CIP FP cecheste

Desarrollo Web en Entorno Cliente

Javascript

Asincronía [3]

Callbacks

Son las funciones que pasamos como parámetro de otras funciones de forma predefinida:

- setTimeout(callback, milisegundos, argumentos);
- setInterval(callback, milisegundos, argumentos);
- array.forEach(callback);
- elemento.addEventListener(evento, callback);
- …

Cuando empleamos funciones asíncronas como `setTimeout()`, y queremos que varias funciones se ejecuten en tiempos diferentes, por ejemplo, una al medio segundo, otra un segundo después de que se ejecute la primera, otra 2 segundos después de que se ejecute la tercera, … ¿Cómo lo haríamos?

Callbacks

Callback hell

Anidamiento de funciones que dependen unas de otras y generan un fuerte acoplamiento.

```
setTimeout(medioSegundo, 500);

function medioSegundo() {
    console.log("medioSegundo");
    setTimeout(unSegundo,1000);
}

function unSegundo() {
    console.log("Un segundo");
    setTimeout(dosSegundos,2000);
}

function dosSegundos(){
    console.log("Dos segundos");
}
```

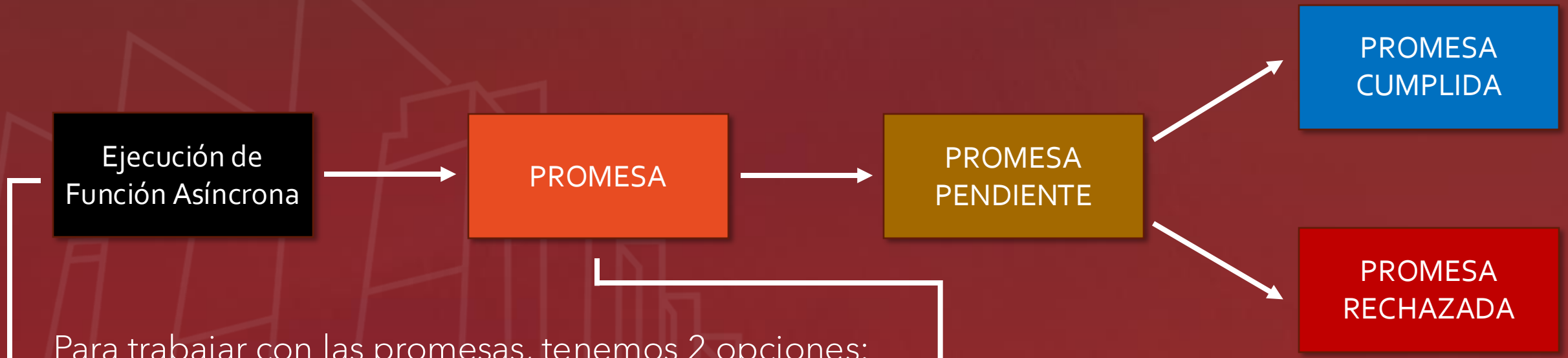
```
setTimeout(() => {
    console.log("medioSegundo");
    setTimeout(() => {
        console.log("Un segundo");
        setTimeout(() => {
            console.log("Dos segundos");
        }, 2000)
    },1000);
}, 500);
```

Debemos intentar evitarlo.

¿Cómo?

promise

Una promesa es algo que esperamos se cumpla en el futuro. Dentro de la asincronía , estas promesas están vinculadas con la ejecución de funciones asíncronas. Pueden ocurrir diferentes cosas:



Para trabajar con las promesas, tenemos 2 opciones:

1. Utilizarlas a través de funciones asíncronas.
2. Crearlas.

Promise [1]

Funciones asíncronas

thenables: Nos permiten realizar tareas a partir de los diferentes estados de las promesas:

Métodos	Descripción
.then(`función`)	Se ejecuta cuando la promesa se cumple
.catch(`función`)	Se ejecuta cuando la promesa se rechaza
.then(`funciónOk`, `funciónFail`)	Las dos anteriores en el mismo then
.finally(`función`)	cuando la promesa se queda pendiente

fetch

Funciones asíncronas

Método que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

```
let url = "http://www.dwecmola.com";
```

```
let promesa = fetch(url);
```

```
let promesa2 = promesa.then(callback)
```

```
promesa2.then(callback)
```

fetch

Funciones asíncronas

```
let url = "http://www.dwecmola.com";
```

```
let promesa = fetch(url);
```

```
let promesa2 = promesa.then(res => res.json());
let promesa2 = promesa.then(res => res.text());
```

Existen más métodos como blob() o formData().

```
promesa2.then(response => console.log(response));
```

