

Code

Issues


Pull requests 1

Actions

Projects

Security

Insights

 master ▾

...

[FT-M3](#) / [07-AsyncAwait](#) /



JJSolari ...

on 8 Apr 

..



AsyncAwait

7 months ago



Generators

13 months ago



README.json

7 months ago



README.md

9 months ago



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

# Async/Await y Generators

## Generators

"Generator Functions are functions that can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances"

## ☰ README.md

podrían detener su ejecución para luego retomar el control más adelante en el flujo del programa. Al volver a la función luego de una "pausa", el contexto sigue siendo el mismo, por lo que sus variables no se van a ver afectadas.

Podrían hacer la analogía de este tipo de funciones con un control remoto en el cual podemos pausar lo que estemos viendo y luego, cuando lo deseemos, volver a darle "play" o incluso dejar de ver lo que estábamos mirando (stop).

## 🔗 Run-to-completion Model

Lo que usualmente conocemos como "Run-to-completion Model" es el paradigma sobre el cual se basan todas las funciones de JS, a excepción de los generators. La idea principal consiste en que toda función va a ejecutarse por completo hasta llegar a su última instrucción o punto de salida, ya sea por:

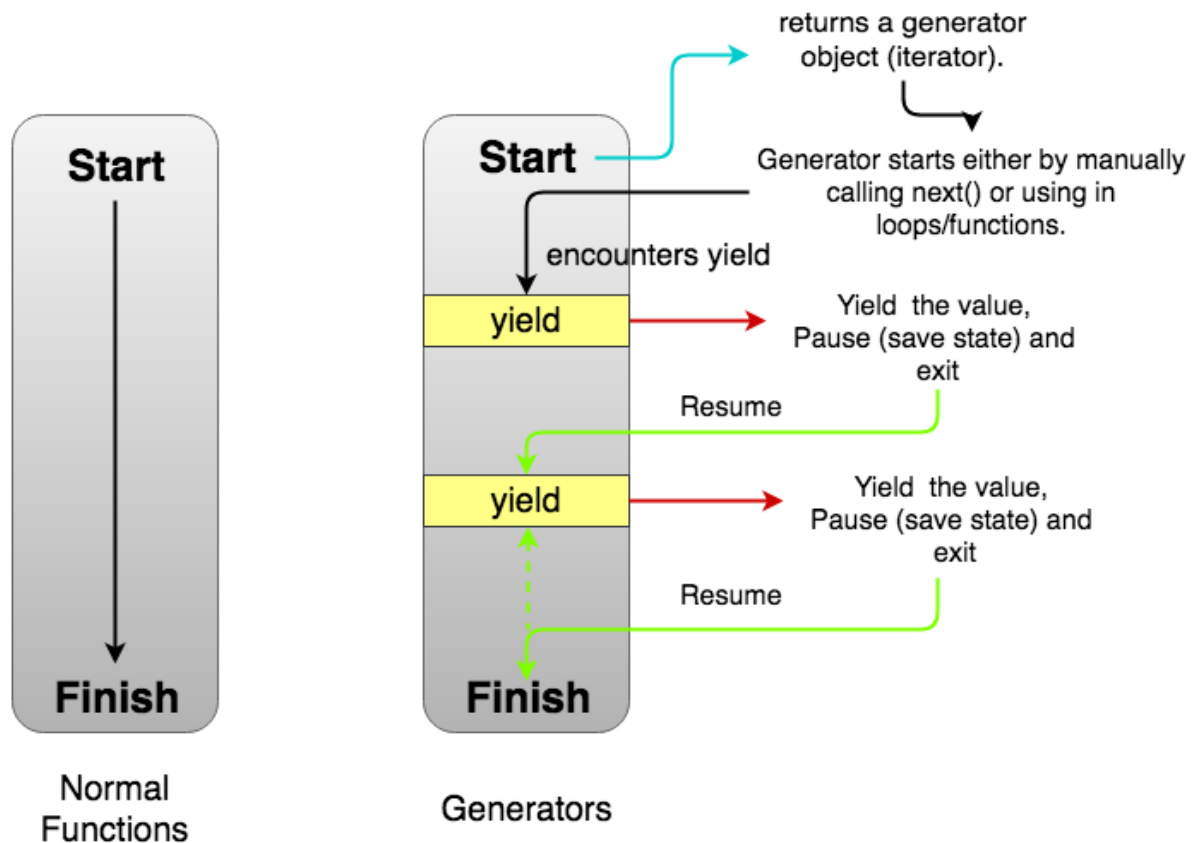
- Return statement
- Error thrown
- Fin de instrucciones (implicit undefined return)

