Agenda

- JS Variables Scope
- Loop in JS
- JS Array Methods
- JS Promise Chaining
- JS Promise Combinators
- 6 Promise Use Cases

JS Variables Scope

Definition

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript.

The visibility of the variables is referred to as the scope:

Javascript has 3 types of scope:

1. Block Scope.
2. Function Scope.
3. Local Scope.
4. Global Scope.

Here are the code snippets with explanations:

1. **Block Scope:**
   There was no idea of block scope before ES6. The let and const keywords in ES6 introduced a new feature called block scoping. The block's variables can't be accessed from outside of it.

   ```
   {
     let x = 10;
   }
   // x cannot be used here.
   ```

2. **Function Scope:**
   Variables defined inside a function are not accessible(visible) from outside the function.

   Variables declared with var, let and const are quite similar when declared inside a function.

   ```
   function Person() {
     var personName = "XYZ"; // function scope
   }
   ```

3. **Local Scope:**
   Variables declared within a Javascript function, become LOCAL to the function.

```
// code here cannot use carName
function myFunction() {
  let carName = "Tata";
  // code here can use carName
}
// code here can not use carName
```

4. **Global Scope:**
   Global scope variables can be accessed from anywhere in the program. When defined outside of a block, variables declared using var, let and const are very similar.

```
let carName = "Tata";
// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

JS Loops

Definition

A loop is a programming tool that is used to repeat a set of instructions. Iterate is a generic term that means “to repeat” in the context of loops. A loop will continue to iterate until a specified condition, commonly known as a stopping condition, is met.

There are 5 types of Loop is there in JS:

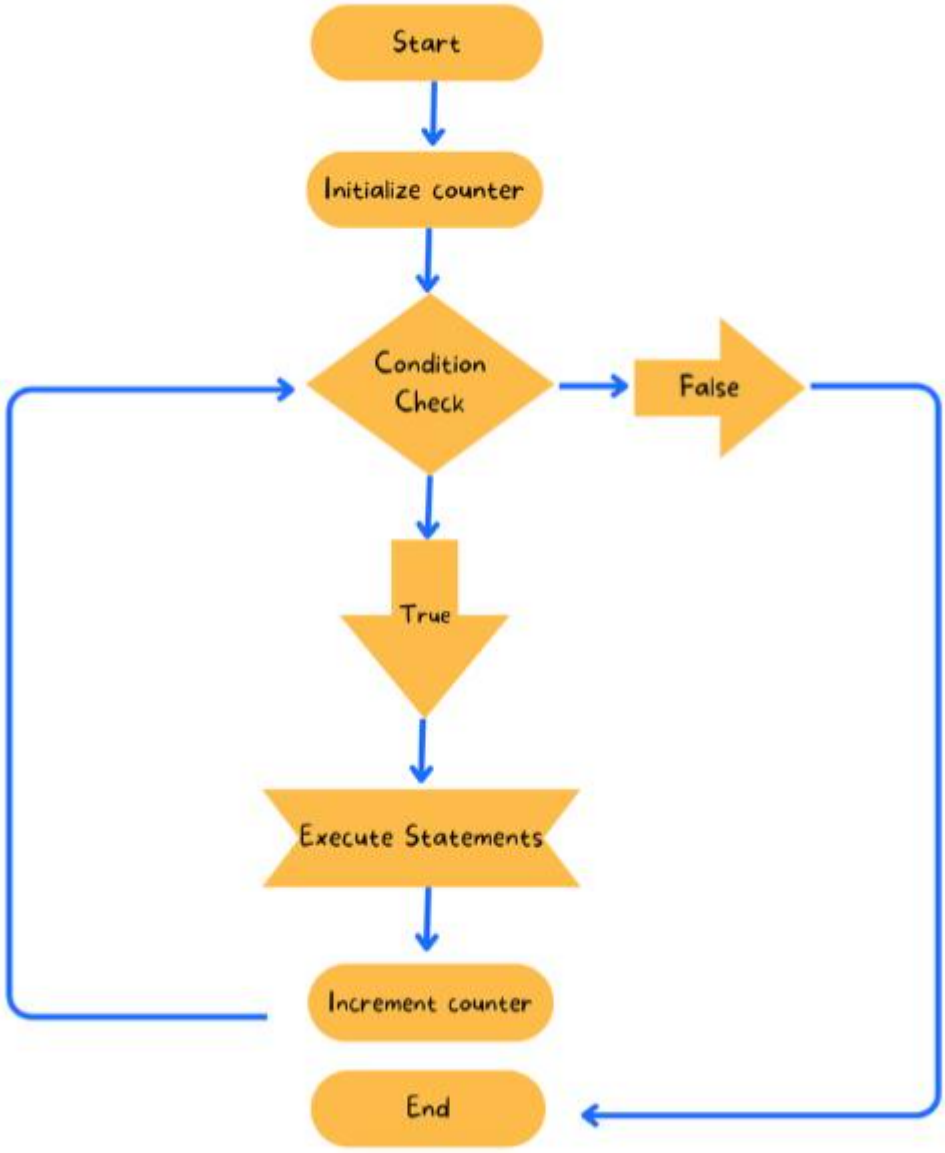
1. for Loop.
2. for in Loop.
3. for of Loop.
4. while Loop.
5. do...while Loop.

Here are the code snippets with explanations:

1. **for Loop:**

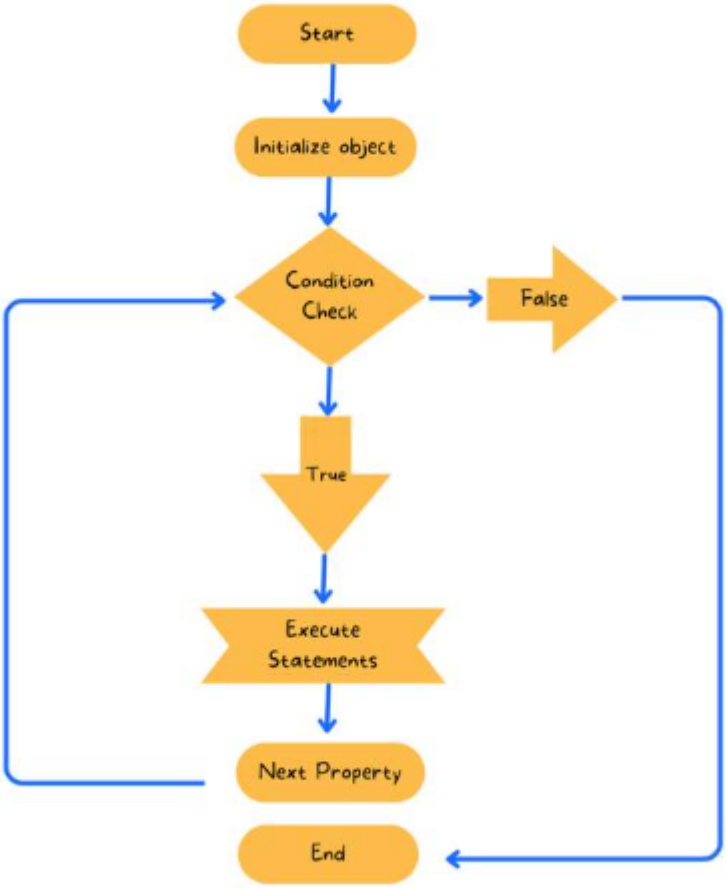
   Used for repeated execution with a defined iniliazition, condition, and iteration step.

