Recuerden poner a grabar la clase

Clase 18

Promesas

Diplomatura UNTREF

Temario

Promesas en JavaScript:

- qué son
- estados de una promesa
- resolver una promesa
- rechazar una promesa.
- Objeto Math y la Clase Date().

Promesas

Promesas

En JavaScript, las "promesas" (promises) son un mecanismo que se utiliza para manejar operaciones asincrónicas de manera más ordenada y legible.

Las promesas representan un valor que puede estar disponible ahora, en el futuro o nunca, y permiten ejecutar código en función del resultado de la operación asincrónica.

Promesas

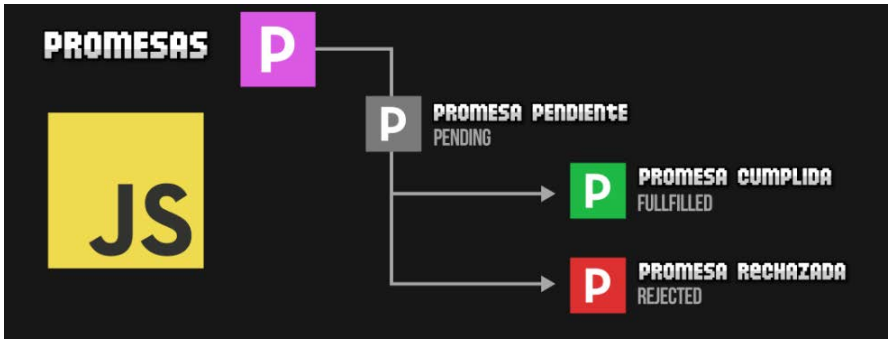
Las promesas tienen tres estados principales:

Pendiente (Pending):

Una promesa comienza en el estado "Pendiente".

Significa que la operación asincrónica asociada aún no se ha completado ni se ha rechazado. La promesa está en espera de un resultado.

Durante este estado, la promesa puede eventualmente cambiar a uno de los otros dos estados: "Resuelta" o "Rechazada".



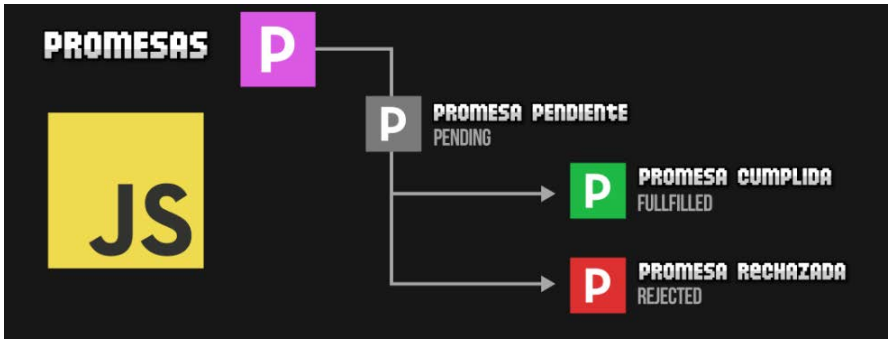
Promesas

Resuelta (Fulfilled):

Una promesa entra en el estado "Resuelta" cuando la operación asincrónica se ha completado con éxito.

En este estado, se proporciona un valor resultante (a menudo llamado "valor resuelto") que representa el resultado exitoso de la operación.

Una vez que una promesa está en este estado, no puede cambiar a ningún otro estado, y el valor resuelto es inmutable.



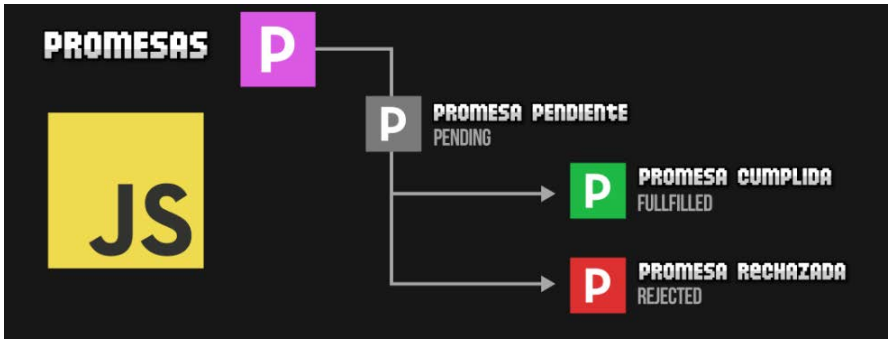
Promesas

Rechazada (Rejected):

Una promesa entra en el estado "Rechazada" cuando la operación asincrónica falla o encuentra un error.

