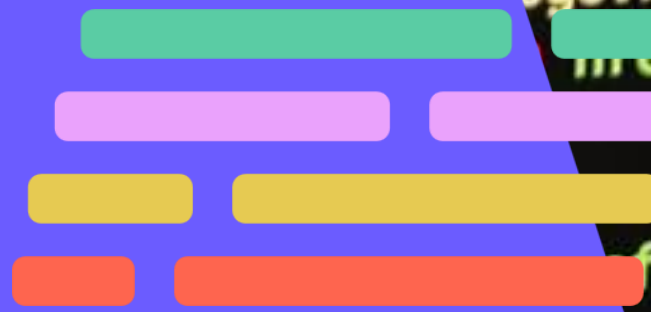
Somos un ecosistema de desarrolladores de software

JavaScript ASYNC- AWAIT



ASYNC- AWAIT

Asincronía

➢ La asincronía en JavaScript se refiere a la capacidad del lenguaje para ejecutar operaciones de manera no bloqueante, permitiendo que otras tareas continúen mientras una tarea en particular está en curso.

➢ Sirve para el manejo eficiente de operaciones que pueden llevar tiempo, como la lectura de archivos, solicitudes de red y operaciones intensivas en CPU.

➢ JavaScript es un lenguaje de programación de un solo hilo, lo que significa que ejecuta una operación a la vez en un solo hilo de ejecución.

➢ Para lograr asincronia a partir de un solo hilo, Javascript hace uso de los siguientes mecanismos: Callbacks, Promesas y async/await.

ASYNC- AWAIT

Sincronía vs Asincronía

Cuando comenzamos a programar, normalmente realizamos tareas de forma síncrona, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que el orden o flujo del programa es sencillo y fácil de observar en el código:

```
</Riwi>

primera_funcion();    // Tarea 1: Se ejecuta primero
segunda_funcion();    // Tarea 2: Se ejecuta cuando termina
tercera_funcion();    // Tarea 3: Se ejecuta cuando termina
```

En un caso real necesitaremos realizar operaciones asíncronas, donde se han de realizar tareas que tienen que esperar a que ocurra un determinado suceso que está fuera de nuestro control, y reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.

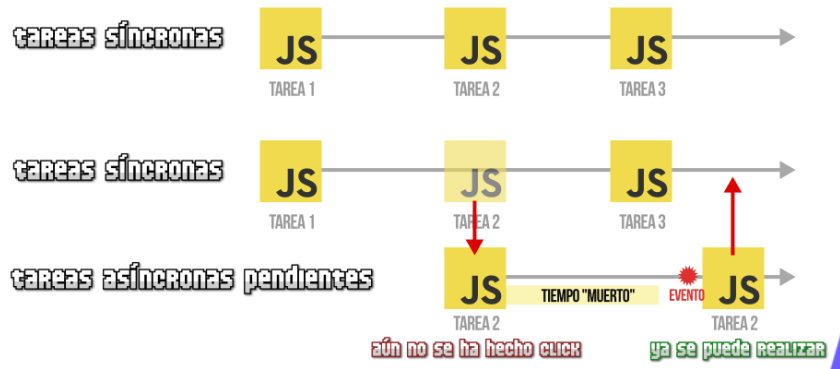
```
</Riwi>

const response = await fetch("/robots.txt");
const data = await response.text();
console.log(data);

console.log("Código síncrono.");
```