```
for (initialization; condition; increment / decrement) {
  // code to repeat
}
```



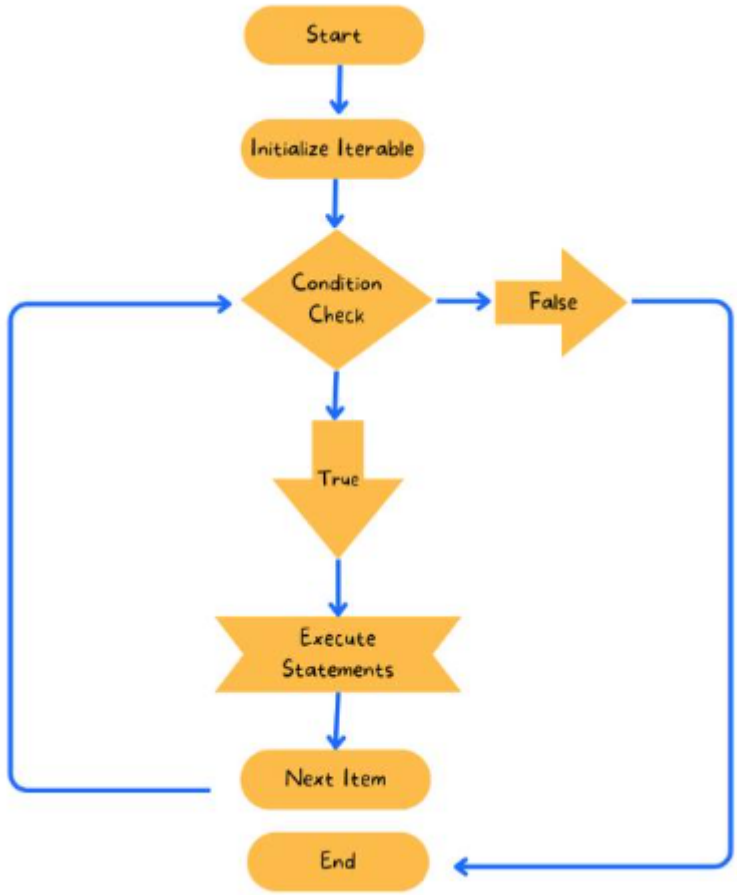
2. **for in Loop:**
   Iterates through object properties, executing code for each property.

```
for (variable in object) {
  // code for each property
}
```



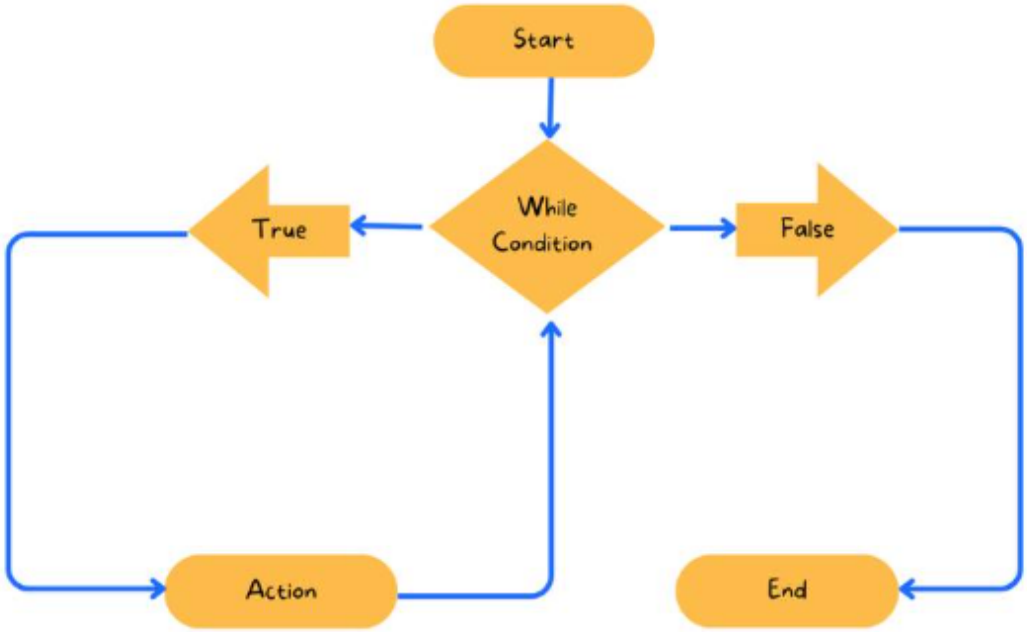
3. **for of Loop:**
   Iterates over iterable objects like arrays, executing code for each item.