En este estado, se proporciona un motivo de rechazo (generalmente un objeto Error) que describe la causa del fallo.

Al igual que en el estado "Resuelta", una vez que una promesa está en este estado, no puede cambiar a ningún otro estado, y el motivo de rechazo es inmutable.



Promesas

Estos estados son cruciales para manejar operaciones asincrónicas y garantizar un flujo de control adecuado en tu código.

Puedes utilizar los métodos .then() y .catch()
para manejar las transiciones entre estos estados y realizar acciones específicas en función del resultado de la promesa.

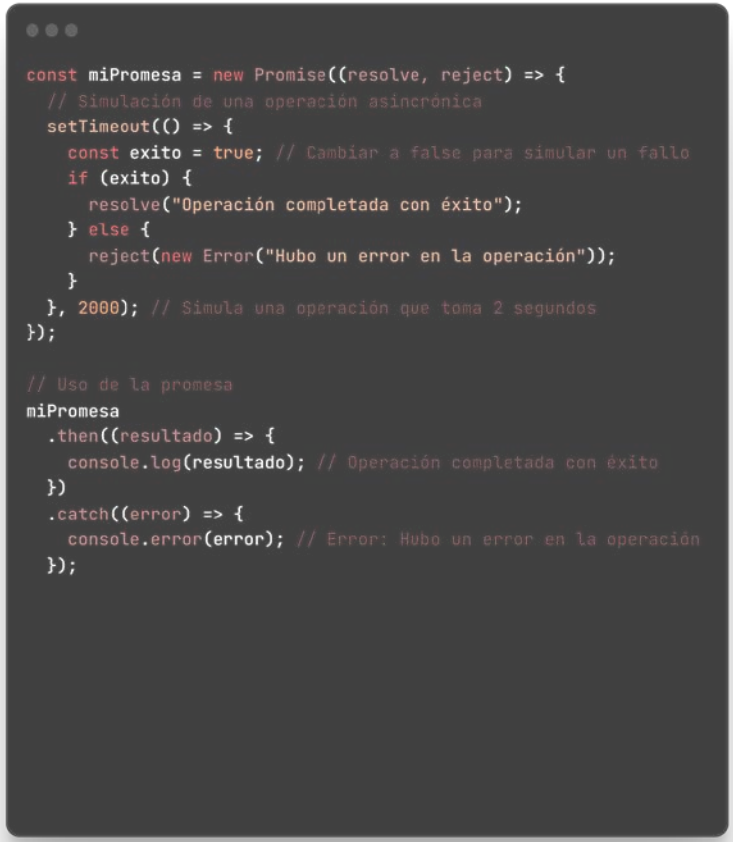
```
const miPromesa = new Promise((resolve, reject) => {
  // Simulación de una operación asincrónica exitosa
  setTimeout(() => {
    resolve("Operación completada con éxito");
  }, 2000);
});

miPromesa
  .then((resultado) => {
    // La promesa está en estado "Resuelta"
    console.log(resultado); // Imprime "Operación completada con
éxito"
  })
  .catch((error) => {
    // La promesa está en estado "Rechazada"
    console.error(error);
  });
```

Promesas

Para crear una promesa en JavaScript, puedes usar el constructor Promise, que toma una función con dos argumentos:

resolve y reject.

Dentro de esta función, realizas la **operación asincrónica** y luego llamas a **resolve** si la operación se completa con éxito o a **reject** si falla.



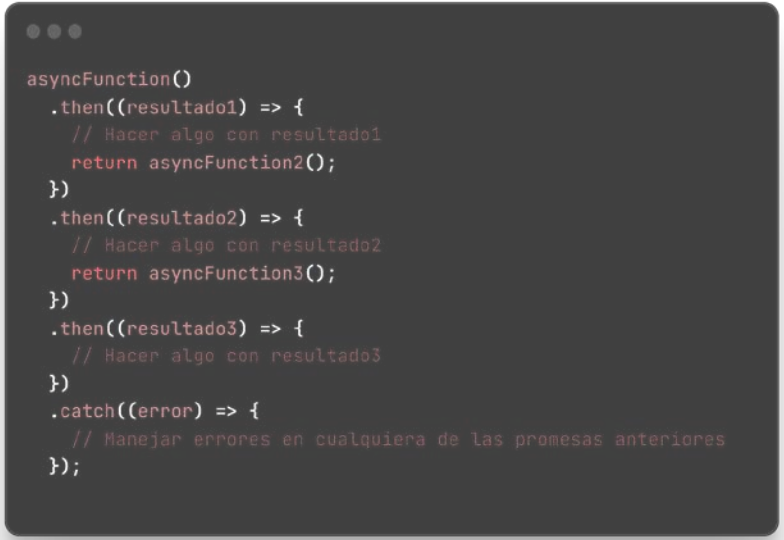
```
const miPromesa = new Promise((resolve, reject) => {
  // Simulación de una operación asincrónica
  setTimeout(() => {
    const exito = true; // Cambiar a false para simular un fallo
    if (exito) {
      resolve("Operación completada con éxito");
    } else {
      reject(new Error("Hubo un error en la operación"));
    }
  }, 2000); // Simula una operación que toma 2 segundos
});

// Uso de la promesa
miPromesa
  .then((resultado) => {
    console.log(resultado); // Operación completada con éxito
  })
  .catch((error) => {
    console.error(error); // Error: Hubo un error en la operación
  });
```

