

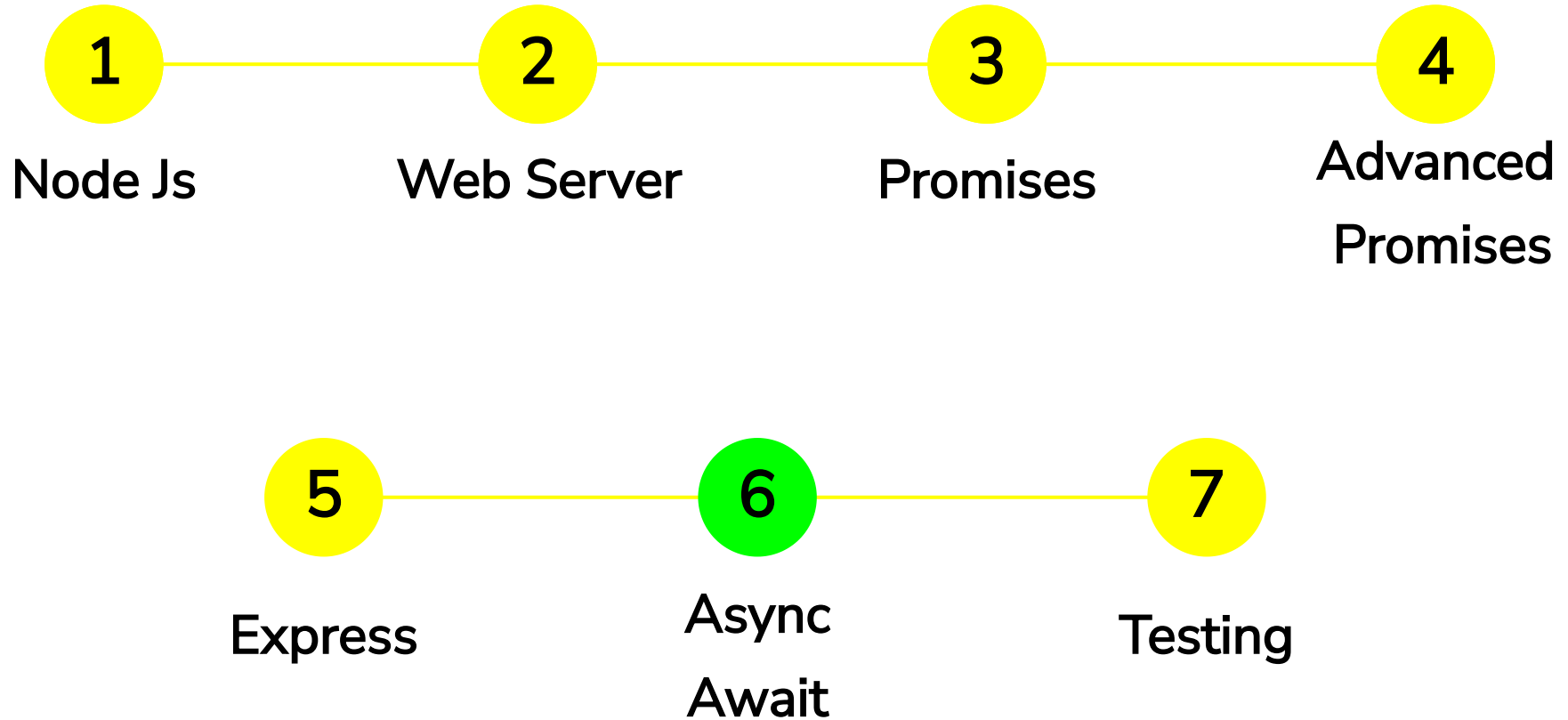
HENRY

A bright yellow beam of light originates from the left edge of the frame and points towards the letter 'R' in the word 'HENRY'. The beam is wider on the left and tapers as it moves to the right. The word 'HENRY' is written in a bold, black, sans-serif font.