## Generators Model

Por su parte el modelo en que se basan los generators previamente explicado consta de

- **Ejecución inicial:** retorna un "generator object" que va a ser nuestro iterador
- **Comienzo:** comienza a ejecutarse el generator cuando se invoca la instrucción `next` o a través de algún ciclo de iteración
- **"Pausas":** puede tener una cantidad infinita de puntos de pausa en los cuales se guarda su contexto y se cede nuevamente el control a la función o programa que invocó al generator
- **Resume:** luego de cada pausa es posible que el generator tome nuevamente el control para continuar con sus operaciones
- **Fin:** una vez completadas o pasados todos los puntos de pausa el generator llega a su fin

La siguiente imagen muestra en detalle las diferencias entre ambos modelos recién expuestas:



## Generators Syntax

```
function* generatorShowInstructors() {
  console.log("Iniciando generator function");
  yield "Franco";
  yield "Toni";
  console.log("Generator function terminada");
}

var generatorObject = generatorShowInstructors();

generatorObject.next();
```

En primer lugar para poder definir un generator debemos utilizar la sintaxis `function*` lo que va a indicar que la misma debe retornar un `Generator Object`.

Una vez definida dicha función podemos invocarla al igual que cualquiera de las funciones que ya conocíamos de antes. La particularidad que tienen los generators es que no se va a ejecutar el cuerpo de la función una vez que hagamos la invocación de la misma, en el ejemplo: `generatorShowInstructors()` sino que simplemente nos va a devolver como mencionamos antes un `Generator Object` que nos va a permitir luego "controlar" esta función.

Una vez obtenido nuestro iterador podemos invocar las veces que queramos al método `next()` que se va a encargar ahora si de ejecutar el cuerpo del generator previamente invocado hasta llegar a un "punto de pausa" que se van a identificar con la palabra reservada `yield`.

Para que quede más claro veamos como sería el flow de ejecución del ejemplo de arriba

```
var generatorObject = generatorShowInstructors();
// En este punto todavía no se ejecutó nada del cuerpo de la función generatorShowIns
// Simplemente tenemos guardado en la variable generatorObject el justamente Genereto
// que nos va a servir como iterador y poder controlar el generator

var firstNext = generatorObject.next();
// La primera vez que ejecutamos el método next sobre el iterador vamos a ejecutar la
// de la función generatorShowInstructors hasta encontrar el primer yield

function* generatorShowInstructors() {
  console.log("Iniciando generator function"); // <-- Se ejecuta
  yield "Franco";                             // <-- Se pausa le ejecución
  yield "Toni"
  console.log("Generator function terminada");
}

// Si observamos que quedó almacenado en firstNext obtendremos un objeto de la sigueie
{
  done: false,
  value: "Franco"
}
// Es decir tenemos información sobre el estado de nuestro generator.
// Por un lado la propiedad done nos indica que el generator aún no ha finalizado
// y por otro lado el value corresponde con el valor actual que tiene el generator en
// este punto de pausa que se corresponde con el valor que se coloqué después de la p
// yield

// Si volvemos a ejecutar next, ¿qué creen que sucederá?
var secondNext = generatorObject.next();

function* generatorShowInstructors() {
  console.log("Iniciando generator function");
  yield "Franco";
  yield "Toni"                                // <-- Avanza hasta acá y se pausa le
  console.log("Generator function terminada");
}

// Lo que sucede es que vuelve a tomar el control el generator y vuelve a avanzar has
// punto de pausa o hasta su finalización (lo que ocurra primero)
// Por lo que si observamos ahora que contiene secondNext:
{
  done: false,
  value: "Toni"
}
```

```
// Otra vez :D
var thirdNext = generatorObject.next();

// Obtenemos:
{
  done: true,
  value: undefined
}

// Que nos está indicando que efectivamente el generator llegó a su fin ya que done e
```

## Yield vs Return

Si bien a simple vista pueden parecer equivalentes tanto el `yield` como el `return`, no lo son:

- Yield: se encarga de establecer los "puntos de pausa" por lo que al llegar a un `yield` se pausa el generator y se retorna un objeto como el que vimos en el ejemplo de arriba que indica el estado actual del generator
- Return: una vez que se alcanza un `return` statement dentro de un generator, se finaliza su ejecución seteando el valor de "done" del objeto devuelto en `true`.

```
function* generatorUnreachableValue() {
  console.log("Iniciando generator function");
  yield "First reachable value";
  yield "Second reachable value";
  return "Return executed";
  yield "Unreachable value"
}
```

```
var generatorObject = generatorUnreachableValue();
```

```
generatorObject.next(); // <-- {done: false, value: "First reachable value"}
generatorObject.next(); // <-- {done: false, value: "Second reachable value"}
generatorObject.next(); // <-- {done: true, value: "Return executed"}
generatorObject.next(); // <-- {done: true, value: undefined}
```

```
// Vamos a poder seguir ejecutando el método "next" sobre generatorObject pero como
// ya finalizó vamos a seguir obteniendo siempre el mismo resultado: {done: true, va
// En este ejemplo la instrucción `yield "Unreachable value"` nunca va a ser ejecut
```