.then()

.then

El método .then() para manejar el éxito de la promesa:

Cuando una promesa se resuelve exitosamente (es decir, pasa al estado "Fulfilled"), el método .then() se ejecuta, y puedes especificar una función de devolución de llamada (callback) que maneje el resultado de la promesa.



```
asyncFunction()
  .then((resultado1) => {
    // Hacer algo con resultado1
    return asyncFunction2();
  })
  .then((resultado2) => {
    // Hacer algo con resultado2
    return asyncFunction3();
  })
  .then((resultado3) => {
    // Hacer algo con resultado3
  })
  .catch((error) => {
    // Manejar errores en cualquiera de las promesas anteriores
  });
```

.then

Esta función de devolución de llamada toma un argumento, que es el valor con el que se resolvió la promesa. Puedes nombrar este argumento como desees.

La ejecución de múltiples .then() te permite realizar una serie de acciones secuenciales con el valor resuelto de la promesa. Esto es especialmente útil cuando tienes una secuencia de operaciones asincrónicas que dependen una de la otra.

```
asyncFunction()
  .then((resultado1) => {
    // Hacer algo con resultado1
    return asyncFunction2();
  })
  .then((resultado2) => {
    // Hacer algo con resultado2
    return asyncFunction3();
  })
  .then((resultado3) => {
    // Hacer algo con resultado3
  })
  .catch((error) => {
    // Manejar errores en cualquiera de las promesas anteriores
  });
```

.catch()

.catch

El **método .catch()** para manejar errores: Cuando una promesa es rechazada (es decir, pasa al estado "Rejected"), el **método .catch()** se ejecuta, y puedes proporcionar una función de devolución de llamada que maneje el motivo del rechazo.

Esta función de devolución de llamada toma un argumento, que es generalmente un objeto Error que contiene información sobre la causa del error.

```
asyncFunction()
  .then((resultado) => {
    // Hacer algo con resultado
    return asyncFunction2();
  })
  .then((resultado2) => {
    // Hacer algo con resultado2
    return asyncFunction3();
  })
  .then((resultado3) => {
    // Hacer algo con resultado3
  })
  .catch((error) => {
    // Manejar errores en cualquiera de las promesas anteriores
    console.error(error);
  });
```

.catch

UNTREF
UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Usar .catch() al final de una cadena de promesas te permite manejar errores en cualquiera de las promesas anteriores. Esto evita que los errores se propaguen y detiene la ejecución de la cadena de promesas cuando se encuentra un error.

```
asyncFunction()
  .then((resultado) => {
    // Hacer algo con resultado
    return asyncFunction2();
  })
  .then((resultado2) => {
    // Hacer algo con resultado2
    return asyncFunction3();
  })
  .then((resultado3) => {
    // Hacer algo con resultado3
  })
  .catch((error) => {
    // Manejar errores en cualquiera de las promesas anteriores
    console.error(error);
  });
```

Resolver Promesas

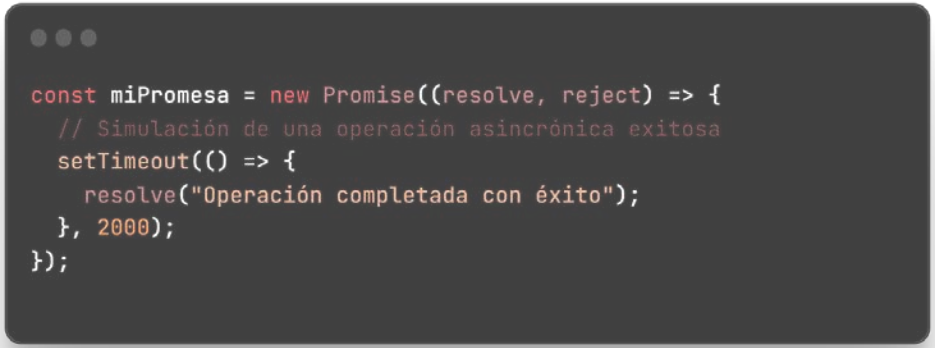
resolve en promesas

Resolver una promesa en el contexto de las promesas en JavaScript significa llevarla del estado "**Pendiente**" **al estado "Resuelta"**.