Lecture

Async Await

Contenido M3



Recapitulemos...

1

callback -> Una de las primeras maneras que los desarrolladores usaban para manejar la información asincrónica que consta de una función anidada dentro de otra función como argumento. Esto provoca los *callbacks hell*.

2

Promesas -> Llegaron con ES6 a solucionar la problemática de los *callbacks hell*. Una promesa es un objeto que representa el resultado de una operación asincrónica que se resuelve con un valor o se rechaza con una razón.



OBJETIVO DE LA CLASE



Aprender sobre las funciones generadoras y conocer otra forma de manejar las promesas con Async Await.



¿Qué veremos hoy?

- ✓ Event Loop
- ✓ Generator Functions
- ✓ Async Await



Espacio de interacción

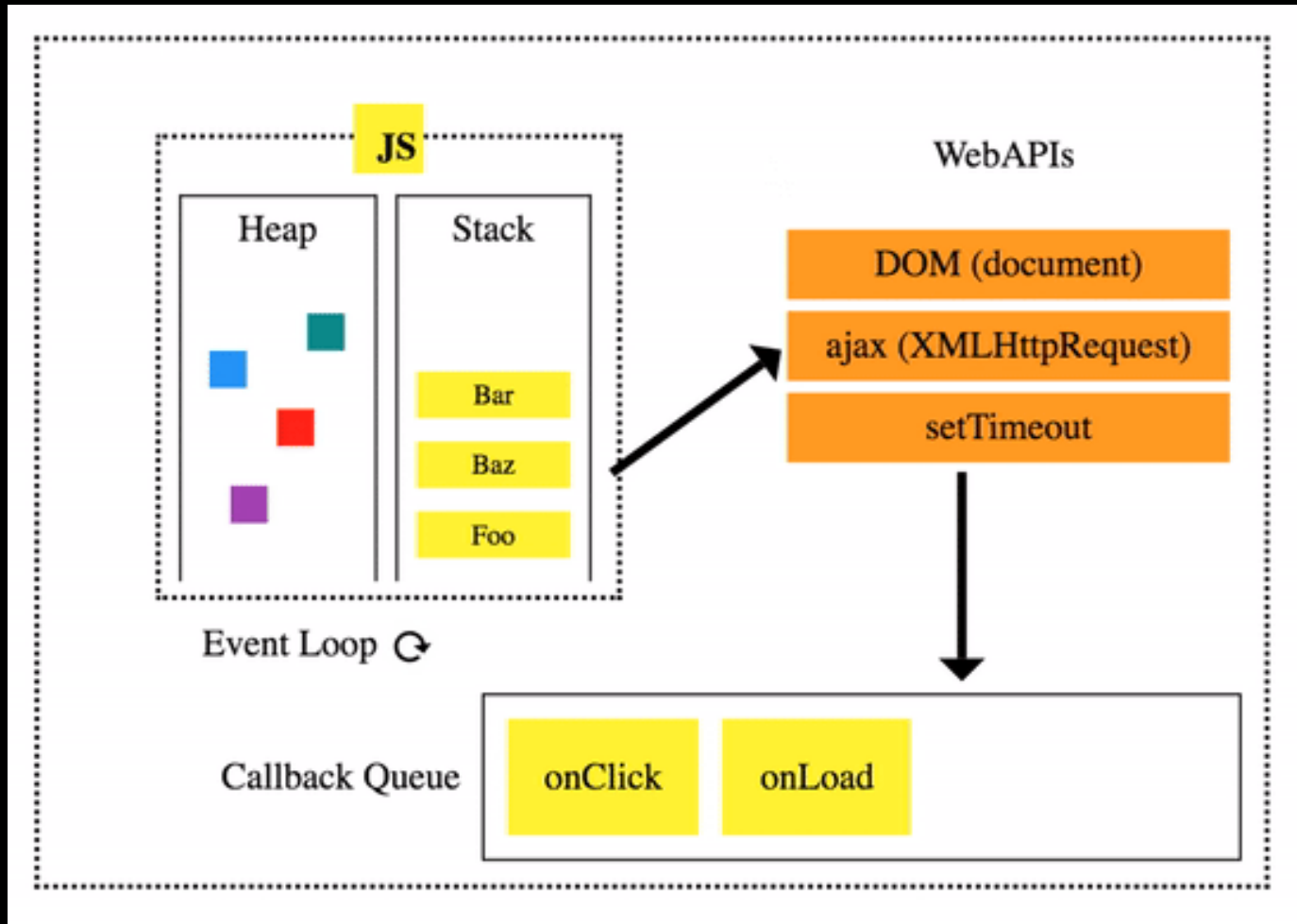
Recuerden hacer las preguntas en el **Q&A** con contexto para que no se pierdan en el chat.