## Infinite Generator

Es posible también definir generators cuyo flow de ejecución sea infinito que no quiere decir que va a estar ejecutandose en background por tiempo indefinido sino que nosotros somos quienes tenemos el control y podríamos en caso de querer ejecutar el método `next` infinitas veces.

A continuación definiremos un generador de números naturales:

```
function* naturalNumbers() {  
  let number = 1;  
  while(true) {  
    yield number;  
    number = number + 1;  
  }  
}  
  
var generatorObject = naturalNumbers();  
  
generatorObject.next(); // <-- Retorna {done: false, value: 1}  
generatorObject.next(); // <-- Retorna {done: false, value: 2}  
generatorObject.next(); // <-- Retorna {done: false, value: 3}  
generatorObject.next(); // <-- Retorna {done: false, value: 4}  
  
// En value tendremos la secuencia de números naturales que queríamos
```

## Homework 1

---

Completa la tarea descrita en el archivo [README](#)

## Async/Await

---

Las funciones asíncronas o async functions nos van a permitir, como su nombre lo indica, definir código asíncrono con una sintaxis distinta a la que veníamos utilizando con las promesas por lo que no tendremos que encadenarlas nosotros mismos de forma explícita.

### Sintaxis

Para utilizar este tipo de funciones debemos definirlas con una sintaxis en particular:

```
async function asyncCall() {  
  const result = await resolveAfter2Seconds();  
}
```

La palabra `async` es la que va a informarle al intérprete que se trata de una `async function` y nos va a permitir hacer uso de la palabra reservada `await` en el cuerpo de dicha función. Como su nombre nos sugiere, lo que va a ocurrir cuando la ejecución del programa se tope con un `await` es que se detendrá la ejecución de esa función de forma momentánea hasta que la función o instrucciones que se encuentren a la derecha de dicha palabra finalicen.

## Basic Flow

A continuación analizaremos un pequeño ejemplo para comprender mejor el flow de estas nuevas funciones:

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling'); // <-- Se ejecuta luego de la invocación de asyncCall()
  const result = await resolveAfter2Seconds(); // <-- Detiene la ejecución de asyncCa
  console.log(result); // <-- No se va a ejecutar hasta que la línea anterior finalic
}

asyncCall()
```

## Return

Una particularidad de las `async functions` es que siempre retornan una promesa.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Promesa resuelta!');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('Iniciando asyncCall');
  const result = await resolveAfter2Seconds();
  console.log(result); // <--- Va a loguear "Promesa resuelta!"
}

var p1 = asyncCall(); // p1 --> Va a ser una promesa la cual dependiendo el momento e
```

En el caso de que la async function no tenga un return statment, la promesa que devolverá tendrá un value undefined , por ejemplo el caso anterior almacena en p1 lo siguiente:

```
Promise --> { state: "fulfilled", value: undefined }
```

En cambio si tuvieramos un return statment, por ejemplo algo así:

```
async function asyncCall() {  
  console.log('Iniciando asyncCall');  
  const result = await resolveAfter2Seconds();  
  console.log(result);  
  return "Franco";  
}
```

```
var p2 = asyncCall();
```

Ahora p2 será una promesa cuyo valor de resolución será "Franco":

```
Promise --> { state: "fulfilled", value: "Franco" }
```

Como ya sabíamos de cuando estudiamos promesas, las mismas pueden resolverse o rechazarse por lo que en caso de success como vimos en el ejemplo de arriba, la promesa retornada se va a resolver al valor retornado por la función asíncrona y en el caso de error , la promesa retornada se va a rechazar con la excepción lanzada por la función asíncrona.

Para más ejemplos que incluyen todos los distintos casos posibles ver la demo

demoRetunValue.js

## Yielding Control

Algo importante de comprender es como continua el flow del resto del programa/aplicación una vez que se encuentra con un await dentro de una async function. Como ya hemos mencionado antes, lo que sucede es que en ese instante se le retorna el control a la función o punto del programa desde donde se había invocado a la async function y va a continuar su flow normalmente.

