Class-8: Async Await

Callback hell

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Callback Example</title>
  </head>
  <body>
    <script>
      function washClothes(time, callback) {
        setTimeout(() => {
          console.log("Clothes are washed!");
          callback();
        }, time);
      }

      function dryClothes(time, callback) {
        setTimeout(() => {
          console.log("Clothes are dry!");
          callback();
        }, time);
      }

      function foldClothes(time, callback) {
        setTimeout(() => {
          console.log("Clothes are folded!");
          callback();
        }, time);
      }
```

```
      function ironClothes(time, callback) {
        setTimeout(() => {
          console.log("Clothes are ironed!");
          callback();
        }, time);
      }

      function wearClothes(time) {
        setTimeout(() => {
          console.log("Clothes are worn!");
        }, time);
      }

      // Sequentially process clothes by passing callback functi
      washClothes(2000, () => {
        dryClothes(3000, () => {
          foldClothes(2000, () => {
            ironClothes(1000, () => {
              wearClothes(1000);
            });
          });
        });
      });
    </script>
  </body>
</html>
```

Promises

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-s
    <title>Promises Example</title>
  </head>
  <body>
    <script>
      function washClothes(time) {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Clothes are washed!");
            resolve();
          }, time);
        });
      }

      function dryClothes(time) {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Clothes are dry!");
            resolve();
          }, time);
        });
      }

      function foldClothes(time) {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Clothes are folded!");
            resolve();
          }, time);
        });
      }

      function ironClothes(time) {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Clothes are ironed!");
```

```
            resolve();
          }, time);
        });
      }

      function wearClothes(time) {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Clothes are worn!");
            resolve();
          }, time);
        });
      }

      // Sequentially process clothes using promises
      washClothes(2000)
        .then(() => dryClothes(3000))
        .then(() => foldClothes(2000))
        .then(() => ironClothes(1000))
        .then(() => wearClothes(1000));
    </script>
  </body>
</html>
```

Promises with reject

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Promises Example with Errors</title>
  </head>
  <body>
```

```
<script>
  function washClothes(time) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (time < 1000) {
          reject("Washing time too short!");
        } else {
          console.log("Clothes are washed!");
          resolve("washed");
        }
      }, time);
    });
  }

  function dryClothes(time) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (time < 2000) {
          reject("Drying time too short!");
        } else {
          console.log("Clothes are dry!");
          resolve("dry");
        }
      }, time);
    });
  }

  function foldClothes(time) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (time < 1500) {
          reject("Folding time too short!");
        } else {
          console.log("Clothes are folded!");
          resolve("folded");
        }
```

```
    }, time);
  });
}

function ironClothes(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (time < 1000) {
        reject("Ironing time too short!");
      } else {
        console.log("Clothes are ironed!");
        resolve("ironed");
      }
    }, time);
  });
}

function wearClothes(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (time < 500) {
        reject("Wearing time too short!");
      } else {
        console.log("Clothes are worn!");
        resolve("worn");
      }
    }, time);
  });
}

// Sequentially process clothes with promises and error ha
washClothes(2000)
  .then(() => dryClothes(1500))
  .then(() => foldClothes(2000))
  .then(() => ironClothes(1000))
  .then(() => wearClothes(1000))
```

```
        .then(() => console.log("All tasks completed successfull
        .catch(err => console.error(err));
    </script>
  </body>
</html>
```

Promise.all()

Definition: `Promise.all()` takes an array of promises and returns a single promise that resolves when all of the promises in the array have resolved, or rejects if any of the promises reject.

Syntax:

```
javascriptCopy code
Promise.all(iterable);
```

- `iterable`: An iterable object, such as an array, containing promises.

Resolve Example

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0" />
    <title>Promise.all Example</title>
  </head>
  <body>
    <script>
```

```
      const promise1 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        resolve("Task 1 complete!");
      });

      const promise2 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        resolve("Task 2 complete!");
      });

      const promise3 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        resolve("Task 3 complete!");
      });

      Promise.all([promise1, promise2, promise3])
        .then((results) => {
          console.log("All tasks completed:");
          console.log(results);
          results.forEach((res) => console.log(res));
        })
        .catch((error) => {
          console.error("One or more tasks failed:", error);
        });
    </script>
  </body>
</html>
```