```
for (variable of iterable) {
  // code of eact item
}
```



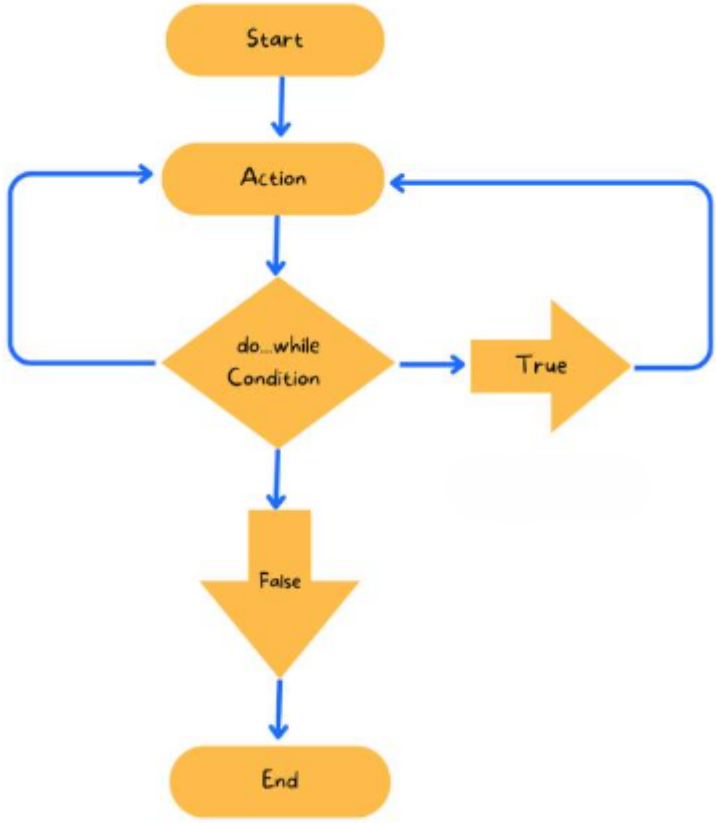
4. **while Loop:**
   Executes a block of code repeatedly as long as a specified condition is true.

```
while (condition) {
  // code to repeat
}
```



5. **do...while Loop:**
   Similar to a while but ensures at least one execution of the code block.

```
do {
  // code to repeat
} while (condition);
```



JS Array Methods

Definition

A pair of `square brackets[]` represents an in JavaScript. All the elements in the `array` are `comma(,)` separated.

In JavaScript, arrays can be a collections of elements of any type. This means that you can create an array with elements of type String, Boolean, Number, Objects, and even other arrays.

```
const mixedTypedArray = [100, true, "JavaScript", {}];
```

Add Element in Array

The `unshift()` method adds a new element to an array (at the begining), and "unshift" older elements.

```
const fruits = ["Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
console.log(fruits);
/* Output: [ 'Lemon', 'Orange', 'Apple', 'Mango' ] */
```

The `push()` method adds a new element to an array (at the end):

```
const fruits = ["Orange", "Apple", "Mango"];
fruits.push("Lemon");
console.log(fruits);
/* Output: [ 'Orange', 'Apple', 'Mango', 'Lemon' ] */
```

Access & Update Elements

To access:

```
let colors = ["red", "blue", "green"];
console.log(colors[2]); // Output: green
```

To Update:

```
let colors = ["red", "blue", "green"];
colors[1] = "purple";
console.log(colors);
// Output: [ 'red', 'purple', 'green' ]
```

Delete Element

The `shift()` method removes the first array element and shifts all other elements to a lower index.

```
const fruits = ["Orange", "Apple", "Mango"];
fruits.shift();
console.log(fruits);
/* Output: [ 'Apple', 'Mango' ] */
```

The `pop()` method removes the last element from an array:

```
const fruits = ["Orange", "Apple", "Mango"];
fruits.pop();
console.log(fruits);
// Output: [ 'Orange', 'Apple' ]
```

Arrays with let and const

The contents of an array declared with the keyword `const` and `let` can be change, however the `const` declared array cannot be reassigned a new array or different value.

```
const capitals = ["Athens", "Paris", "London"];
capitals[1] = "Berlin";
console.log(capitals);
// Output: [ 'Athens', 'Berlin', 'London' ]

capitals = "Moscow";
// Output: TypeError: Assignment to constant variable.
```

Built-in Properties

Properties are used to retrieve certain information the instance of a data type.
The `length` property returns the number of elements in the array capitals.