Esto implica que la **operación asincrónica** asociada con la promesa se ha completado con **éxito** y se ha proporcionado un valor resultante.

Cuando resuelves una promesa, generalmente lo haces llamando a la **función resolve()** que se pasa como argumento al constructor de la promesa cuando se crea.

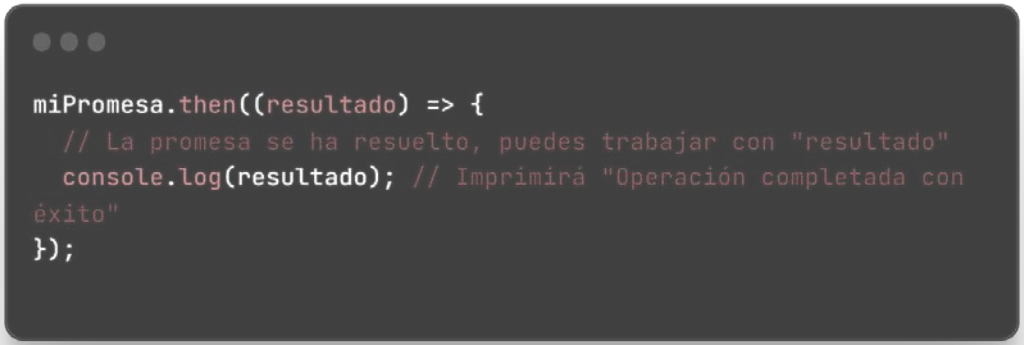
Esta función **resolve()** se llama con el valor que deseas asociar con la promesa cuando se completa la operación asincrónica.



```
const miPromesa = new Promise((resolve, reject) => {
  // Simulación de una operación asincrónica exitosa
  setTimeout(() => {
    resolve("Operación completada con éxito");
  }, 2000);
});
```

resolve en promesas

Una vez que una promesa se ha resuelto, puedes utilizar el método .then() para manejar el valor resultante y realizar acciones adicionales en función de ese valor.



```
miPromesa.then((resultado) => {
  // La promesa se ha resuelto, puedes trabajar con "resultado"
  console.log(resultado); // Imprimirá "Operación completada con
éxito"
});
```

Rechazar Promesas

Reject en promesas

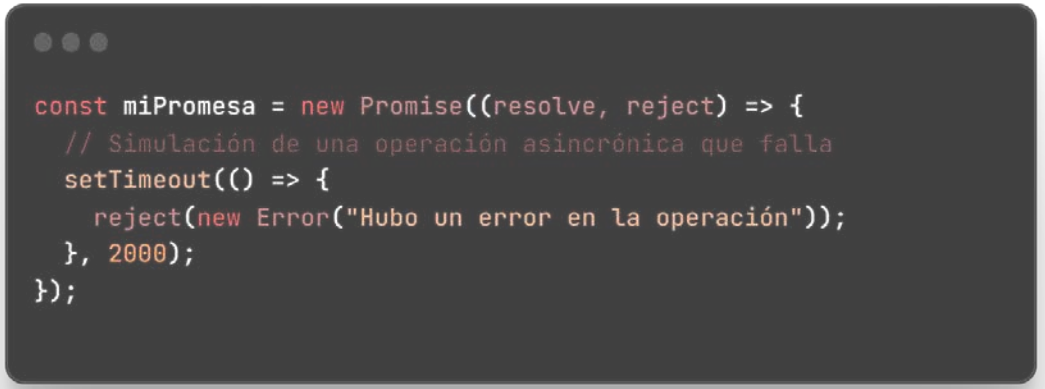
Rechazar una promesa en el contexto de las promesas en JavaScript significa llevarla del estado **"Pendiente" al estado "Rechazada".** Esto ocurre cuando la operación asincrónica asociada a la promesa encuentra un error o falla de alguna manera.

```
const miPromesa = new Promise((resolve, reject) => {
  // Simulación de una operación asincrónica que falla
  setTimeout(() => {
    reject(new Error("Hubo un error en la operación"));
  }, 2000);
});
```

Reject en promesas

Cuando **rechazas una promesa**, generalmente lo haces llamando a la **función reject()** que se pasa como argumento al constructor de la promesa cuando se crea.