Reject Example

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
```

```
l-scale=1.0" />
    <title>Promise.all Example</title>
  </head>
  <body>
    <script>
      const promise1 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        reject("Task 1 complete!");
      });

      const promise2 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        resolve("Task 2 complete!");
      });

      const promise3 = new Promise((resolve, reject) => {
        // Simulating immediate task completion
        resolve("Task 3 complete!");
      });

      Promise.all([promise1, promise2, promise3])
        .then((results) => {
          console.log("All tasks completed:");
          console.log(results);
        })
        .catch((error) => {
          console.error("One or more tasks failed:", error);
        });
    </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Promise.all Train Ticket with Window Seat Check</titl
</head>
<body>
    <script>
        // Simulate booking a train ticket
        function bookTrain() {
            return new Promise((resolve, reject) => {
                const isSeatBooked = true; // Simulate successfu
                if (isSeatBooked) {
                    resolve("Train seat booked successfully");
                } else {
                    reject("Train booking failed");
                }
            });
        }

        // Simulate checking if the booked seat is a window seat
        function checkWindowSeat() {
            return new Promise((resolve, reject) => {
                const isWindowSeat = false; // Change to true if
                if (isWindowSeat) {
                    resolve("It's a window seat");
                } else {
                    reject("It's not a window seat");
                }
            });
        }

        // Use Promise.all to ensure both the ticket is booked a
        Promise.all([bookTrain(), checkWindowSeat()])
            .then((results) => {
                console.log("Booking successful with a window se
                console.log(results);
                // Proceed with the travel plans
```

```
            })
            .catch((error) => {
                console.error("Booking failed:", error);
                // Handle the failure, e.g., try booking another
            });
    </script>
</body>
</html>
```

Promise.allSettled()

Definition: `Promise.allSettled()` takes an array of promises and returns a single promise that resolves when all of the promises have settled, whether they have resolved or rejected.

Syntax:

```
javascriptCopy code
Promise.allSettled(iterable);
```

- `iterable`: An iterable object, such as an array, containing promises.

Resolve and Reject Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0" />
    <title>Promise.allSettled Example with Three Tasks</title
>
</head>
<body>
    <script>
        const promise1 = new Promise((resolve, reject) => {
```

```
            // Simulating immediate task completion
            resolve('Task 1 complete!');
        });

        const promise2 = new Promise((resolve, reject) => {
            // Simulating task failure
            reject('Task 2 failed!');
        });

        const promise3 = new Promise((resolve, reject) => {
            // Simulating immediate task completion
            resolve('Task 3 complete!');
        });

        Promise.allSettled([promise1, promise2, promise3])
            .then((results) => {
                console.log('All tasks settled:');
                console.log(results)
                for (let i = 0; i < results.length; i++) {
                    const result = results[i];
                    if (result.status === 'fulfilled') {
                        console.log(`Task ${i + 1}: ${result.
value}`);
                    } else {
                        console.log(`Task ${i + 1}: ${result.
reason}`);
                    }
                }
            });
    </script>
</body>
</html>
```

