Error Handling in Promises

- The code of a promise executor and promise handlers has an "invisible try..catch" around it. If an exception happens, it gets caught and treated as a rejection.
- The "invisible try..catch" around the executor automatically catches the error and turns it into rejected promise.
- Error handling in JavaScript promises is essential for managing the outcomes of asynchronous operations.
- When dealing with promises, errors can occur during the execution of asynchronous code, and there are several ways to handle these errors gracefully.
    - Handling Error with reject()
    - Handling Error with throw

Handling Errors with reject()

- When a promise encounters an error, the reject() function is called, which rejects the promise and passes the error to the .catch() method in the promise chain.
- It is an inbuilt function in JavaScript that returns a Promise object which has been rejected for a particular given reason.

Example

```
const p = new Promise((resolve, reject) => {
   reject('promise failed!');
});
p.catch(err => {
   console.log(err);
});
```

Example

```
new Promise((resolve, reject) => {
   reject(new Error("Error Occured!"));
 }).catch(console.log);
```

Example

```
const p = new Promise( ( resolve, reject ) => {
    setTimeout( () => {
      reject('promise failed!' );
  }, 1000);
});
  p.catch( (err) => {
    console.log( err );
});
```

Example

```
let promise = new Promise(function(res,rej){
    try{
                    const data = "This is my data";
                    res(data)
    }catch(e){
                    rej(new Error(e))
    }finally{
                    console.log("Promise Settled")
    }
})
console.log(promise)
```

Example

```
let promise = new Promise((resolve, reject) => {
                    const success = false;
                     if (success) {
                                        resolve("Operation succeeded");
                     } else {
                                        reject("Operation failed");
                     }
});
promise
  .then((result) => console.log(result))
  .catch((error) => console.error("Error:", error)); // Output: "Error: Operation failed"
```

Handling Error with throw

- You can also throw errors inside a promise. A thrown error is automatically caught by the nearest .catch() in the chain.
- "throw" is used in JavaScript to create and throw user defined exceptions.
- Using JavaScript throw statement, we can completely control program flow and generate the user define error messages.
- "throw" can be used in any Javascript try .. catch block and not only with promises.

Example

```
let promise = new Promise((resolve, reject) => {
  throw new Error("Something went wrong!");
});
promise
  .then((result) => console.log(result))
  .catch((error) => console.error("Caught:", error.message));
```

Example

```
const p = new Promise( ( resolve, reject ) => {
    throw('promise failed!' );
 });
 p.catch(err => {
     console.log( err );
 });
```

Example

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
        throw ('promise failed!');
    }, 1000);
});
p.catch((err) => {
    console.log(err);
});
```

Example

```
new Promise((resolve, reject) => {
                  resolve("ok");
}).then((result) => {
                   throw new Error("Error Occured!"); // rejects the promise
}).catch(alert);
```

Example

```
const p = new Promise((resolve, reject) => {
   throw ('promise failed!');
   console.log("Here");
});
p.catch(err => {
   console.log(err)
});
```

Chaining Promises with Error Handling: Errors can be handled at any point in the chain using .catch(). An error anywhere in the chain will propagate to the next .catch().

Example

```
let promise = new Promise((resolve, reject) => resolve("First task"));

promise
  .then((result) => {
    console.log(result);
    throw new Error("Error in second task");
  })
  .then((result) => console.log(result)) // Skipped due to error
  .catch((error) => console.error("Error caught:", error.message)); // Output: "Error
                                                              caught: Error in second task"
```

Handling Errors with .catch() in the Middle of a Chain : You can place .catch() anywhere in the promise chain. If an error occurs before it, it will catch the error and prevent it from propagating further.

Example

```
let promise = new Promise((resolve, reject) => resolve("First task"));

promise
  .then((result) => {
    console.log(result);
    throw new Error("Error in second task");
  })
  .catch((error) => {
    console.error("Caught early:", error.message); // Output: "Caught early: Error in
                                                      second task"
    return "Task Recovered";
  })
  .then((result) => console.log(result)); // Output: "Task Recovered"
```

Re-throwing Errors in .catch(): You can re-throw an error inside .catch() to propagate it further down the chain.

Example

```
let promise = new Promise((resolve, reject) => reject("Initial error"));

promise
  .catch((error) => {
    console.error("Caught:", error);
    throw new Error("New error occurred");
  })
  .then((result) => console.log(result)) // Skipped
  .catch((error) => console.error("Final catch:", error.message)); // Output: "Final
                                                                      catch: New error occurred"
```

Handling Synchronous Errors in Promise Chains: Even synchronous errors (like calling an undefined function) are caught by the .catch() in a promise chain.

Example

