del resto del código.

Las funciones generadoras son denominadas generalmente "**pausables**". Son funciones que **se ejecutan a demanda y que se detienen cada vez que encuentran la palabra `yield`**.

A diferencia de las funciones normales, cuando guardamos una función generadora en una variable, guardamos el objeto **`generator`**, un objeto especial que cumple con el concepto de *iterator*.

Declaración

Las funciones generadoras se declaran de la misma forma que declaramos una función normal pero agregando un * (asterisco) al lado de la palabra **`function`**.

Contamos con las siguientes variantes:

```
function* generador() {}
function * generador(){}
function *generador(){}

const generador = function* () {}
const generador = function * (){}
const generador = function *(){}

class MiClase {
    *generador() {}
    * generador() {}
}

const objeto = {
    *generador() {},
    * generador() {}
}
```

Yield

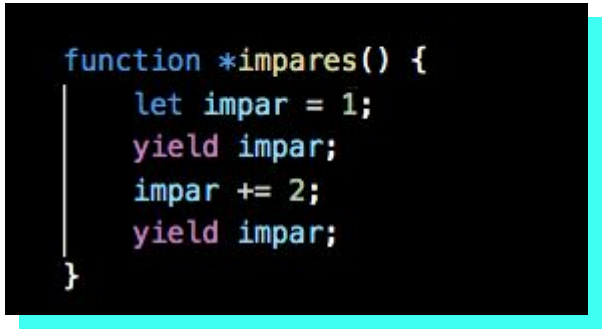
Cuando definimos lo que eran las funciones generadoras, dijimos que las mismas se ejecutan parando o pausando cada vez que encuentran la palabra **`yield`**.

`Yield` funciona, salvando las distancias, **como un `return` pero que no termina la ejecución de la función** sino que **la detiene hasta la próxima vez que se pida que avance**.

```
function *impares() {
    let impar = 1;
    return impar;
    impar += 2;
    return impar;
}
```

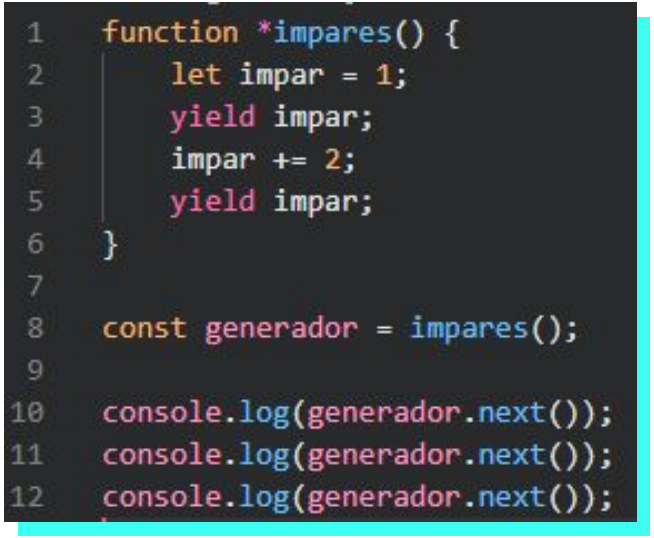
En el ejemplo anterior, la función **`impares`** solo correrá hasta el primer `return`. Cuando la llamemos solo obtendremos el valor **1** y **no podremos obtener nada más**. En el ejemplo de la derecha veremos cómo quedaría con `yield`.

Ahora la función **`impares`** es **generadora**. Cuando vayamos necesitando sus valores la función sabrá **cuándo detenerse y cuándo devolver información**.



Uso

Cuando llamamos a una función generadora para invocar, la misma devuelve un objeto **generador**. Lo normal es guardar este objeto generador en una variable para luego utilizar el método **next()** para ir avanzando entre sus valores:



```
function *impares() {
    let impar = 1;
    yield impar;
    impar += 2;
    yield impar;
}

const generador = impares();

console.log(generador.next());
console.log(generador.next());
console.log(generador.next());
```

Cada vez que llamemos a **`impares()`** estamos generando **una nueva instancia del generador**, por eso tenemos que guardar en una variable esa instancia para poder trabajarla.

El método **`next()` iniciará o continuará la ejecución de la función**. Devuelve un objeto con `value` y done. **`Value`** es **el valor que `yield` tiene**, y **done** es **un `flag` para que sepamos si la función terminó su ejecución o no**.

Vamos a entender qué está pasando.

Cuando llamamos por primera vez a **`next`**, la función **inicia su ejecución** y devuelve el primer valor impar, es decir **`1`**. Como todavía hay más instrucciones, el valor de **`done`** es **`false`**.

Cuando la llamamos por segunda vez, la función va a retomar su ejecución desde donde había quedado, entonces, el **`yield`** termina de ejecutarse (más adelante veremos qué significa), suma 2 a la variable impar y llega hasta el segundo **`yield`** que nos devuelve el valor actual de impar que es 3.

Entonces **`value`** va a ser 3 y **`done`** va a ser **`false`** porque, si bien no quedan más líneas de código por ejecutar, todavía **no terminó de ejecutarse la línea de `yield` propiamente dicha**.

La tercera vez que ejecutemos **`next()`** la función **retoma desde este punto**, finaliza la línea de `yield` y, **como no hay más instrucciones, termina su ejecución**. Devuelve por lo tanto un **`value undefined`** y un **`done true`**.

Return

Podemos utilizar **`return`** para mejorar un poco el comportamiento de nuestra función.

Si hacemos esto ahora, la segunda vez que llamemos a **`next()`** el valor de **done** será **`true`**, ya que se ejecuta un `return` que, por defecto, **finaliza la ejecución de la función**.

```
1  function *impares() {
2      let impar = 1;
3      yield impar;
4      impar += 2;
5      return impar;
6  }
```

Pasando datos al generador

Ya vimos que el método **`Next`** devuelve **el próximo valor que la función me quiere devolver** y **un *flag* para saber si existen más líneas de código para ejecutar o no**.

Nosotros también podemos pasarle valores a esa función generadora como parámetros del método **`next()`**.

Veremos un ejemplo en la próxima slide.

```
const SubscriptionsList = [
    'aperez@gmail.com',
    'llopez@hotmail.com',
    'yuperez@live.com'
];

function *SendSubscription() {
    while(true) {
        const email = yield true;
        yield apiCallSendSubscription(email);
    }
}

const sendSubscription = SendSubscription();

SubscriptionsList.forEach((email) => {
    const currentEmail = sendSubscription.next(currentEmail);
    const apiCallResponse = sendSubscription.next();
});
```

Analicemos el ejemplo anterior:

1. Primero creo **una lista de personajes**. Imaginemos que **esta información,** en realidad **puede venir desde un servidor**.
2. Después **declaro mi función generadora**. Ponemos la generación de información dentro de un ciclo infinito, esto es una práctica muy común.
3. Dentro del ciclo primero **creo un ID al azar y hago `yield` de ese ID**. Cuando se llame al método **`next()`** por primera vez, **va a devolver el valor de ese ID generado**.
4. Luego de esto, se asigna el valor de **`yield ID`** a **`const personaje`**. Es decir, esperamos recibir información desde el exterior de la función. Esto lo vemos en la última línea del script cuando llamamos al método **`next`** pasándole como argumento el personaje en cuestión.
5. Finalmente realizar un **`console.log`** saludando al personaje.

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!