Output:

```
cssCopy code
All tasks settled:
{ status: "fulfilled", value: "Task 1 complete!" }
{ status: "rejected", reason: "Task 2 failed!" }
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Promise.allSettled Simple Example</title>
</head>
<body>
    <script>
        // Simulate ordering a pizza
        function orderPizza() {
            return new Promise((resolve, reject) => {
                const pizzaAvailable = true; // Change to false
                if (pizzaAvailable) {
                    resolve("Pizza ordered successfully");
                } else {
                    reject("Pizza not available");
                }
            });
        }

        // Simulate ordering a burger
        function orderBurger() {
            return new Promise((resolve, reject) => {
                const burgerAvailable = false; // Change to true
                if (burgerAvailable) {
                    resolve("Burger ordered successfully");
                } else {
                    reject("Burger not available");
```

```
                }
            });
        }

        // Simulate ordering a drink
        function orderDrink() {
            return new Promise((resolve, reject) => {
                const drinkAvailable = true; // Change to false
                if (drinkAvailable) {
                    resolve("Drink ordered successfully");
                } else {
                    reject("Drink not available");
                }
            });
        }

        // Use Promise.allSettled to handle all food orders
        Promise.allSettled([orderPizza(), orderBurger(), orderDr
            .then((results) => {
                console.log("Order results:");
                results.forEach((result) => {
                    if (result.status === "fulfilled") {
                        console.log(`Success: ${result.value}`)
                    } else {
                        console.log(`Failed: ${result.reason}`)
                    }
                });
            });
    </script>
</body>
</html>
```

Promise.any()

Definition: `Promise.any()` takes an array of promises and returns a single promise that resolves as soon as any of the promises in the array resolves, or rejects if all

of the promises reject.

Syntax:

```
javascriptCopy code
Promise.any(iterable);
```

- `iterable`: An iterable object, such as an array, containing promises.

Resolve Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0" />
    <title>Promise.any Example with Three Tasks</title>
</head>
<body>
    <script>
        const promise1 = new Promise((resolve, reject) => {
            // Simulating task failure
            reject('Task 1 failed!');
        });

        const promise2 = new Promise((resolve, reject) => {
            // Simulating task failure
            reject('Task 2 failed!');
        });

        const promise3 = new Promise((resolve, reject) => {
            // Simulating immediate task completion
            resolve('Task 3 complete!');
        });
```

```
        Promise.any([promise1, promise2, promise3])
            .then((result) => {
                console.log('First task successfully complete
d:', result);
            })
            .catch((error) => {
                console.error('All tasks failed:', error.erro
rs);
            });
    </script>
</body>
</html>
```

Output:

```
arduinoCopy code
First task completed: Task 1 complete!
```

Reject Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
l-scale=1.0" />
    <title>Promise.any Example with Three Tasks</title>
</head>
<body>
    <script>
        const promise1 = new Promise((resolve, reject) => {
            // Simulating task failure
            reject('Task 1 failed!');
        });
```

```
        const promise2 = new Promise((resolve, reject) => {
            // Simulating task failure
            reject('Task 2 failed!');
        });

        const promise3 = new Promise((resolve, reject) => {
            // Simulating immediate task completion
            reject('Task 3 complete!');
        });

        Promise.any([promise1, promise2, promise3])
            .then((result) => {
                console.log('First task successfully complete
d:', result);
            })
            .catch((error) => {
                console.error('All tasks failed:', error.erro
rs);
            });
    </script>
</body>
</html>
```

Output:

```
lessCopy code
All promises failed: AggregateError: All promises were reject
ed
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Promise.any Practical Example</title>
```

```
</head>
<body>
    <script>
        // Simulate trying to book a ride with Uber
        function bookUber() {
            return new Promise((resolve, reject) => {
                const uberAvailable = false; // Change to true t
                if (uberAvailable) {
                    resolve("Uber ride booked successfully");
                } else {
                    reject("Uber ride not available");
                }
            });
        }

        // Simulate trying to book a ride with Lyft
        function bookLyft() {
            return new Promise((resolve, reject) => {
                const lyftAvailable = true; // Change to false t
                if (lyftAvailable) {
                    resolve("Lyft ride booked successfully");
                } else {
                    reject("Lyft ride not available");
                }
            });
        }

        // Simulate trying to book a ride with a local taxi serv
        function bookLocalTaxi() {
            return new Promise((resolve, reject) => {
                const taxiAvailable = false; // Change to true t
                if (taxiAvailable) {
                    resolve("Local taxi booked successfully");
                } else {
                    reject("Local taxi not available");
                }
```

```
            });
        }

        // Use Promise.any to book the first available ride
        Promise.any([bookUber(), bookLyft(), bookLocalTaxi()])
            .then((result) => {
                console.log("Ride booked:", result);
            })
            .catch((error) => {
                console.error("No rides available:", error);
            });
    </script>
</body>
</html>
```

Promise.race()

Definition: `Promise.race()` takes an array of promises and returns a single promise that resolves or rejects as soon as any of the promises in the array resolves or rejects.