ASYNC- AWAIT

Lenguaje no bloqueante

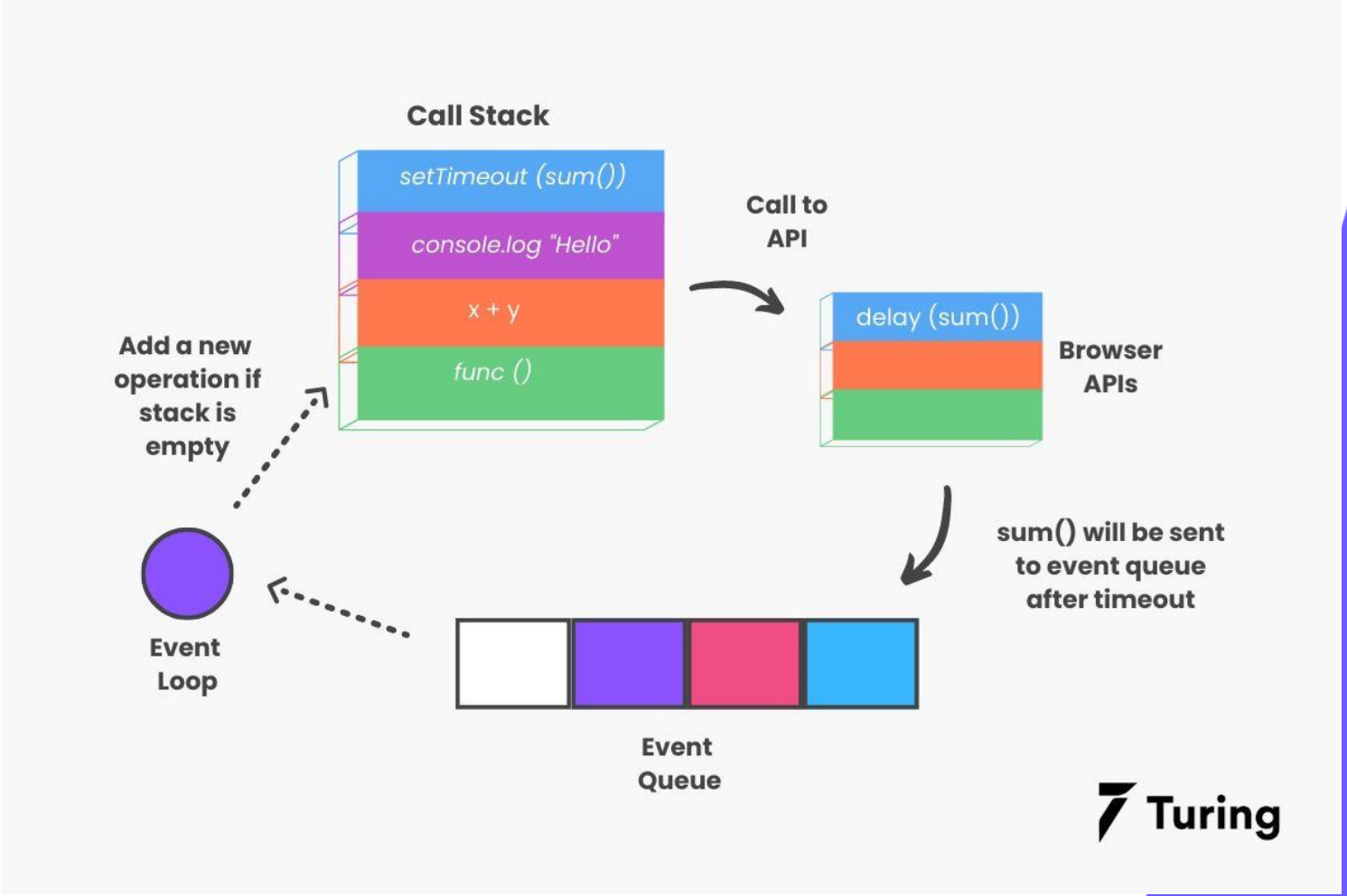
Un lenguaje no bloqueante hace referencia a que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.

Por ejemplo, la segunda_funcion() del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. En caso de ser bloqueante, hasta que el usuario no haga click, Javascript no seguiría ejecutando las demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea:



Pero como Javascript es un lenguaje no bloqueante, lo que hará es mover esa tarea a una lista de tareas pendientes a las que irá «prestándole atención» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de la segunda.

ASYNC- AWAIT


Call Stack
setTimeout (sum())
console.log "Hello"
x + y
func ()
Call to API
delay (sum())
Browser APIs
Add a new operation if stack is empty
Event Loop
sum() will be sent to event queue after timeout
Event Queue
Turing

ASYNC- AWAIT

Callbacks

las funciones callback son un tipo de funciones que se pasan por parámetro a otras funciones. Además, los parámetros de dichas funciones toman un valor especial en el contexto del interior de la función.

</Riwi>

```
const list = ["A", "B", "C"];

for (let i = 0; i < list.length; i++) {
  console.log("i=", i, " list=", list[i]);
}
```

</Riwi>

```
const list = ["A", "B", "C"];

function action(element, index) {
  console.log("i=", index, "list=", element);
}

list.forEach(action);
```

ASYNC- AWAIT

Callbacks y Asincronía

Para ejemplificar las funciones callbacks utilizadas para realizar tareas asíncronas, probablemente, el caso más fácil de entender es utilizar un temporizador mediante la función setTimeout(callback, time).
Por ejemplo se tiene una función que será llamada luego de un tiempo (2seg) y realizará lo que hay dentro de ella. Se puede expresar de manera explícita o mediante arrow function.

```
</Riwi>

function action() {
  console.log("He ejecutado la función");
}
setTimeout(action, 2000);
```

```
</Riwi>

setTimeout(() => {
  console.log("He ejecutado la función");
}, 2000);
```

```
</Riwi>

setTimeout(() => {
  console.log("Código asíncrono.");
}, 2000);

console.log("Código síncrono.");
```

En el ejemplo de la izquierda se ejecuta primero la instrucción que está al final y luego la primera.

