Error Handling and Reduce

What Is Error Handling?

Error handling means catching and fixing **unexpected problems** in your code — without stopping the entire program.

Imagine this line:

```
let result = 10 / 0;
```

It runs fine in JavaScript, but what if you try:

```
let total = price * quantity; // But price is not defined!
```

Boom 💥 – **ReferenceError**.

You need a way to **catch that error** so your app doesn't crash. That's where `try...catch` comes in.

---

1. `try...catch` – Catch and Handle Errors

Example 1: Using an undefined variable

```
try { let total = price * 5; // price is not declared console.log("Total:", t
otal); } catch (error) { console.log("Something went wrong!"); console.log("E
rror message:", error.message); }
```

Output:

```
Something went wrong! Error message: price is not defined
```

What Happened?

- JavaScript **stopped execution** when it saw an error
- `catch` block **caught** that error
- Instead of crashing, it showed a helpful message

2. `finally` – Code That Always Runs

Example 2: Cleaning up after an error

```
try { let marks = [90, 85, 100]; console.log(marks[5].toString()); // undefin
ed.toString() = error } catch (error) { console.log("Caught error:", error.me
ssage); } finally { console.log("This line runs no matter what."); }
```

Output:

```
Caught error: Cannot read properties of undefined (reading 'toString') This l
ine runs no matter what.
```

`finally` is useful for:

- Hiding loading spinners
- Closing files
- Resetting UI

---

3. `throw` – Create Your Own Errors

Sometimes you want to raise an error on purpose.

Example 3: Custom error for invalid age

```
function checkAge(age) { if (age < 0) { throw new Error("Age cannot be negati
ve"); } console.log("Valid age:", age); } try { checkAge(-5); } catch (error)
{ console.log("Caught:", error.message); }
```

Output:

```
Caught: Age cannot be negative
```

Why use `throw`?

To **stop execution** if something is clearly wrong, like wrong inputs or unexpected behavior.

---

4. Errors Have Names & Messages

When an error happens, JavaScript gives you a built-in **Error object**:

```
try { abc(); // not defined } catch (err) { console.log("Name:", err.name);
// ReferenceError console.log("Message:", err.message); // abc is not defined
}
```

---

5. Errors in Functions

Example 4: Divide numbers with error check

```
function divide(a, b) { if (b === 0) { throw new Error("Cannot divide by zer
o"); } return a / b; } try { let result = divide(10, 0); console.log("Resul
t:", result); } catch (e) { console.log("Error:", e.message); }
```

Summary

Concept	Example Use
`try...catch`	Catch errors (like undefined variables)
`finally`	Run cleanup (always runs)
`throw`	Raise your own error
`Error object`	Holds name, message, and stack info

Final Thoughts

- You don't need to catch *every* small issue — only those that **might fail unpredictably**
- Keep `try...catch` **short and specific**
- Use `throw` to make your code **safer and more predictable**

Reduce

What is `reduce()`?

The `reduce()` method is used to **reduce an array to a single value** — like a sum, product, max value, or even a new object or array.

It works by **applying a function to each element** and carrying forward an **accumulator**.

Syntax

```
array.reduce(callback(accumulator, currentValue), initialValue);
```

- `accumulator`: holds the result of the previous operation
- `currentValue`: the current item in the array
- `initialValue`: (optional) initial value of accumulator

Example 1: Sum of Array

```
const numbers = [1, 2, 3, 4, 5]; const total = numbers.reduce(function (acc, curr) { return acc + curr; }, 0); console.log(total); // Output: 15
```

Explanation:

- Starts with `acc = 0`
- Adds each element to it:

  `0 + 1 = 1`, `1 + 2 = 3`, `3 + 3 = 6` ... etc.

Example 2: Multiply All Numbers

```
const numbers = [2, 3, 4]; const product = numbers.reduce(function (acc, curr) { return acc * curr; }, 1); console.log(product); // Output: 24
```

Example 3: Find Maximum

```
const nums = [10, 45, 22, 100, 8]; const max = nums.reduce(function (acc, cur
r) { return curr > acc ? curr : acc; }); console.log(max); // Output: 100
```

Example 4: Count Occurrences

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"]; con
st count = fruits.reduce(function (acc, fruit) { acc[fruit] = (acc[fruit] ||
0) + 1; return acc; }, {}); console.log(count); // Output: { apple: 3, banan
a: 2, orange: 1 }
```

Example 5: Flatten a 2D Array

```
const arr = [[1, 2], [3, 4], [5, 6]]; const flat = arr.reduce(function (acc,
curr) { return acc.concat(curr); }, []); console.log(flat); // Output: [1, 2,
3, 4, 5, 6]
```

Example 6: Arrow Function Version

```
const nums = [1, 2, 3]; const sum = nums.reduce((acc, curr) => acc + curr,
0); console.log(sum); // Output: 6
```

Summary

Task	Code Example
Sum all items	`arr.reduce((a, c) => a + c, 0)`
Multiply items	`arr.reduce((a, c) => a * c, 1)`
Find max	`arr.reduce((a, c) => c > a ? c : a)`
Count values	See fruit example above
Flatten arrays	`arr.reduce((a, c) => a.concat(c), [])`