Syntax:

```
javascriptCopy code
Promise.race(iterable);
```

Resolve Example

In this example, we have two promises that resolve after different durations. The promise that resolves first will determine the outcome of `Promise.race()`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initia
```

```
l-scale=1.0" />
    <title>Promise.race Example with Three Tasks</title>
</head>
<body>
    <script>
        const promise1 = new Promise((resolve, reject) => {
            // Simulating immediate task failure
            reject('Task 1 failed!');
        });

        const promise2 = new Promise((resolve, reject) => {
            // Simulating immediate task completion
            resolve('Task 2 complete!');
        });

        const promise3 = new Promise((resolve, reject) => {
            // Simulating task that would complete later (not
reached due to race)
            resolve('Task 3 complete!');
        });

        Promise.race([promise1, promise2, promise3])
            .then((result) => {
                console.log('First task completed (resolve or
reject):', result);
            })
            .catch((error) => {
                console.error('First task failed:', error);
            });
    </script>
</body>
</html>
```

Output:

```
arduinoCopy code
First task settled: Task 1 complete!
```

Detailed Explanation of `Promise.race`

`Promise.race` is a method provided by JavaScript's Promise API that takes an iterable of promises (like an array) and returns a single promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects. This means that the outcome of `Promise.race` is determined by the first promise in the iterable to complete, whether it completes successfully (resolves) or unsuccessfully (rejects).

How `Promise.race` Works:

1. **Multiple Promises**:
   - When you pass multiple promises to `Promise.race`, it starts all of them simultaneously.
   - It doesn't matter what order the promises are passed in. What matters is which one completes first.
2. **Resolution or Rejection**:
   - The first promise to either resolve or reject will determine the outcome of the `Promise.race`.
   - If the first promise that completes is resolved, `Promise.race` will resolve with the value of that promise.
   - If the first promise that completes is rejected, `Promise.race` will reject with the reason for that rejection.
3. **Other Promises**:
   - After one promise resolves or rejects, the other promises still continue executing in the background, but their results do not affect the outcome of the `Promise.race`.

- This is important because it means that any side effects of those promises (like network requests, timers, etc.) will still occur, even though their results are ignored.

Promises vs Async/Await in JavaScript

Before async/await, to make a promise we wrote this:

```
function order(){
   return new Promise( (resolve, reject) =>{

    // Write code here
   } )
}
```

Now using async/await, we write one like this:

```
//👇 the magical keyword
 async function order() {
    // Write code here
 }
```

Async Await

- An async function in JavaScript is a function declared with the async keyword.
- The async function always returns a promise.
- If the function returns a value, the corresponding promise will be fulfilled with that value. Conversely, if the async function generates an error, the promise will be rejected with the said error.

```
<!DOCTYPE html>
<html lang="en">
```

```
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
  <script>
    async function javascript(){
        console.log("Welcome to JS")
    }

    //run javascript() in console and show that async function r

    async function javascript() {
      return ("Welcome to JS");
    }

    //run javascript() in console and show that async function r

  </script>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
  <script>
    async function javascript() {
      return "Welcome to JS";
    }
```