ASYNC- AWAIT

Callbacks y Asincronía

Las funciones callback pueden utilizarse como un primer intento de manejar la asincronía en un programa.
El siguiente código muestra la estructura que tendría la declaración y el llamado de una función callback asíncrona.

```
</Riwi>

function doTask(number, callback) {
  /* Código de la función */
}

doTask(42, function(err, result) {
  /* Trabajamos con err o result según nos interese */
});
```

FUN FACT!
Las funciones callback eran muy utilizadas en la época dorada de jQuery.

ASYNC- AWAIT

Callbacks y Asincronía

Se presenta un ejemplo completo de función callback asíncrona.

Declaración

```
const doTask = (iterations, callback) => {
  const numbers = [];

  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      callback({
        error: true,
        message: "Se ha sacado un 6"
      });
      return;
    }
  }

  /* Termina bucle y no se ha sacado 6 */
  return callback(null, {
    error: false,
    value: numbers
  });
}
```

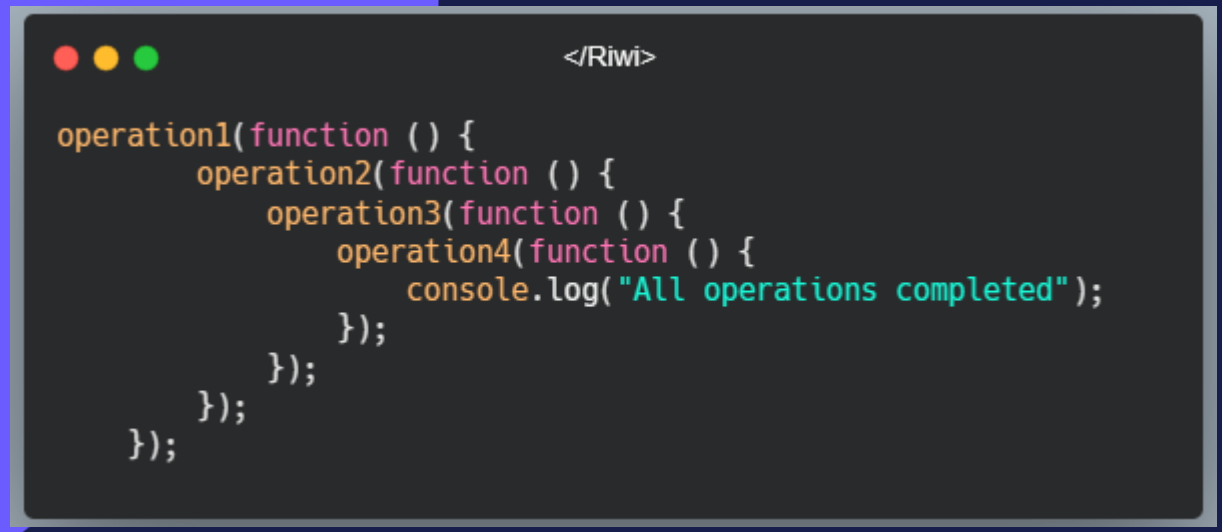
Llamado

```
doTask(10, function(err, result) {
  if (err) {
    console.error("Se ha sacado un ", err.message);
    return;
  }
  console.log("Tiradas correctas: ", result.value);
});
```

ASYNC- AWAIT

Malas Prácticas - Callback Hell

Las funciones callbacks tienen ciertas desventajas evidentes. En primer lugar, el código creado con las funciones tiende a ser caótico. Por ejemplo, pasar un null por parámetro en una función, no es recomendable.
Cuando se tiene que gestionar la asincronía varias veces en una misma función, se produce lo que se conoce como Callback Hell. El Callback Hell es el resultado de anidar varias funciones con callback en su interior.



Para evitar este problema, se pensó en las promesas.