```
let promise = new Promise((resolve, reject) => {
  resolve("Success");
});

promise
  .then((result) => {
    console.log(result);
    undefinedFunction(); // This will throw an error
  })
  .catch((error) => console.error("Caught:", error.message)); // Output: "Caught:
                                                undefinedFunction is not defined"
```

Event Loop

- The event loop in JavaScript plays a crucial role in managing asynchronous operations, like Promises.
- The event loop in JavaScript is responsible for managing asynchronous operations. It continuously checks the call stack (for synchronous code) and processes pending tasks.
- Once the call stack is empty, it checks the microtask queue (containing tasks like Promise callbacks) and executes them first.
- After all microtasks are completed, it moves on to the macrotask queue (containing tasks like setTimeout). This ensures non-blocking execution and efficient handling of asynchronous tasks.

microtasks vs macrotasks

In JavaScript, microtasks and macrotasks represent different types of tasks that the JavaScript runtime handles during asynchronous operations. They are scheduled and processed by the event loop in different phases.

Microtasks: Microtasks are smaller, high-priority tasks that need to be executed after the current execution context but before any macrotask. They are executed as soon as the current synchronous code is completed.

Examples

Promise callbacks (.then(), .catch(), .finally())
MutationObserver callbacks (DOM change listeners)

How Microtasks are Processed?

Once the current synchronous code (in the call stack) is completed, the event loop will process all microtasks before moving on to any macrotasks.

Macrotasks: Macrotasks (also called tasks) are lower-priority tasks that are handled after all microtasks have been executed. They are usually scheduled by timers or external events.

Examples

setTimeout and setInterval callbacks
I/O operations (like network requests, file system interactions)
UI rendering

How Macrotasks are Processed?

After all microtasks are completed, the event loop checks the macrotask queue and processes the tasks in that queue, one at a time.

Example

```
console.log("Start");
setTimeout(() => {
                console.log("Macrotask: setTimeout");
                }, 0);
Promise.resolve().then(() => {
                console.log("Microtask: Promise 1");
}).then(() => {
                console.log("Microtask: Promise 2");
});
console.log("End");
```

Example

```
console.log("Start");
setTimeout(() => {
                    console.log("Macrotask 1");
                  }, 0);
Promise.resolve().then(() => {
                  console.log("Microtask");
});
setTimeout(() => {
                  console.log("Macrotask 2");
}, 0);
console.log("End");
```

Example: Promise with Event Loop

```
console.log("Start");
setTimeout(() => {
                    console.log("Timeout");
                  }, 0);
Promise.resolve().then(() => {
                  console.log("Promise 1");
}).then(() => {
                  console.log("Promise 2");
});
console.log("End");
```

Use Cases of Event Loop with Promises

- **Efficient Task Scheduling:** Promises help break down tasks, allowing non-blocking code execution and better performance.
- **Chaining Asynchronous Tasks:** By using .then(), we can chain multiple asynchronous operations without nesting (unlike callbacks).
- **Handling Data Fetching:** In web apps, Promises are used for fetching data from APIs asynchronously without blocking the main thread.

Promise static methods

The main static methods of the Promise object in JavaScript are as follows

- Promise.resolve()
- Promise.reject()
- Promise.all()
- Promise.allSettled()
- Promise.any()
- Promise.race()
- Promise.prototype.finally()

Promise.resolve(): Returns a Promise that is resolved with a given value. If the value is a Promise, it returns that Promise; otherwise, it returns a new Promise that resolves to the value.

Example: Resolving with a value

```
Promise.resolve(42).then(value => console.log(value));
```

Example: Resolving with another Promise

```
const promise = Promise.resolve("Hello Students");
Promise.resolve(promise).then(value => console.log(value));
```

Example: Resolving with an object

```
Promise.resolve({ name: "Mohit Singh" }).then(value => console.log(value.name));
```

Promise.reject(): Returns a Promise that is rejected with a given reason (error). This can be used to create rejected Promises.

Example: Rejecting with an error message

```
Promise.reject("Error occurred").catch(error => console.log(error));
```

Example: Rejecting with an Error object

```
Promise.reject(new Error("Error Occurred Again")).catch(error =>
console.log(error.message));
```

Example: Rejecting with a custom reason

```
Promise.reject({ code: 404, message: "Source Not Found" }).catch(error =>
console.log(error.message));
```

Promise.all(): It takes an iterable of Promises and returns a single Promise that resolves when all of the Promises in the iterable have resolved or rejects with the reason of the first Promise that rejects.

Example: Resolving all Promises