```
    javascript().then((res) => console.log(res));
  </script>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
  <script>
    async function javascript() {
      // This will cause a ReferenceError because nonExistentFu
      nonExistentFunction();
    }

    javascript()
      .then((res) => console.log(res))
      .catch((error) => console.error("Caught an error:", error
  </script>
</html>
```

Introducing `await`

1. **Explain What `await` Does**:
   - `await` pauses the execution of an `async` function until the promise is resolved or rejected.
   - It makes your asynchronous code look more like synchronous code, which can be easier to read and write.
2. **Basic Example Using `await`**:

- Now, let's refactor the `javascript()` function you provided to use `await`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
</html>

<script>
  async function greet() {
    let promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve("Hello"), 5000);
    });

    let result = await promise; // wait until the promise resolv

    console.log(result); // "Hello"
    console.log("ahssnsknssa");
  }

  greet();
</script>
```

Example: Using `await` with the `javascript()` Function

```
// Simulating an order processing system
let processOrder = new Promise(function (resolve, reject) {
    setTimeout(function () {
        const isOrderSuccessful = true; // Change to false to si
        if (isOrderSuccessful) {
            resolve('Order processed successfully');
```

```
        } else {
            reject('Order processing failed: Payment error or it
        }
    }, 4000); // Simulate a 4-second delay for processing the or
});

// Using .then() to handle the resolved value with error handlin
processOrder
    .then(function(result) {
        console.log(result);  // Logs 'Order processed successfu
        console.log('Thank you for your purchase!');  // Additio
    })
    .catch(function(error) {
        console.error('Error:', error); // Logs the error if the
        console.log('Please try again or contact customer suppor
    });
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-sca
  <title>Order Processing Example</title>
</head>
<body>
<script>
  // Simulating an order processing system
  function processOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const isOrderSuccessful = true; // Change to false to si
        if (isOrderSuccessful) {
          resolve('Order processed successfully');
        } else {
          reject('Order processing failed: Payment error or iten
```

```
        }
      }, 4000); // Simulate a 4-second delay for processing the
    });
  }

  // Async function to handle the order processing
  async function handleOrder() {
    try {
      const result = await processOrder(); // Wait for the orde
      console.log(result); // Logs 'Order processed successfully
      console.log('Thank you for your purchase!'); // Additional
    } catch (error) {
      console.error('Error:', error); // Logs the error if the p
      console.log('Please try again or contact customer support
    }
  }

  // Call the async function
  handleOrder();
</script>
</body>
</html>
```

Explanation:

- **Refactoring with `await`**:
  - The `displayMessage()` function is an `async` function that uses `await` to handle the promise returned by `javascript()`.
  - The `await` keyword pauses the execution of `displayMessage()` until `javascript()` resolves, making the code cleaner and easier to follow.

Next Steps:

1. **Multiple `await` Calls**:

- Now, demonstrate how you can use multiple `await` calls in sequence, which will help students understand how to manage dependent asynchronous tasks.

Example: Multi-Step Order Processing with `.then()` and `.catch()`

```
htmlCopy code
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Multi-Step Order Processing Example</title>
</head>
<body>
<script>
  // Simulate order validation
  function validateOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const isValid = true; // Change to false to simulate
an error
        if (isValid) {
          resolve('Order validation successful');
        } else {
          reject('Order validation failed');
        }
      }, 1000); // Simulate a 1-second delay
    });
  }

  // Simulate payment processing
  function processPayment() {
    return new Promise((resolve, reject) => {
```

```
      setTimeout(() => {
        const paymentSuccessful = true; // Change to false to
simulate an error
        if (paymentSuccessful) {
          resolve('Payment processed successfully');
        } else {
          reject('Payment failed: Insufficient funds or card
declined');
        }
      }, 2000); // Simulate a 2-second delay
    });
  }

  // Simulate packaging the order
  function packageOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('Order packaged successfully');
      }, 1500); // Simulate a 1.5-second delay
    });
  }

  // Simulate shipping the order
  function shipOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('Order shipped successfully');
      }, 2500); // Simulate a 2.5-second delay
    });
  }

  // Handling the full order processing using .then() and .ca
tch()
  validateOrder()
    .then(result => {
      console.log(result); // Logs 'Order validation successf
```

```
ul'
      return processPayment(); // Proceed to payment processi
ng
    })
    .then(result => {
      console.log(result); // Logs 'Payment processed success
fully'
      return packageOrder(); // Proceed to packaging the orde
r
    })
    .then(result => {
      console.log(result); // Logs 'Order packaged successful
ly'
      return shipOrder(); // Proceed to shipping the order
    })
    .then(result => {
      console.log(result); // Logs 'Order shipped successfull
y'
      console.log('Thank you for your purchase!');
    })
    .catch(error => {
      console.error('Error:', error); // Logs the error if an
y of the steps fail
      console.log('Please try again or contact customer suppo
rt.');
    });

</script>
</body>
</html>
```