¿Cómo podríamos hacerlo todo en una sola línea?

```
fetch("http://www.dwecmola.com").then(promesa => promesa.json()).then(json => console.log(json));
```

fetch

Funciones asíncronas

¿Cómo mejoraríamos la legibilidad?

```
fetch("http://www.dwecmola.com")
        .then(promesa => promesa.json())
        .then(json => console.log(json));
```

¿Se podrían emplear funciones en el formato convencional?

```
fetch("http://www.dwecmola.com")
    .then(function(promesa) {
        return promesa.json();
    })
    .then(function(json) {
        console.log(json);
    });
```

```
fetch("http://www.dwecmola.com")
    .then(movidas)
    .then(movidas2);

function movidas(promesa) {
    return promesa.json();
}

function movidas2(json) {
    console.log(json);
}
```

fetch

Funciones asíncronas

Por defecto, todas las peticiones que hacemos son mediante el método GET, pero...

¿y si queremos hacerlas por otro método como POST, PUT o DELETE?

¿Y si queremos enviar un formulario como hicimos en AJAX?

Añadimos opciones a la petición.

fetch

Funciones asíncronas

```
const opciones = {
    method: "GET"
};
```

Valor por defecto

```
fetch("http://www.dwecmola.com", opciones)
        .then(promesa => promesa.json())
        .then(json => console.log(json));
```

Campo	Descripción
method	Método HTTP de la petición. Por defecto, GET. Otras opciones: HEAD, POST, etc...
headers	Cabeceras HTTP. Por defecto, {}.
body	Cuerpo de la petición HTTP. Puede ser de varios tipos: String, FormData, Blob, etc...
credentials	Modo de credenciales. Por defecto, omit. Otras opciones: same-origin e include.

fetch

Funciones asíncronas

Ejemplo:

```
// Datos que queremos enviar en el cuerpo de la solicitud
const datos = {
    name: "John Doe",
    email: "john.doe@example.com",
    age: 30,
};

// Opciones de la solicitud
const options = {
    method: "POST", // Método HTTP (GET, POST, PUT, DELETE, …)
    headers: {
        "Content-Type": "application/json", // Tipo de contenido del cuerpo
        "Authorization": "Bearer YOUR_ACCESS_TOKEN", // Token de autorización si es necesario
        "Custom-Header": "CustomValue", // Otro encabezado personalizado
    },
    body: JSON.stringify(datos), // Convertir los datos a formato JSON
    credentials: "include", // Incluir cookies para solicitudes CORS ("omit", "same-origin", "include")
    mode: "cors", // Modo de la solicitud ("cors", "no-cors", "same-origin")
    cache: "no-cache", // Configuración de caché ("default", "no-cache", "reload", etc.)
    redirect: "follow", // Cómo manejar redirecciones ("follow", "error", "manual")
    referrer: "no-referrer", // Controla qué referrer se envía ("no-referrer", "client", etc.)
};
```

async y await

Con los thennables obtenemos un comportamiento secuencial de las promesas, esperando a la obtención de un resultado o estado de una promesa.

Otra opción, más natural, para controlar la secuencialidad de la ejecución de funciones asíncronas con promesas son las herramientas async/await.

- async: se coloca delante de la declaración de una función, y automáticamente lo que devuelve la función será una promesa.

```
async function asincrona() {return 3}
```

- await: se coloca delante de la llamada a una función asíncrona que trabaja con promesas, bloqueando la ejecución del código hasta que se resuelva.

```
async function asincrona() {
    await asincrona2();
}
```

NOTA: Todo `await` estará dentro de uná función declarada como `async`, siempre que no estemos en el contexto global (top-level await) y hayamos declarado el script como `type="module"`.

async y await

Ejemplo: Como resolver un `fetch()` sin thennables.

```
const url = "http://www.dwecmola.com";
```

```
const opciones = {
    method: "GET"
};
```

```
cargar(url, opciones);
```

```
async function cargar(url, opciones) {
    const promesa1 = await fetch(url, opciones);
    const promesa2 = await promesa1.json();
    listarPersonajes(promesa2);
}
```

En este caso las promesas rechazadas las deberíamos gestionar con `try... catch`

new promise

Además de `async`, tenemos otra vía para que nuestra función devuelva una promesa.

```
let promesa = new Promise();
```

```
function soloNumeroCinco() {
    console.log("Movidas guapas");

    return new Promise((bien, mal) => {
        // Obtenemos números aleatorios del 1 al 10
        let number = 1 + Math.floor(Math.random() * 10);

        if (number === 5) bien(number); // Si el número es 5, la promesa se ha cumplido
        else mal(number); // Si no, la rechazamos
    });
}

soloNumeroCinco()
    .then(numero => console.log("¡¡CONSEGUIDO!!", numero))
    .catch(numero => console.log("¡¡NO CONSEGUIDO!!", numero));
```

Funciones `callback` que indican si la promesa se ha cumplido o no, y devuelven un mensaje.