Esta función **reject()** se llama con un motivo (generalmente un objeto Error o una descripción del error) que explica por qué la operación asincrónica no se pudo completar correctamente.



```
const miPromesa = new Promise((resolve, reject) => {
  // Simulación de una operación asincrónica que falla
  setTimeout(() => {
    reject(new Error("Hubo un error en la operación"));
  }, 2000);
});
```

Ejemplo con un set de datos

Ejemplo con un set de datos

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Definimos una función llamada cargarDatosDesdeServidor, que simula una operación asincrónica para cargar datos desde un servidor.

Dentro de la función cargarDatosDesdeServidor, creamos una nueva promesa utilizando el constructor Promise. Esta promesa tiene dos funciones de devolución de llamada, resolve y reject, que representan el éxito y el error de la operación, respectivamente.

Simulamos la operación asincrónica dentro del setTimeout. En este caso, hemos establecido exito en true para simular una carga exitosa de datos. Si quisieras simular un error, podrías cambiar exito a false.

Si exito es true, llamamos a resolve con los datos cargados (datos). Esto resuelve la promesa con éxito y proporciona los datos.

```
// Simulación de una función para cargar datos desde un servidor
function cargarDatosDesdeServidor() {
  // Creamos una nueva promesa utilizando el constructor
`Promise`.
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos una variable `exito` para controlar si la
operación fue exitosa o no.
      const exito = true; // Cambiar a false para simular un
error

      if (exito) {
        // Si la operación fue exitosa, resolvemos la promesa con
los datos cargados.
        const datos = ['dato1', 'dato2', 'dato3'];
        resolve(datos); // Datos cargados exitosamente
      } else {
        // Si hubo un error, rechazamos la promesa con un objeto
Error que describe el problema.
        reject(new Error('Error al cargar los datos')); //
Simular un error
      }
    }, 2000); // Simula una operación que toma 2 segundos
  });
}
```

Ejemplo con un set de datos

Si exito es false, llamamos a reject con un objeto Error que describe el motivo del error. Esto rechaza la promesa y proporciona un mensaje de error.

Fuera de la función **cargarDatosDesdeServidor,** usamos la promesa llamando a **cargarDatosDesdeServidor().**

Utilizamos el método .then() para manejar el caso en el que la promesa se resuelve exitosamente, es decir, cuando los datos se cargan correctamente. Dentro de esta función de devolución de llamada, imprimimos los datos cargados.

Usamos el método .catch() para manejar el caso en el que la promesa se rechaza, es decir, cuando ocurre un error durante la carga de datos. Dentro de esta función de devolución de llamada, imprimimos el mensaje de error.

```
// Uso de la promesa para cargar datos
cargarDatosDesdeServidor()
  .then((datos) => {
    // Esta parte se ejecutará si la promesa se resuelve
exitosamente.
    console.log('Datos cargados exitosamente:', datos);
  })
  .catch((error) => {
    // Esta parte se ejecutará si la promesa se rechaza (hubo un
error).
    console.error('Error al cargar datos:', error.message);
  });
```

Ejemplo con sweet Alert y multiples botones

sweetAlert y multiples botones ejemplo

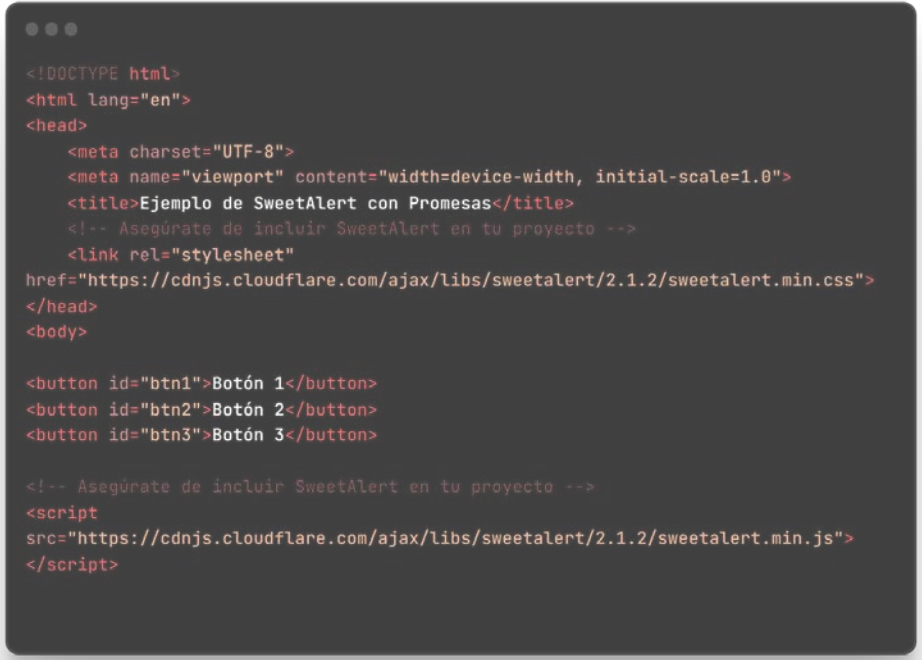
UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 1 (HTML Básico):

Creamos un documento HTML básico con tres botones (btn1, btn2 y btn3).