Explanation:

1. `validateOrder`, `processPayment`, `packageOrder`, `shipOrder` **Functions**:

- These functions simulate different steps in the order processing workflow, each returning a promise that either resolves or rejects based on the simulated conditions.

2. **Promise Chain with `.then()` and `.catch()`:**
   - **Step 1**: Start with `validateOrder()`. If successful, move to the next step.
   - **Step 2**: Proceed to `processPayment()` and, if successful, move on to packaging.
   - **Step 3**: Continue to `packageOrder()` and, if successful, move on to shipping.
   - **Step 4**: Finally, `shipOrder()` and, if successful, log a thank you message.
   - **Error Handling**: If any step fails, the `.catch()` block catches the error and logs the failure message along with suggested next steps.

Benefits of Using `.then()` and `.catch()`:

- **Chained Execution**: Each step is executed in sequence, and the next step only begins after the previous one is resolved.
- **Error Handling**: Errors at any step are caught and handled in a centralized `.catch()` block.
- **Clear Structure**: The chaining structure clearly shows the flow of operations, making it easy to follow.

This version using `.then()` and `.catch()` offers a practical approach for handling sequential asynchronous operations in a structured and readable way, while also ensuring robust error handling at every step of the process.

Example: Multi-Step Order Processing with Multiple `await` Calls

```
htmlCopy code
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
  <title>Multi-Step Order Processing Example</title>
</head>
<body>
<script>
  // Simulate order validation
  function validateOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const isValid = true; // Change to false to simulate
an error
        if (isValid) {
          resolve('Order validation successful');
        } else {
          reject('Order validation failed');
        }
      }, 1000); // Simulate a 1-second delay
    });
  }

  // Simulate payment processing
  function processPayment() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const paymentSuccessful = true; // Change to false to
simulate an error
        if (paymentSuccessful) {
          resolve('Payment processed successfully');
        } else {
          reject('Payment failed: Insufficient funds or card
declined');
        }
      }, 2000); // Simulate a 2-second delay
    });
  }

  // Simulate packaging the order
```

```
  function packageOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('Order packaged successfully');
      }, 1500); // Simulate a 1.5-second delay
    });
  }

  // Simulate shipping the order
  function shipOrder() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('Order shipped successfully');
      }, 2500); // Simulate a 2.5-second delay
    });
  }

  // Async function to handle the full order processing
  async function handleOrder() {
    try {
      const validation = await validateOrder();
      console.log(validation); // Logs 'Order validation succ
essful'

      const payment = await processPayment();
      console.log(payment); // Logs 'Payment processed succes
sfully'

      const packaging = await packageOrder();
      console.log(packaging); // Logs 'Order packaged success
fully'

      const shipping = await shipOrder();
      console.log(shipping); // Logs 'Order shipped successfu
lly'
```

```
      console.log('Thank you for your purchase!');
    } catch (error) {
      console.error('Error:', error); // Logs the error if an
y of the steps fail
      console.log('Please try again or contact customer suppo
rt.');
    }
  }

  // Call the async function
  handleOrder();
</script>
</body>
</html>
```

Explanation:

1. `validateOrder` **Function**:
   - Simulates the validation of an order. It resolves if the order is valid and rejects if not.
2. `processPayment` **Function**:
   - Simulates processing the payment. The promise resolves if the payment is successful and rejects if there is an error.
3. `packageOrder` **Function**:
   - Simulates packaging the order. This step always resolves successfully in this example.
4. `shipOrder` **Function**:
   - Simulates shipping the order. This step also always resolves successfully.
5. `handleOrder` **Function**:
   - This `async` function sequentially processes the order by awaiting each step.