```
const capitals = ["Athens", "Paris", "London"];
console.log(capitals.length); // Output: 4
```

Built-in Methods

Methods are called on an array to execute certain tasks like adding and removing elements.

`.push()` is a method that adds items to the end of an array.

```
const fruits = ["Orange", "Apple", "Mango"];
fruits.push("Lemon");
console.log(fruits);
/* Output: [ 'Orange', 'Apple', 'Mango', 'Lemon' ] */
```

Arrays and Functions

If an array is mutated within a function, that change will be maintained outside of the function as well.

```
const names = ["John", "Maria", "Will"];

function addName(arr) {
  arr.push("Samuel");
}

addName(names);
console.log(names);

// Output: [ 'John', 'Maria', 'Will', 'Samuel' ]
```

Nested Arrays

Arrays can be nested or contain other arrays.

To access the element within the nested array, chain more bracket notation with index values.

```
const pizzaOrders = [
  ["Peperoni", "Coke"],
  ["Margherita", "Pepsi"],
];
console.log(pizzaOrders[1]);
console.log(pizzaOrders[1][0]);
// Output: [ 'Margherita', 'Pepsi' ]
// Margherita
```

Filter Element

The `filter()` method creates a new array filled with elements that pass a test provided by a function. The `filter()` method does not execute the function for empty element.

```
const users = [
  {
    firstName: "Joe",
    lastName: "Doe",
  },
  {
    firstName: "Ales",
    lastName: "Clay",
  },
  {
    firstName: "Opie",
    lastName: "Winston",
  },
  {
    firstName: "Wasten",
    lastName: "Doe",
  },
];

const newUser = users.filter((user) => (user.firstName = "Opie"));
console.log(newUser);
```

Reverse Element

The `reverse()` method reverses the order of the elements in an array.
The `reverse()` method overwrites the original array.

```
const originalArray = ["one", "two", "three"];
console.log("array1:", array1); // originalArray: [ 'one', 'two', 'three' ]
```

```
const reversed = originalArray.reverse();
console.log("reversed:", reversed); // reversed: [ 'three', 'two', 'one' ]
```

Sort

The `sort()` method sorts an array alphabatically.

```
const originalArray = ["one", "two", "three"];
console.log("array1:", array1); // originalArray: [ 'one', 'two', 'three' ]

const sortedArray = originalArray.sort();
console.log("Sorted Array:", sortedArray); // Sorted Array: [ 'one', 'three',
'two' ]
```

JS Promise Chaining

Definition

Promise chaining is a technique to execure multiple asynchronous operations in sequence. Each Operation starts only when the previous one completes, and the result of one operation can be passed to the next.

in promise chaining we may initialize another promise iniside our .then() method and accordingly we may execute our results.

```
// 2 Different promises
// promise chaining
function getCar(message) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resole(`Finally purchased the car ${message}`);
    }, 1000);
  });
}

function worldTour(message) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(`World Tour ${message}`);
    }, 100);
  });
}

getCar("Jaguar")
  .then((res) => {
    console.log(res);
    // Output: Finally purchased the car Jaguar
```

```
    return worldTour("Mainly USA, London and Australia");
    //Returning new promise to execute i.e., promise chaining
  })
  .then((res) => {
    console.log(res);
    // Output: World Tour Mainly USA, London and Australia

    /* if you have more promise to execute after the completion of the above
promises then you can continue with more .then() */
  })
  .catch((err) => {
    console.log(err);
  });
```

Explanation:

This code demonstrates the concept of promise chaining in JavaScript. It defines two functions, `getCar()` and `worldTour()`, both of which return promises. The `getCar()` function resolves after a delay of 1 second, while the `worldTour()` function rejects after a delay of 0.1 seconds.

The `getCar()` function is called first, and its resolved value is logged to the console. Then, the `worldTour()` function is called by returning it from the first `.then()` method. However, since `worldTour()` rejects its promise, the execution is passed to the `.catch()` method, where the error message is logged to the console.

In summary, this code shows how to chain promises together and handle errors using the `.catch()` method. It also demonstrates how to return new promises from within a `.then()` method to continue the chain.

JS Promise Combinators

Definition

Promise combinators basically help us to execute more than one promise at once and then return the result accordingly.

There are 4 promises combinators:

1. Promise.all()
2. Promise.race()
3. Promise.allSettled()
4. Promise.any()

Here are the code snippets with explanations and their respective outputs:

1. **Promise.all() method:**

   It takes an array of promises and returns an array with all the fulfilled promises.

But if any of the promises reject, the returned promise immediately rejects with the reason of the first promise that rejects.