También hemos incluido las bibliotecas de SweetAlert (CSS y JavaScript) en el encabezado del documento.

Asegúrate de tener acceso a estas bibliotecas para que el código funcione correctamente.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo de SweetAlert con Promesas</title>
    <!-- Asegúrate de incluir SweetAlert en tu proyecto -->
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/sweetalert/2.1.2/sweetalert.min.css">
</head>
<body>

<button id="btn1">Botón 1</button>
<button id="btn2">Botón 2</button>
<button id="btn3">Botón 3</button>

<!-- Asegúrate de incluir SweetAlert en tu proyecto -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/sweetalert/2.1.2/sweetalert.min.js">
</script>
```

sweetAlert y multiples botones ejemplo

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 2 (Escucha de Eventos de Botón - Botón 1):

Agregamos un eventListener al botón con id="btn1".

Cuando se hace clic en este botón, se ejecuta una función anónima.

Dentro de esta función, llamamos a ejecutarOperacion('Operación 1') para simular una operación asincrónica y manejamos la promesa resultante con .then() y .catch().

```
<script>
document.getElementById('btn1').addEventListener('click', () => {
  ejecutarOperacion('Operación 1')
    .then((resultado) => {
      mostrarMensaje('Operación 1 completada con éxito: ' + resultado);
    })
    .catch((error) => {
      mostrarMensaje('Operación 1 fallida: ' + error.message, 'error');
    });
});
```

sweetAlert y multiples botones ejemplo

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 3 (Escucha de Eventos de Botón - Botón 2):

Agregamos un eventListener al botón con id="btn2".

Cuando se hace clic en este botón, se ejecuta una función anónima similar a la anterior.

Llamamos a ejecutarOperacion('Operación 2') para simular otra operación asincrónica y manejamos la promesa resultante con .then() y .catch().

```
document.getElementById('btn2').addEventListener('click', () => {
  ejecutarOperacion('Operación 2')
    .then((resultado) => {
      mostrarMensaje('Operación 2 completada con éxito: ' + resultado);
    })
    .catch((error) => {
      mostrarMensaje('Operación 2 fallida: ' + error.message, 'error');
    });
});
```

sweetAlert y multiples botones ejemplo

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 4 (Escucha de Eventos de Botón - Botón 3):

Agregamos un eventListener al botón con id="btn3".

Cuando se hace clic en este botón, se ejecuta una función anónima similar a las anteriores.

Llamamos a ejecutarOperacion('Operación 3') para simular una tercera operación asincrónica y manejamos la promesa resultante con .then() y .catch().

```
document.getElementById('btn3').addEventListener('click', () => {
  ejecutarOperacion('Operación 3')
    .then((resultado) => {
      mostrarMensaje('Operación 3 completada con éxito: ' + resultado);
    })
    .catch((error) => {
      mostrarMensaje('Operación 3 fallida: ' + error.message, 'error');
    });
});
```

sweetAlert y multiples botones ejemplo

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 5 (ejecutarOperacion - Simulación de Promesa):

Creamos una función llamada **ejecutarOperacion(nombreOperacion)** que devuelve una promesa.

Dentro de esta función, utilizamos **setTimeout** para simular una operación asincrónica que toma 2 segundos en completarse.

Aleatoriamente decidimos si la operación es exitosa o no **(exito).**

Si la operación es exitosa, resolvemos la promesa con un mensaje que incluye el nombre de la operación.
Si la operación falla, rechazamos la promesa con un mensaje de error.

```
function ejecutarOperacion(nombreOperacion) {
  return new Promise((resolve, reject) => {
    // Simulación de operación asincrónica
    setTimeout(() => {
      const exito = Math.random() < 0.5; // Simulamos una operación exitosa o fallida aleatoriamente
      if (exito) {
        resolve(nombreOperacion + ' exitosa');
      } else {
        reject(new Error(nombreOperacion + ' fallida'));
      }
    }, 2000); // Simula una operación que toma 2 segundos
  });
}
```

sweetAlert y multiples botones ejemplo

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Bloque 6 (mostrarMensaje - Función SweetAlert):

Creamos una función llamada **mostrarMensaje** que toma un mensaje y un tipo (por defecto, 'success') como argumentos.

Utilizamos SweetAlert **(swal())** para mostrar un cuadro de diálogo modal con el mensaje y un icono correspondiente al tipo.

En este caso, hemos configurado un botón de confirmación **("OK")** en el cuadro de diálogo para que el usuario lo cierre.