- It logs the result of each step to the console.
- If any step fails, the `catch` block handles the error, logs the issue, and suggests next steps.

6. **Calling `handleOrder`:**
    - The `handleOrder` function is called to start the multi-step order processing.

Benefits of Multiple `await` Calls:

- **Sequential Processing**: Each step is processed in order, ensuring that the next step only starts once the previous one has successfully completed.
- **Clear Error Handling**: If any step fails, the error is caught and handled, providing a clear flow of control.
- **Realistic Workflow**: This example mimics a real-world order processing workflow, with multiple dependent steps.

This approach demonstrates how you can use multiple `await` calls in an `async` function to handle a series of related asynchronous operations in a clear and structured way.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Async Await Example with Error Handling</title>
</head>
<body>
    <script>
        function washClothes(time) {
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    console.log("Clothes are washed!");
                    resolve();
```

```
                // reject("Failed to wash clothes"); // Unco
            }, time);
        });
    }

    function dryClothes(time) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                console.log("Clothes are dry!");
                resolve();
                // reject("Failed to dry clothes"); // Uncor
            }, time);
        });
    }

    function foldClothes(time) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                console.log("Clothes are folded!");
                resolve();
                // reject("Failed to fold clothes"); // Unco
            }, time);
        });
    }

    function ironClothes(time) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                console.log("Clothes are ironed!");
                resolve();
                // reject("Failed to iron clothes"); // Unco
            }, time);
        });
    }

    function wearClothes(time) {
```

```
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    console.log("Clothes are worn!");
                    resolve();
                    // reject("Failed to wear clothes"); // Unc
                }, time);
            });
        }

        // Async function to process clothes sequentially with e
        async function processClothes() {
            try {
                await washClothes(2000);
                await dryClothes(3000);
                await foldClothes(2000);
                await ironClothes(1000);
                await wearClothes(1000);
                console.log("All tasks completed successfully!"
            } catch (error) {
                console.error("Error during clothes processing:"
            }
        }

        // Call the async function
        processClothes();
    </
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
```

```
</html>

<script>
  function fetchWeatherData(cityId, getNextCity) {
    setTimeout(() => {
      console.log(`Weather data fetched for city ID: ${cityId}`
      if (getNextCity) {
        getNextCity();
      }
    }, 2000);
  }

  // Callback hell example for fetching weather data
  fetchWeatherData(1, () => {
    console.log("Fetching weather data for city ID 2 ...");
    fetchWeatherData(2, () => {
      console.log("Fetching weather data for city ID 3 ...");
      fetchWeatherData(3, () => {
        console.log("Fetching weather data for city ID 4 ...");
        fetchWeatherData(4);
      });
    });
  });
</script>
```

Using promises

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-sca
  <title>Document</title>
</head>
<body></body>
```

```
<script>
  function fetchWeatherData(cityId) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log(`Weather data fetched for city ID: ${cityId}
        resolve("Success");
      }, 2000);
    });
  }

  // Promise chain example for fetching weather data
  console.log("Fetching weather data for city ID 1 ...");
  fetchWeatherData(1)
    .then(() => {
      console.log("Fetching weather data for city ID 2 ...");
      return fetchWeatherData(2);
    })
    .then(() => {
      console.log("Fetching weather data for city ID 3 ...");
      return fetchWeatherData(3);
    })
    .then(() => {
      console.log("Fetching weather data for city ID 4 ...");
      return fetchWeatherData(4);
    })
    .then(() => {
      console.log("All weather data fetched.");
    })
    .catch(err => console.error(err));
</script>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-sca
  <title>Document</title>
</head>
<body></body>
<script>
  function fetchWeatherData(cityId) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log(`Weather data fetched for city ID: ${cityId
        resolve("Success");
      }, 2000);
    });
  }

  // Async/await example for fetching weather data
  async function fetchAllWeatherData() {
    try {
      console.log("Fetching weather data for city ID 1 ...");
      await fetchWeatherData(1);

      console.log("Fetching weather data for city ID 2 ...");
      await fetchWeatherData(2);

      console.log("Fetching weather data for city ID 3 ...");
      await fetchWeatherData(3);

      console.log("Fetching weather data for city ID 4 ...");
      await fetchWeatherData(4);

      console.log("All weather data fetched.");
    } catch (err) {
      console.error(err);
    }
  }
```

```
  // Call the async function
  fetchAllWeatherData();
</script>
</html>
```