```
const promise1 = Promise.resolve(1); // fulfilled
const promise2 = Promise.resolve("Second promise resolved"); //fulfilled
const promise3 = new Promise((resulve, reject) => {
  setTimeout(resolve, 100, "Done!"); // fulfilled
});

Promise.all([promise1, promise2, promise3])
  .then((values) => {
    console.log(values); // [1, 'Second promise resolved', 'Done!']
  })
  .catch((error) => {
    console.log(error);
  });
```

Explanation:

Three promises are created: `promise1`, `promise2`, and `promise3`. The first two are immediately resolved with values `1` and `"Second promise resolved"`, respectively. The third promise is resolved after a delay of 100 milliseconds with the value `"Done!"`.

The `Promise.all()` method is used to create a new promise that resolves when all of the input promises have resolved. The input promises are passed as an array to `Promise.all()`.

When all of the input promises have resolved, the `.then()` method is called with an array of the resolved values. In this case, the array is `[1, 'Second promise resolved', 'Done!']`, which is logged to the console.

If any of the input promises are rejected, the `Promise.all()` promise will also be rejected, and the `.catch()` method will be called with the error. In this case, there is no error, so the `.catch()` method is not executed.

2. **Promise.race() method:**

It takes an array of promises and returns the first promise that gets fulfilled or rejected ignoring the others.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, "One"); // fulfilled
});
const promise2 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, "Two"); // rejected
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, "Three"); // fulfilled
});
```

```
Promise.race([promise1, promise2, promise3])
  .then((values) => {
    console.log(values); // Two
  })
  .catch((error) => {
    console.log(error);
  });
```

Explantion:

Three promises are created: `promise1`, `promise2`, and `promise3`. The first promise is resolved after a delay of `1` second with the value `"One"`, the second promise is rejected after a delay of `0.1` seconds with the value `"Two"`, and the third promise is resolved after a delay of `0.5` seconds with the value `"Three"`.

The `Promise.race()` method is used to create a new promise that resolves or rejects as soon as any of the input promises resolve or reject. The input promises are passed as an array to `Promise.race()`.

In this case, `promise2` is rejected first, so the `Promise.race()` promise is also rejected with the value `"Two"`. The `.catch()` method is called with the error, which is logged to the console.

If `promise2` had resolved instead of rejecting, the `Promise.race()` promise would have resolved with the value `"Two"`, and the `.then()` method would have been called with that value.

3. **Promise.any() method:**

   It takes an array of promises and returns the first fulfill promise and ignores the rest promises.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, "One"); // rejected
});
const promise2 = Promise.resolve("Two"); // fulfilled
const promise3 = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, "Three"); // rejected
});

Promise.any([promise1, promise2, promise3])
  .then((values) => {
    console.log(values); // Two
  })
  .catch((error) => {
    console.log(error);
  });
```

Explanation:

Three promises are created: `promise1`, `promise2`, and `promise3`. The first promise is rejected after a delay of `0.1` seconds with the value `"One"`, the second promise is fulfilled with the value `"Two"`,

and the third promise is rejected after a delay of `1` second with the value `"Three"`.

The `Promise.any()` method is used to create a new promise that resolves or rejects as soon as any of the input promises are fulfilled. The input promises are passed as an array to `Promise.any()`.

In this case, `promise2` is fulfilled first, so the `Promise.any()` promise is also fulfilled with the value `"Two"`. The `.then()` method is called with that `value`, which is logged to the console.

If none of the promises were fulfilled, the `Promise.any()` promise would have been rejected with an `AggregateError` object containing the rejection reasons for all the input promises.

4. **Promise.allSettled() method:**

   It takes an array of promises and wait for all them to settle, regardless of whether they are resolved or rejected. It's going to return all the promises with an array of objects that each describes the outcome of each promise.