Event Loop



Event Loop





Generator Functions



Generator Function

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

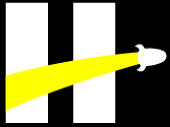




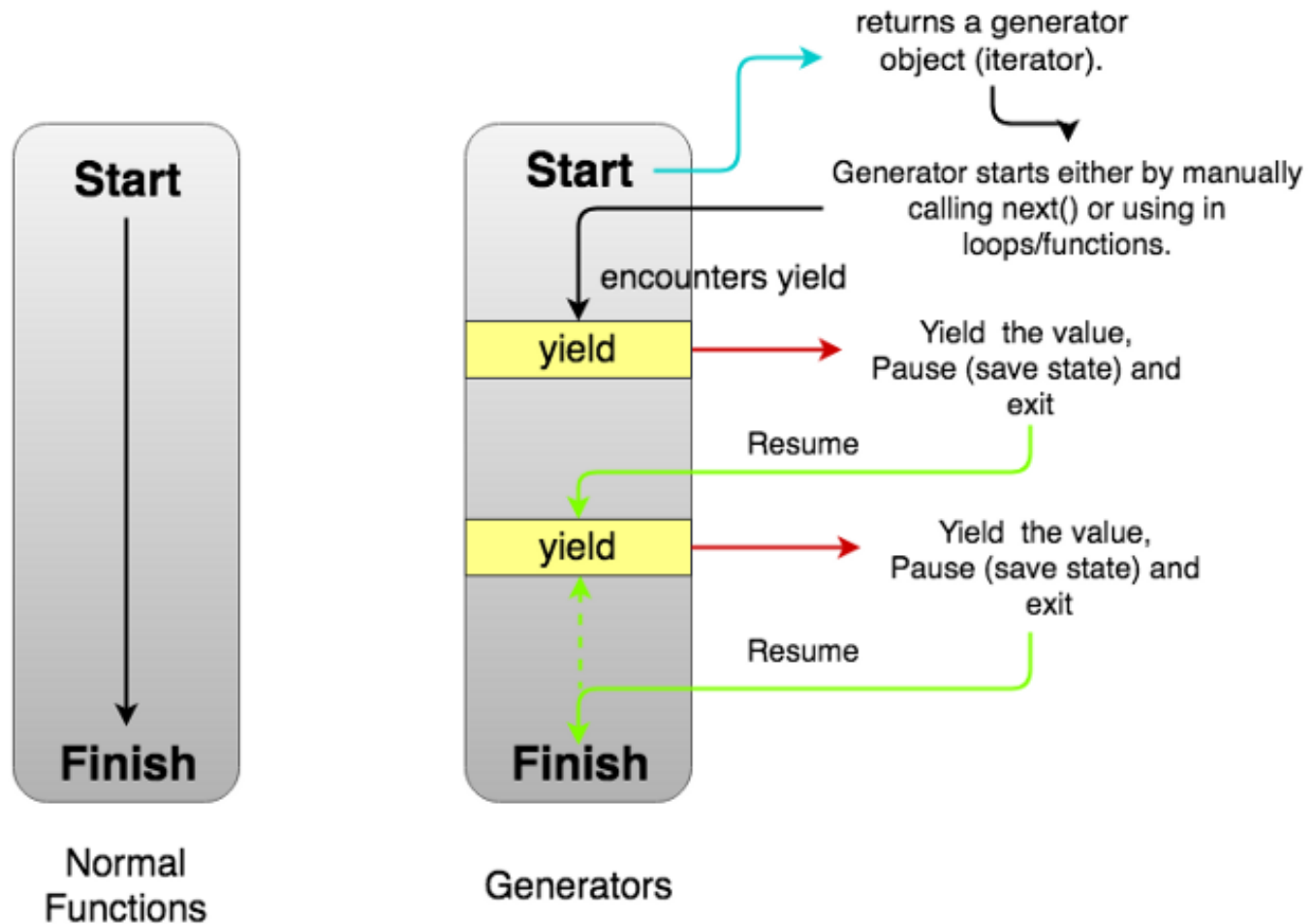
run-to-completion model

Las funciones que conocíamos hasta ahora en JS seguían este modelo donde la función va a ejecutarse por completo hasta completarse (return/error)

```
1 function normalFunction() {  
2   console.log("Iniciando función");  
3   console.log("Continuando función");  
4   console.log("Finalizando función");  
5   console.log("Fin!");  
6 }
```



Flow Differences





Generator Function return value

`function*` sirve para declarar un generator que retorna un *Generator object* sobre el cual se puede invocar el método `next()`

```
1 function* generatorShowInstructors() {
2   console.log("Iniciando generator function");
3   yield "Franco";
4   yield "Toni";
5   console.log("Generator function terminada");
6 }
7
8 var generatorObject = generatorShowInstructors();
9
10 generatorObject.next();
```

¡No se ejecuta el cuerpo de la función de forma instantánea!

<demoGF1.js />



yield vs return

- **Yield:** Pausa el generator y "retorna" el valor especificado
- **Return:** Finaliza el generator seteando el valor de done a true

```
1 function* generatorUnreachableValue() {  