```
function mostrarMensaje(mensaje, tipo = 'success') {
  swal({
    title: mensaje,
    icon: tipo,
    buttons: {
      confirm: 'OK',
    },
  });
}
</script>
</body>
</html>
```

Objeto Math()

Objeto Math

El **objeto Math** en JavaScript es un objeto incorporado que proporciona una amplia variedad de métodos y propiedades para realizar operaciones matemáticas. No es necesario crear una instancia de este objeto, ya que está disponible globalmente en el entorno de JavaScript.

Objeto Math

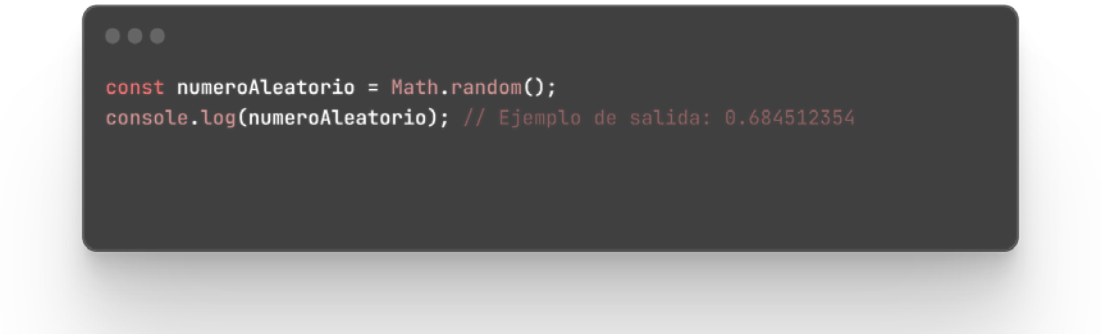
El **objeto Math** incluye métodos y propiedades para realizar operaciones comunes en matemáticas y operaciones numéricas avanzadas.

Metodos de math

Algunos de los métodos y propiedades más comunes de Math incluyen:

Math.random():

Genera un número decimal aleatorio entre 0 (inclusive) y 1 (exclusivo).



Metodos de math

Math.round():

Redondea un número al número entero más cercano.

```
const numeroRedondeado = Math.round(3.5);
console.log(numeroRedondeado); // Ejemplo de salida: 4
```

Math.floor():

Redondea un número hacia abajo al número entero más cercano.

```
const numeroRedondeadoAbajo = Math.floor(3.9);
console.log(numeroRedondeadoAbajo); // Ejemplo de salida: 3
```

Metodos de math

Math.ceil():

Redondea un número hacia arriba al número entero más cercano.

```
const numeroRedondeadoArriba = Math.ceil(3.1);
console.log(numeroRedondeadoArriba); // Ejemplo de salida: 4
```

Math.abs():

Devuelve el valor absoluto de un número.

```
const valorAbsoluto = Math.abs(-5);
console.log(valorAbsoluto); // Ejemplo de salida: 5
```

Metodos de math

Math.max():

Devuelve el valor más grande de un conjunto de números.

```
const numeroMaximo = Math.max(5, 8, 2);
console.log(numeroMaximo); // Ejemplo de salida: 8
```

Math.min():

Devuelve el valor más pequeño de un conjunto de números

```
const numeroMinimo = Math.min(5, 8, 2);
console.log(numeroMinimo); // Ejemplo de salida: 2
```

Metodos de math

Math.PI:

Contiene el valor de Pi (π).

```
console.log(Math.PI); // Ejemplo de salida: 3.141592653589793
```

Math.E:

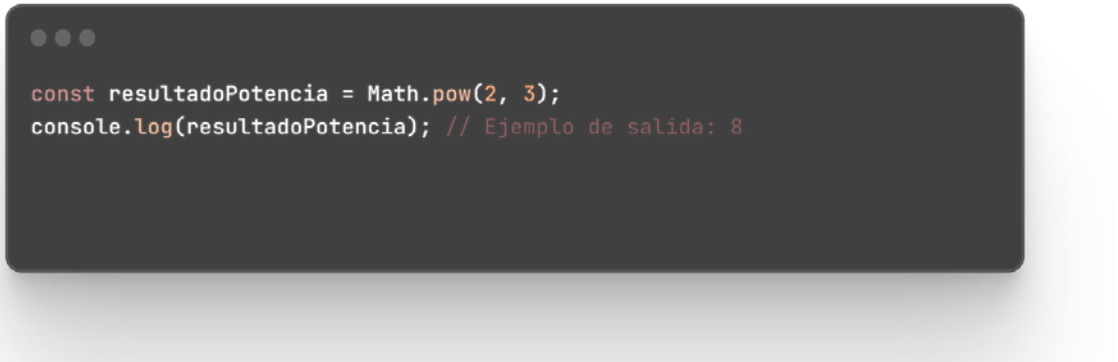
Contiene el valor de la base del logaritmo natural, número de Euler.