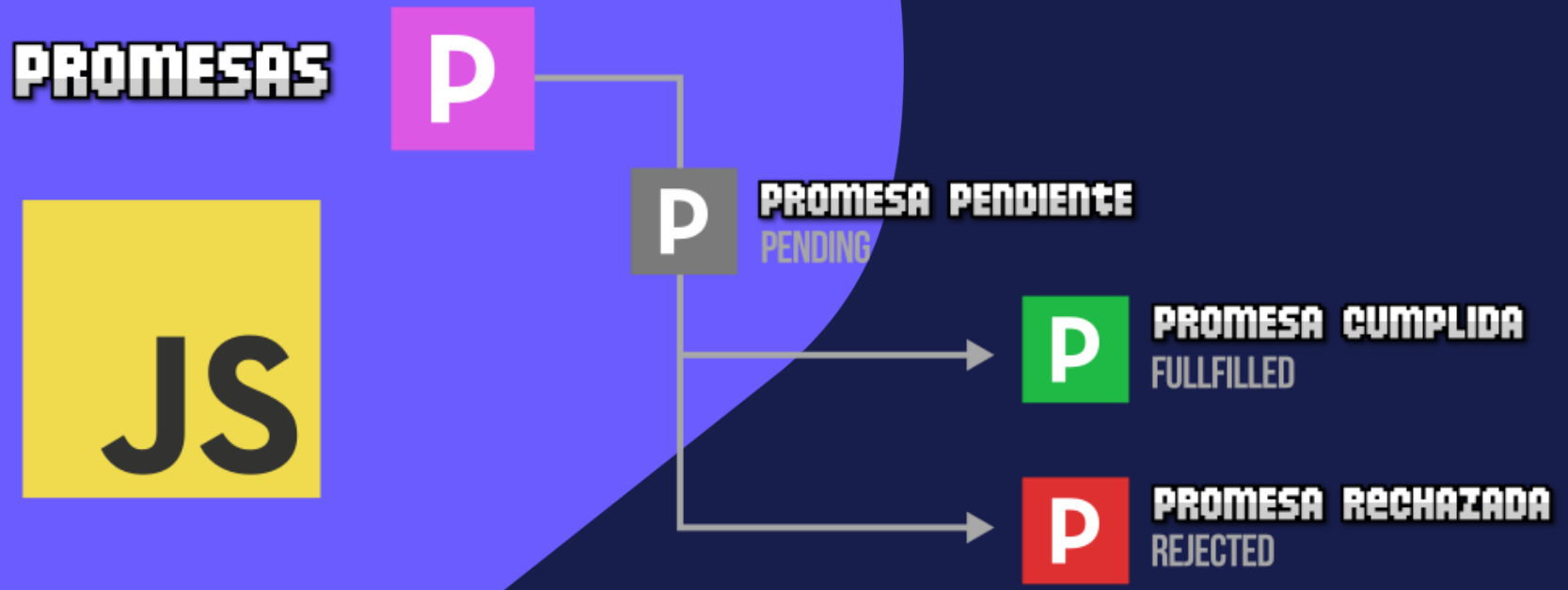
ASYNC- AWAIT

</Riwi>

Promesas

Una promesa es una acción que en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:

- La promesa se cumple (promesa resuelta).
- La promesa no se cumple (promesa rechazada).
- La promesa se queda en un estado incierto indefinidamente (promesa pendiente).



Se debe tener en cuenta que existen dos partes importantes de las promesas: como consumirlas (utilizar promesas) y como crearlas (preparar una función para que use promesas y se puedan consumir).

ASYNC- AWAIT

Promesas

Las promesas en Javascript se representan a través de un object , y cada promesa estará en un estado concreto: pendiente, aceptada o rechazada. Además, cada promesa tiene los siguientes métodos:

Métodos	Descripción
`.then(` FUNCTION `resolve)`	Ejecuta la función callback `resolve` cuando la promesa se cumple.
`.catch(` FUNCTION `reject)`	Ejecuta la función callback `reject` cuando la promesa se rechaza.
`.then(` FUNCTION `resolve,` FUNCTION `reject)`	Método equivalente a las dos anteriores en el mismo `.then()`.
`.finally(` FUNCTION `end)`	Ejecuta la función callback `end` tanto si se cumple como si se rechaza.

ASYNC- AWAIT

Promesas- Consumo

Para consumir una promesa se utiliza el .then() con un sólo parámetro, ya que lo que se busca es realizar una acción cuando la promesa se cumpla.

```
</Riwi>

fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

Se hace uso del método .catch() para actuar cuando se rechaza una promesa:
Para los métodos .then() y .catch() se suele ver de la siguiente forma para que sea más legible.