```
const promise1 = Promise.resolve("One");
const promise2 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, "Two");
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(rejected, 1000, "Three");
});

Promise.allSettled([promise1, promise2, promise3])
  .then((values) => {
    console.log(values);
  })
  .catch((error) => {
    console.log(error);
  });
```

Explanation:

Three promises are created: `promise1`, `promise2`, and `promise3`. The first promise is fulfilled with the value `"One"`, the second promise is rejected after a delay of `0.1` seconds with the value `"Two"`, and the third promise is rejected after a delay of `1` second with the value `"Three"`.

The `Promise.allSettled()` method is used to create a new promise that resolves when all of the input promises have settled, either by being fulfilled or rejected. The input promises are passed as an array to `Promise.allSettled()`.

In this case, `Promise.allSettled()` will wait for all three promises to settle, and then resolve with an array of objects, each representing the outcome of the corresponding promise.

The output will be:

```
[
  { status: "fulfilled", value: "One" },
  { status: "rejected", value: "Two" },
  { status: "rejected", value: "Three" },
];
```

The `.then()` method is called with this array, which is logged to the console.

Note that `Promise.allSettled()` does not reject if any of the input promises are rejected. Instead, it waits for all promises to settle and returns an array of outcomes.

In this example, the `.catch()` method is not called because `Promise.allSettled()` does not reject even if some of the input promises are rejected. If you want to catch errors, you would need to iterate over the resulting array and check for rejected promises.

How you can iterate over the resulting array from `Promise.allSettled()` and check for rejected promises:

```
const promise1 = Promise.resolve("One");
const promise2 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, "Two");
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, "Three");
});

Promise.allSettled([promise1, promise2, promise3])
  .then((results) => {
    results.forEach((result) => {
      if (result.status === "rejected") {
        console.log(result.reason);
      }
    });
  })
  .catch((error) => {
    console.log(error);
  });
```

In this example, the `.then()` method is called with the results array, which contains objects representing the outcome of each promise. The `forEach()` method is used to iterate over the array and check the `status` property of each object. If the status is `"rejected"`, the `reason` property is logged to the console.

This way, you can catch errors for individual promises and handle them separately.

6 Promise Use Cases in JavaScript

1. **Loading Data from an API:**

```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.log("Failed to load data", error));
```

Use Case:
Fetch data from a remote server and process it when available.

2. **Waiting for Multiple Request to Complete:**

```
Promise.all([
  fetch("https://api.example.com.data1"),
  fetch("https://api.example.com.data2"),
])
  .then((response) => {
    return Promise.all(response.map((res) => res.json()));
  })
  .then((data) => console.log(data));
```

Use Case:
Execute multiple API calls simutaneously and wait for all of them to complete.

3. **User Authentication:**

```
function loginUser(email, password) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (email === "user@example.com" && password === "password123") {
        resolve({ userId: 1, profile: "User Profile" });
      } else {
        reject(new Error("Authentication failed"));
      }
    }, 1000);
  });
}
```

Use Case:
Simulate user login, resolving on success and rejecting on failure.

4. **Data Processing Pipeline:**

```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => processData(data))
  .then((processData) => console.log(processData))
```

```
  .catch((error) => console.log("Failed in processing pipeline", error));

function processData(data) {
  //process data
  return modifiedData;
}
```

Use case:
Chain operations where each step depends on the successful completion of the previous one.

5. **Image Loading:**

```
function loadImage(url) {
  return new Promise((resolve, reject) => {
    const img = new Image();
    img.src = url;

    img.onload = () => {
      resolve(img);
    };

    img.onerror = () => {
      reject(new Error(`Failed to load image from ${url}`));
    };
  });
}
```

You can use this function like this:

```
imageLoading("https://example.com/image.jpg")
  .then((img) => {
    console.log("Image loaded successfully!");
    // Do something with the loaded image, e.g. add it to the DOM
    document.body.appendChild(img);
  })
  .catch((error) => {
    console.error("Error loading image:", error);
  });
```

Use case:
Load an Image and perform actions once it is fully loaded or handle errors if it fails to load.

6. **Timeouts:**

```
function delay(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
```

```
}

delay(1000).then(() => console.log("Delayed for 1 second"));
```

Use Case:
Execute some action after a delay without blocking the main thread.