Para terminar de comprenderlo analicemos el siguiente ejemplo:

```
async function showInstructors() {  
  const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco')  
  console.log(instructor1);  
  const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')  
  console.log(instructor2);  
}
```



```
function henryAwait() {  
  console.log("¿Quienes son los intstructores de Henry?");  
  showInstructors();  
  console.log("Gracias vuelvan pronto");  
}  
  
henryAwait()  
console.log("FIN");
```

¿Cuál será el orden de ejecución de el código de arriba?

Luego de definir tanto ambas funciones, una asíncrona y la otra no, se ejecuta henryAwait que no es asíncrona. Hasta ahí todo normal:

```
function henryAwait() {  
  console.log("¿Quienes son los intstructores de Henry?"); // <-- Loguea "¿Quienes so  
  showInstructors(); // <-- Invoca a showInstructors (Cede el control)  
  console.log("Gracias vuelvan pronto"); // <-- Aún no se invocó...  
}
```

Veamos ahora que sucede al ingresar a showInstructors:

```
async function showInstructors() {  
  const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco  
  console.log(instructor1);  
  const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')  
  console.log(instructor2);  
}
```

Como ahora nuevamente el control lo tiene henryAwait, continua con su flow normal:

```
function henryAwait() {  
  console.log("¿Quienes son los intstructores de Henry?");  
  showInstructors(); // <-- Se había quedado acá, ahora avanza a la siguiente...  
  console.log("Gracias vuelvan pronto"); // <-- Loguea "Gracias vuelvan pronto"  
}
```

Y ahora como henryAwait finalizo continua hacía:

```
henryAwait() // <-- Ya finalizó, avanza...  
console.log("FIN"); // <-- Logua "FIN"
```

Luego de todo esto recién ahí, y si la promesa donde se había pausado `showInstructors` ya finalizó, vuelve a tomar el control y continua con las sentencias que quedaron sin ejecutarse:

```
async function showInstructors() {  
  const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco'  
  console.log(instructor1); // <-- Loguea "Franco"  
  const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')  
  console.log(instructor2); // <-- Una vez finalizada loguea "Toni"  
}
```

Si quisieramos que el orden de ejecución sea:

1. ¿Quiénes son los instructores de Henry?
2. Franco
3. Toni
4. Gracias vuelvan pronto
5. FIN

¿Cómo deberíamos modificar el código previo?



```
async function showInstructors() {  
  const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco'  
  console.log(instructor1);  
  const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')  
  console.log(instructor2);  
}  
  
async function henryAwait() {  
  console.log("¿Quiénes son los instructores de Henry?");  
  await showInstructors();  
  console.log("Gracias vuelvan pronto");  
}  
  
await henryAwait()  
console.log("FIN");
```

## Ventajas

- El código suele ser más prolijo y similar a código sincrónico:

```
const readFilePromise = (archivo) => {
  promisifiedReadFile(archivo)
    .then(file => {
      console.log("Log promise file: ", file);
      return "Lectura exitosa";
    });
}

const readFileAsync = async(archivo) => {
  console.log("Log async file: ", await promisifiedReadFile(archivo));
  return "Lectura exitosa";
}
```

Para más detalle ver la demo `demoCleanCode.js`

- Permite manejar tanto errores de código sincrónico como asincrónico en un mismo lugar (try/catch)

```
const readFileAsync = async(archivo) => {
  try {
    console.log("Log async file: ", await promisifiedReadFile(archivo));
    return "Lectura exitosa";
  } catch (err) {
    console.log("Error unificado: ", err);
  }
}
```

Para más detalle ver la demo `demoErrorHandler.js`

## Desventajas

- El código suele ser más prolijo y similar a código sincrónico. ¿¿¿Qué??? ¿No les habíamos dicho hace un par de líneas que era una ventaja esto?

Si, no estamos locos, esto puede ser un arma de doble filo, porque al maquillar código asíncrono haciendolo parecer como sincrónico muchas veces solemos utilizarlo de forma incorrecta y terminando sin entender el flow del programa. Recuerden el ejemplo que hicimos más arriba de `showInstructors` y verán que si no se comprende bien como funcionan `async` y `await` puede llevar a grandes confusiones.

Por eso mismo, ustedes que ya entienden a la perfección el funcionamiento de promesas si comprender en el fondo que es lo que está ocurriendo y no pensar que es simplemente 'magia'.

Por último podríamos pensar que Async/Await tomó y combinó las ideas de Generators junto con Promesas: ***Async/Await = Generators + Promises***

## Homework 2

---

Completa la tarea descrita en el archivo [README](#)

---