Tasks

1. **Problem Statement - Async/Await Practice**

Implement an async function `calculateTotal` that takes an array of numbers as input. Inside this function:

- Use the `multiply` function to multiply each number in the array by 3.
- Use the `findOdd` function to filter out and return only the odd numbers from the multiplied results.
- Finally, use the `findSum` function to calculate and return the sum of the odd numbers.

Ensure error handling for each async operation and use async/await to handle promises.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body></body>
</html>

<script>
```

```
// Function to multiply each number in the array by 3
function multiply(numbers) {
  return new Promise((resolve, reject) => {
    try {
      const multipliedNumbers = numbers.map((number) => numbe
      resolve(multipliedNumbers);
    } catch (error) {
      reject('Error in multiply function');
    }
  });
}

// Function to filter out only the odd numbers from the array
function findOdd(numbers) {
  return new Promise((resolve, reject) => {
    try {
      const oddNumbers = numbers.filter((number) => number % 2
      resolve(oddNumbers);
    } catch (error) {
      reject('Error in findOdd function');
    }
  });
}

// Function to calculate the sum of the numbers in the array
function findSum(numbers) {
  return new Promise((resolve, reject) => {
    try {
      const totalSum = numbers.reduce((sum, number) => sum + n
      resolve(totalSum);
    } catch (error) {
      reject('Error in findSum function');
    }
  });
}
```

```
  let numbers = [1, 2, 3, 4, 5];

  // Function to calculate the total sum of odd numbers after mu
  multiply(numbers)
    .then((multipliedNumbers) => findOdd(multipliedNumbers))
    .then((oddNumbers) => findSum(oddNumbers))
    .then((totalSum) => {
      console.log("Total sum of odd multiplied numbers:", totalS
    })
    .catch((error) => {
      console.error("Error during calculation:", error);
    });
</script>
```

```
// Function to multiply each number in the array by 3
async function multiply(numbers) {
    return numbers.map(number => number * 3);
}

// Function to filter out only the odd numbers from the array
async function findOdd(numbers) {
    return numbers.filter(number => number % 2 !== 0);
}

// Function to calculate the sum of the numbers in the array
async function findSum(numbers) {
    return numbers.reduce((sum, number) => sum + number, 0);
}

// Function to calculate the total sum of odd numbers after mult
async function calculateTotal(numbers) {
    try {
        // Step 1: Multiply each number by 3
        const multipliedNumbers = await multiply(numbers);
```

```
        // Step 2: Filter out only odd numbers
        const oddNumbers = await findOdd(multipliedNumbers);

        // Step 3: Calculate the sum of the odd numbers
        const totalSum = await findSum(oddNumbers);

        return totalSum;
    } catch (error) {
        console.error('Error during calculation:', error);
    }
}

// Example usage
async function runCalculation() {
    const numbersArray = [1, 2, 3, 4, 5];
    const total = await calculateTotal(numbersArray);
    console.log('Total sum of odd multiplied numbers:', total);
}

// Call the function to start the calculation
runCalculation();
```