```
fetch("/robots.txt")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

ASYNC- AWAIT

Promesas- Consumo

Se pueden encadenar varios .then() si se siguen generando promesas y se devuelven con un return. Tras un .catch() también es posible encadenar .then() para continuar procesando promesas.

Mediante arrow functions se puede mejorar la legibilidad del código. Por último se agrega el método .finally() para añadir una función callback que se ejecutará tanto si la promesa se cumple o se rechaza.

```
fetch("/robots.txt")
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
  /* Código a realizar cuando se rechaza la promesa */
  });
```

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data))
  .finally(() => console.log("Terminado."))
  .catch(error => console.error(data));
```

ASYNC- AWAIT

Promesas- Consumo

Ejemplo: Se consume un recurso de GitHub (Archivo Json), y se procesa el resultado.

```
// Solicitud GET (Request).
fetch('https://api.github.com/users/manishmshiva')
    // Exito
    .then(response => response.json())  // convertir a json
    .then(json => console.log(json))    //imprimir los datos en la consola
    .catch(err => console.log('Solicitud fallida', err)); // Capturar errores
```

ASYNC- AWAIT

Promesas- Creación

Se retoma el ejemplo del callback y se modifica para que retorne una promesa.

```
const doTask = (iterations) => {
  return new Promise( (resolve, reject) => {
    const numbers = [];

    for (let i = 0; i < iterations; i++) {
      const number = 1 + Math.floor(Math.random( ) * 6);
      numbers.push(number);
      if (number === 6) {
        reject({
          error: true,
          message: "Se ha sacado un 6"
        });
      }
    }

    resolve({
      error: false,
      value: numbers
    });
  })
};
```

Se crea un nuevo objeto que envuelve toda la función doTask().
Al new Promise() se le pasa por parámetro una función con dos callbacks:

- El primer callback, resolve, lo utilizaremos cuando se cumpla la promesa.
- El segundo callback, reject, lo utilizaremos cuando se rechace la promesa.

```
doTask(10)
  .then(result => console.log("Tiradas correctas: ",
result.value))
  .catch(err => console.error("Ha ocurrido algo: ",
err.message));
```

ASYNC- AWAIT

Async- Await

En ES2017 se introducen las palabras clave async/await, las cuales se usan para gestionar las promesas de una forma más lineal. Con async/await se sigue manejando promesas, sin embargo, hay ciertos cambios importantes:
Para pasar a gestionar las promesas de forma lineal se abandona el modelo no bloqueante y se pasa a uno bloqueante.
Se parte del ejemplo anterior que las promesas que estén en .then() sean cambiadas por await.
El await lo que hace es bloquear el proceso y lo hace esperar (await), hasta que resuelva la promesa.

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data));

console.log("Código síncrono.");
```

```
const response = await fetch("/robots.txt");
const data = await response.text();
console.log(data);

console.log("Código síncrono.");
```

ASYNC- AWAIT

Async- Await

Para poder utilizar el await dentro de una función, se debe definir la función como asíncrona y al llamarla utilizar nuevamente el await. De esta manera se forma el par Async - Await.
Anteriormente era obligatorio mantener el par Async - Await, sin embargo, actualmente esto ya no es necesario, ya que se permite utilizar await fuera de funciones async si se encuentran en el nivel inicial de la aplicación, es decir en el ámbito global, gracias a algo llamado TOP LEVEL AWAIT.

```
async function request() {
  const response = await fetch("/robots.txt");
  const data = await response.text();
  return data;
}

await request();
```

ASYNC- AWAIT

Async- Await - Ejemplo

```
const doTask = async (iterations) => {
  const numbers = [];

  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      return {
        error: true,
        message: "Se ha sacado un 6"
      };
    }
  }

  return {
    error: false,
    value: numbers
  };
}
```

A partir del ejemplo visto anteriormente en los anteriores capítulos, la función doTask() realiza 10 lanzamientos de un dado y devuelve los resultados obtenidos o detiene la tarea si se obtiene un 6. La implementación de la función sufre algunos cambios, simplificándose considerablemente.

- Se añade la palabra clave async antes de los parámetros de la arrow function.
- Se desaparece cualquier mención a promesas, se devuelven directamente objetos, ya que al ser una función async se devolverá todo envuelto en una Promesa.

ASYNC- AWAIT

Async- Await - Ejemplo

Al momento de consumir promesas hay un cambio importante. No se utiliza .then(), sino que se usa await para esperar la resolución de la promesa, obteniendo el valor directamente:

```
const resultado = await doTask(10);   // Devuelve un objeto, no una promesa
```

</Be a coder>