```
Promise.all([
  Promise.resolve(1),
  Promise.resolve(2),
  Promise.resolve(3)
]).then(values => console.log(values));
```

Example: Handling mixed resolutions

```
Promise.all([
  Promise.resolve(1),
  Promise.reject("Error"),
  Promise.resolve(3)
]).catch(error => console.log(error));
```

Example: Resolving multiple async operations

```
const promise1 = new Promise(resolve => setTimeout(() => resolve("A"), 1000));
const promise2 = new Promise(resolve => setTimeout(() => resolve("B"), 2000));
Promise.all([promise1, promise2])
  .then(values => console.log(values));
```

Promise.allSettled(): Similar to Promise.all(), but instead of waiting for all Promises to resolve, it waits for all to settle (resolve or reject) and returns an array of objects that each describe the outcome.

Example: All Promises settled

```
Promise.allSettled([
  Promise.resolve(1),
  Promise.reject("Error"),
  Promise.resolve(3)
]).then(results => console.log(results));
// Output: [{status: "fulfilled", value: 1}, {status: "rejected", reason: "Error"}, {status: "fulfilled", value: 3}]
```

Example: Handling different outcomes

```
const promises = [
  Promise.resolve("First"),
  Promise.reject("Second"),
  Promise.resolve("Third")
];
Promise.allSettled(promises).then(results => {
  results.forEach((result, index) => {
    console.log(`Promise ${index + 1}:`, result);
  });
});
```

Example: Settled promises with mixed results

```
const promise1 = Promise.resolve("Done");
const promise2 = Promise.reject("Failed");
Promise.allSettled([promise1, promise2]).then(results => console.log(results));
// Output: [{status: "fulfilled", value: "Done"}, {status: "rejected", reason: "Failed"}]
```

Promise.any(): It takes an iterable of Promise objects and, as soon as one of the Promises in the iterable fulfills, returns a single Promise that resolves with the value from that Promise. If no Promise fulfills (i.e., all are rejected), it returns a Promise that is rejected with an AggregateError, a new subclass of Error that groups together individual errors.

Example: One Promise resolves

```
Promise.any([
  Promise.reject("Error 1"),
  Promise.resolve("Success"),
  Promise.reject("Error 2")
]).then(value => console.log(value));
```

Example: All Promises rejected

```
Promise.any([
  Promise.reject("Error 1"),
  Promise.reject("Error 2")
]).catch(error => console.log(error)); // Output: AggregateError: All promises were rejected
```

Example: First successful Promise

```
const promise1 = new Promise((resolve, reject) => setTimeout(reject, 100, "Failed"));
const promise2 = new Promise((resolve) => setTimeout(resolve, 200, "Resolved"));

Promise.any([promise1, promise2]).then(value => console.log(value));
```

Promise.race(): Returns a Promise that resolves or rejects as soon as one of the Promises in the iterable resolves or rejects, with its value or reason.

Example: Fastest Promise wins

```
Promise.race([
  new Promise((resolve) => setTimeout(resolve, 2000, "First")),
  new Promise((resolve) => setTimeout(resolve, 1000, "Second"))
]).then(value => console.log(value)); // Output: Second
```

Example: Rejects with the first rejection

```
Promise.race([
  Promise.reject("Error 1"),
  Promise.resolve("Success"),
  new Promise((resolve, reject) => setTimeout(reject, 50, "Error 2"))
]).catch(error => console.log(error));
```

Example: Handling race conditions

```
const promiseA = new Promise((resolve) => setTimeout(resolve, 150, "A"));
const promiseB = new Promise((resolve) => setTimeout(resolve, 100, "B"));
Promise.race([promiseA, promiseB]).then(value => console.log(value)); // Output: B
```

Promise.prototype.finally(): While technically not a static method, it's important to mention. The finally() method returns a Promise and allows you to execute code after a Promise is settled (either fulfilled or rejected), regardless of the outcome. This is useful for cleanup actions.

Example: Cleanup after promise resolution

```
Promise.resolve("Success")
  .then(result => {
            console.log(result);
  })
  .finally(() => {
            console.log("Cleanup done");
  });
```

Example: Cleanup after promise rejection

```
Promise.reject("Error")
  .catch(error => {
            console.log(error);
  })
  .finally(() => {
    console.log("Cleaning the data");
  });
```

Example: Using finally with chaining

```
Promise.resolve("Hello Students")
  .then(value => {
        console.log(value);
        return value + " , How are you all?";
  })
  .finally(() => {
    console.log("Final cleaning up all the things");
  });
```