2   console.log("Iniciando generator function");  
3   yield "First reachable value";  
4   yield "Second reachable value";  
5   return "Return executed";  
6   yield "Unreachable value"  
7 }  
8  
9 var generatorObject = generatorUnreachableValue();  
10  
11 generatorObject.next();  
12 generatorObject.next();  
13 generatorObject.next();  
14 generatorObject.next();
```



Infinite Generator

```
1 function* naturalNumbers() {  
2   let number = 1;  
3   while(true) {  
4     yield number;  
5     number = number + 1;  
6   }  
7 }  
8  
9 var generatorObject = naturalNumbers();  
10  
11 generatorObject.next();  
12 generatorObject.next();  
13 generatorObject.next();  
14 generatorObject.next();
```




Generator vs Normal

```
1 function* naturalXNumbers(x) {
2   let number = 1;
3   while(number < x) {
4     yield number;
5     number = number + 1;
6   }
7 }
8
9 var generatorNum = naturalXNumbers(10);
10
11 generatorObject.next();
12 generatorObject.next();
13 generatorObject.next();
14 generatorObject.next();
```



```
1 function naturalXNumbers(x) {
2   let number = 1;
3   let numbers = [];
4   while(number < x) {
5     numbers.push(number);
6     number = number + 1;
7   }
8 }
9
10 var numberArray = naturalXNumbers(10);
```

¿Cuál es la diferencia entre estos dos? ¿En cuál uso más memoria?

¿Cuándo usarías cada uno?