```
console.log(Math.E); // Ejemplo de salida: 2.718281828459045
```

Metodos de math

Math.pow():

Calcula una base elevada a una potencia específica.



```
const resultadoPotencia = Math.pow(2, 3);
console.log(resultadoPotencia); // Ejemplo de salida: 8
```

Clase Date()

Metodos de math

La **clase Date** en JavaScript se utiliza para trabajar con fechas y horas.

Permite crear objetos que representan momentos en el tiempo, y ofrece una variedad de métodos para obtener y manipular información relacionada con fechas y horas.

Crear un objeto Date

Puedes crear un objeto Date de varias formas:

Crear un objeto Date con la fecha y hora actual:

```
const fechaActual = new Date();
console.log(fechaActual);
```

Crear un objeto Date

Crear un objeto Date con una fecha específica:

```
const fechaEspecifica = new Date('2023-09-15T12:00:00');
console.log(fechaEspecifica);
```

Crear un objeto Date

Crear un objeto Date especificando año, mes, día, hora, minuto y segundo:

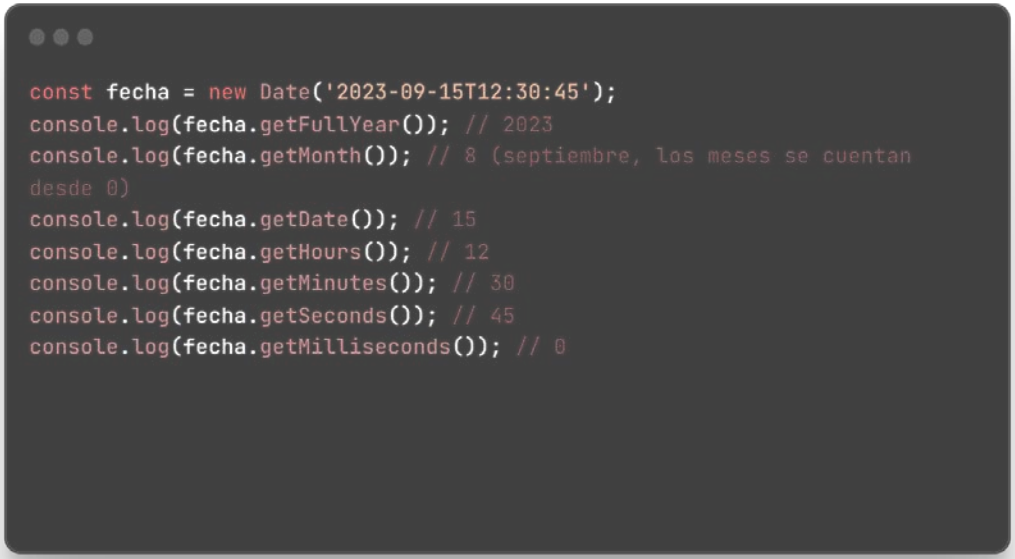
```
const fechaPersonalizada = new Date(2023, 8, 15, 12, 0, 0); // El mes se
cuenta desde 0 (enero) hasta 11 (diciembre)
console.log(fechaPersonalizada);
```

Metodos de la clase Date

Una vez que tienes un objeto Date, puedes utilizar sus métodos para obtener y manipular la información:

getFullYear(), getMonth(), getDate(), getHours(), getMinutes(), getSeconds(), getMilliseconds():

Estos métodos te permiten obtener el **año, mes, día, hora, minutos, segundos y milisegundos de una fecha**, respectivamente.



```
const fecha = new Date('2023-09-15T12:30:45');
console.log(fecha.getFullYear()); // 2023
console.log(fecha.getMonth()); // 8 (septiembre, los meses se cuentan
desde 0)
console.log(fecha.getDate()); // 15
console.log(fecha.getHours()); // 12
console.log(fecha.getMinutes()); // 30
console.log(fecha.getSeconds()); // 45
console.log(fecha.getMilliseconds()); // 0
```

Metodos de la clase Date

`toString()`: Devuelve una representación en formato de cadena de la fecha y hora.

UNTREF
UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

¿Dudas o consultas?

¡Muchas Gracias!