< DemoGF2.js />



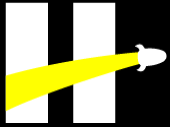
Async/Await



Async Function

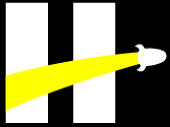
Permiten código asíncrono basado en promesas
sin necesidad de encadenar explícitamente
promesas

```
1 async function asyncCall() {  
2   const result = await resolveAfter2Seconds();  
3 }
```



Basic Flow

```
1 function resolveAfter2Seconds() {  
2   return new Promise(resolve => {  
3     setTimeout(() => {  
4       resolve('resolved');  
5     }, 2000);  
6   });  
7 }  
8  
9 async function asyncCall() {  
10  console.log('calling');  
11  const result = await resolveAfter2Seconds();  
12  console.log(result);  
13 }
```



Async function return value

¡Siempre retorna una promesa!

```
1 function resolveAfter2Seconds() {  
2   return new Promise(resolve => {  
3     setTimeout(() => {  
4       resolve('Promesa resuelta!');  
5     }, 2000);  
6   });  
7 }  
8  
9 async function asyncCall() {  
10  console.log('Iniciando asyncCall');  
11  const result = await resolveAfter2Seconds();  
12  console.log(result);  
13 }  
14  
15 var p1 = asyncCall(); // p1 --> Promise
```



Success

La promesa retornada se va a resolver al valor retornado por la función asíncrona

Error

La promesa retornada se va a rechazar con la excepción lanzada por la función asíncrona

`<demoReturnValue.js />`



Async Flow

"Yielding control"

```
1  async function showInstructors() {
2    const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('E'), 1000));
3    console.log(instructor1);
4    const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('T'), 1000));
5    console.log(instructor2);
6  }
7
8  function henryAwait() {
9    console.log("¿Quiénes son los instructores de Henry?");
10   showInstructors();
11   console.log("Gracias vuelvan pronto");
12 }
13
14 henryAwait()
15 console.log("FIN");
```



Async Flow

"Yielding control"

```
1  async function showInstructors() {
2    const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco')));
3    console.log(instructor1);
4    const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')));
5    console.log(instructor2);
6  }
7
8  async function henryAwait() {
9    console.log("¿Quienes son los intstructores de Henry?");
10   await showInstructors();
11   console.log("Gracias vuelvan pronto");
12 }
13
14 await henryAwait()
15 console.log("FIN");
```




Async/Await in Loops

```
1 const instructores = ['Franco', 'Toni', 'Martu', 'Diego'];
2
3 const delay = 1000;
4
5 async function henryAwait() {
6   console.log("¿Quienes son los intstructores de Henry?");
7   for (let i = 0; i < instructores.length; i++) {
8     const instructor = await new Promise(resolve => setTimeout(
9       () => resolve(instructores[i]),
10      delay
11    ));
12   };
13   console.log(instructor);
14 }
15 console.log("Gracias vuelvan pronto");
16 }
17
18 henryAwait();
```



Si cada promesa tarda mucho en resolverse, al estar encadenandolas de forma secuencial, el tiempo total de ejecución será muy alto





Async/Await in Loops

with callback

```
1 const instructores = ['Franco', 'Toni', 'Martu', 'Diego'];
2
3 async function henryAwait() {
4   console.log("¿Quienes son los intstructores de Henry?");
5   instructores.forEach(async instructor => {
6     const name = await new Promise(resolve => setTimeout(
7       () => resolve(instructor),
8       Math.random() * 1000
9     ));
10  });
11  console.log(name);
12  });
13  console.log("Gracias vuelvan pronto");
14 }
15
16 henryAwait();
```

¿Quienes son los intstructores de Henry?

Gracias vuelvan pronto

← ▶ Promise { <state>: "fulfilled", <value>: undefined }

Martu

Toni

Franco

Diego



<demoLoopCallback.js />



Ventajas

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

```
1  const readFilePromise = (archivo) => {  
2    promisifiedReadFile(archivo)  
3      .then(file => {  
4        console.log("Log promise file: ", file);  
5        return "Lectura exitosa";  
6      });  
7  }  
8  
9  const readFileAsync = async(archivo) => {  
10    console.log("Log async file: ", await promisifiedReadFile(archivo));  
11    return "Lectura exitosa";  
12  }
```

<demoCleanCode.js />



Ventajas

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

```
1 const readFileAsync = async(archivo) => {  
2   try {  
3     console.log("Log async file: ", await promisifiedReadFile(archivo));  
4     return "Lectura exitosa";  
5   } catch (err) {  
6     console.log("Error unificado: ", err);  
7   }  
8 }
```

<demoErrorHandler.js />



Desventajas

El código parece sincrónico, pero en realidad se ejecuta de manera asincrónica.

Tener cuidado porque `async/await` nos **oculta la complejidad**.

Async/Await = Generators + Promises

`<demoAsyncAwaitWithGenerator.js />`

Resumen

- **Generator Functions:** Es una función que muestra un objeto Generator, se definen colocando un asterisco(*) luego de la palabra clave function. Nos permite tener control en la ejecución de la función.
- **next():** Es un método del objeto Generator que muestra el valor siguiente en un generador. Muestra un objeto con las propiedades value y done.
 - value: Representa el valor del yield o del return.
 - done: Indica si el iterador recorrió los valores o no (true/false)
- **yield:** Es un operador que pausa la generator function.

Resumen

- **return:** Muestra un valor de una generator function y finaliza o corta la ejecución.
- **throw:** Genera un error y finaliza la ejecución de la generator function.
- **Estados de un objeto Generator:**
 - **suspended:** la ejecución de la generator function se detuvo, pero el proceso no finalizó.
 - **closed:** el proceso de la generator function finaliza por el return, por la iteración de todos los valores de yield o porque surgió un error.

Resumen

- **async/await:** Una forma de manejar las promesas, permite que ejecutemos funciones de forma asincrónica, nos ayuda a mantener nuestro código limpio, ordenado y simple.
- **Tener cuidado del mal uso en las funciones de async await, considerar usarla para manejo de código asincrónico.**
- **try/catch:** Bloque de instrucciones para manejar errores, donde el try ejecuta el código y el catch detecta los errores.

¿Preguntas?

Homework

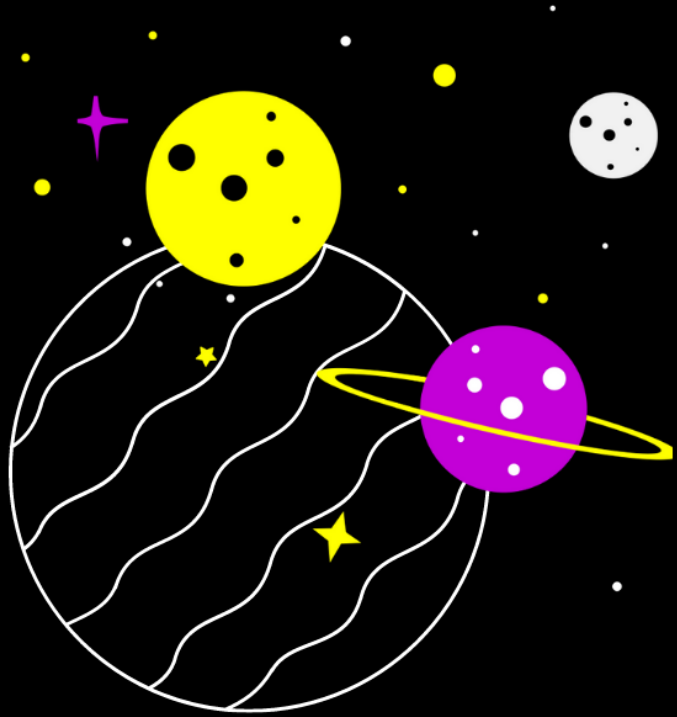
Pair Programming
SUP

¿Preguntas?

PROXIMA CLASE

Testing

HENRY



¡Muchas gracias!