# 🧱 JavaScript Error Handling – Full Guide

## 🔍 What is an Error in JavaScript?

In JavaScript, an **error** is an object that represents something went wrong during code execution (syntax issue, runtime issue, etc.).

---

## 🧠 Analogy: Think of Errors Like Car Warnings 🚗

You're driving your car:

- ⚠️ Minor issue → You might still drive (soft warning)
- 🚨 Critical issue → You must stop the car (throw + catch)

**JavaScript's job?** To give you a warning (error) — your job is to catch it & decide what to do.

---

## 📚 Types of Errors in JavaScript

| Type | Example | Meaning |
|---|---|---|
| **SyntaxError** | `if (a > 5` | Invalid code syntax |
| **ReferenceError** | `console.log(x)` (x not declared) | Using undeclared variables |
| **TypeError** | `null.f()` | Wrong type usage (e.g., call on `null`) |
| **RangeError** | `new Array(-1)` | Value out of range |
| **EvalError** | `eval()` misuse (rare) | Improper use of `eval()` |
| **URIError** | `decodeURIComponent('%')` | Malformed URI components |
| **CustomError** | Created manually | Application-specific logic errors |

---

## ✨ 1. `try` / `catch` / `finally` – The Classic Pattern

```
try {
  // 🧪 Risky code here
  let data = JSON.parse("{name: 'Darshan'}"); // ❌ invalid JSON
} catch (error) {
  console.error("❌ Error caught:", error.message); // Handles the error
} finally {
  console.log("✅ Finally block runs always.");
}
```

### 🔍 Explanation:

- `try`: Place risky code here (like parsing, network calls).
- `catch`: This block executes **if an error occurs** in `try`.
- `finally`: Always runs, whether or not there was an error (used for cleanup).

---

## 🎯 2. Throwing Custom Errors

```
function withdraw(amount) {
  if (amount > 1000) {
    throw new Error("❌ Withdrawal limit exceeded!");
  }
  return `✅ Withdrawn: ${amount}`;
}

try {
  withdraw(1500);
} catch (err) {
  console.error("🚨 Custom Error:", err.message);
}
```

---

## 🧰 3. Creating Custom Error Classes

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function validateAge(age) {
  if (age < 18) throw new ValidationError("🚫 Age must be 18+");
}

try {
  validateAge(16);
} catch (e) {
  if (e instanceof ValidationError) {
    console.log("⚠️ Validation issue:", e.message);
  } else {
    throw e; // re-throw unknown errors
  }
}
```

---

## 🌐 4. Async/Await + Try/Catch

```
async function fetchUser() {
  try {
    let response = await fetch('https://invalid.api.com/user');
    let data = await response.json();
    console.log(data);
  } catch (err) {
    console.error("📡 Network Error:", err.message);
  }
}
fetchUser();
```

☑ **Best Practice:** Always use `try/catch` inside `async` functions when using `await`.

---

## ⛓ 5. Error Handling in `.then()` / `.catch()`

```
fetch('https://api.github.com/users/dpvasani')
  .then(response => response.json())
  .then(data => console.log("📦 Data:", data))
  .catch(err => console.error("❌ Error:", err.message));
```

🔁 Often used in older Promise-based code. Prefer `async/await` in modern apps.

---

## 🧱 6. Graceful Fallback (Default Values)

```
function getName(user) {
  return user?.name ?? "👤 Guest";
}

console.log(getName(null)); // Output: 👤 Guest
```

☑ Use **optional chaining (`?.`)** and **nullish coalescing (`??`)** for safe access and fallbacks.

---

## ☑ Best Practices

☑ **Wrap risky code** with `try/catch` ☑ **Use custom error types** for meaningful error messages ☑ **Don't swallow errors** silently (log them) ☑ **Validate user input early** ☑ **Re-throw errors** when necessary to avoid hiding critical bugs ☑ **Use `finally` for cleanup**, closing resources, clearing timers, etc. ☑ **Avoid catching errors too broadly** (`catch (e) {}` without handling)

---

## 🚨 Error Handling in Node.js (Bonus)

```
process.on('uncaughtException', err => {
  console.error("🔥 Uncaught Exception:", err);
  process.exit(1); // exit safely
});

process.on('unhandledRejection', reason => {
  console.error("🔄 Unhandled Promise Rejection:", reason);
});
```

Used for **global error handling** in production apps.

---

## 🔄 Summary Cheat Sheet

| **Pattern** | **Use When** |
|---|---|
| `try/catch/finally` | General error handling |
| `throw new Error()` | Custom application errors |
| Custom Error Classes | Domain-specific error semantics |
| `async/await + catch` | Asynchronous APIs, DB calls, file/network operations |
| `.then().catch()` | Older Promise handling |
| `??` and `?.` | Graceful property access |
| Global handlers | Node.js crash-proofing |

---

## 🎁 Want an Analogy to Remember?

#### 👨‍🍳 Error handling = Kitchen safety

- `try` = Attempt to cook a new dish 🍜
- `catch` = If something burns 🔥, handle it
- `finally` = Always clean the kitchen 🧽, whether or not the dish succeeded
- `throw` = You're the chef — if something's wrong (bad ingredient), say it loudly 🗣️
- `custom Error` = Label the error specifically: e.g., "SpicyLevelExceededError" 🌶️

---

## 🧪 Mandatory vs Optional Checks in JavaScript

### 🚦 1. **Mandatory Checks** – *Fail Fast if Data Is Missing*

Used when a value **must exist**. If it doesn't, throw an error or halt.

#### 🔥 Example: Mandatory Input Check

```
function getUserAge(user) {
  if (!user || !user.age) {
    throw new Error("🚨 User or age is mandatory!");
  }
  return user.age;
}

getUserAge(null); // ❌ Error: User or age is mandatory!
```

✅ **Best for**: Required fields in form data, critical values in configs, etc.

---

## ❓ 2. **Optional Checks** – *Proceed Safely If Data May Be Missing*

Used when data **might be absent**, and it's okay to use a fallback.

### ✨ Example: Optional Chaining (`?.`) + Nullish Coalescing (`??`)

```
const user = null;

const name = user?.profile?.name ?? "👤 Guest";
console.log(name); // 👤 Guest
```

🧠 This **avoids crashing** on undefined/null chains:

- `?.` safely accesses deep properties.
- `??` provides a **fallback** when the value is `null` or `undefined`.

---

## 🧭 Summary Table

| Feature | Use Case | Code Example | Outcome | | | |
|---|---|---|---|---|---|---|
| `if (!value)` | Mandatory presence | `if (!user) throw ...` | ❌ Throws if not present | | | |
| Optional Chaining `?.` | Safe access to nested properties | `user?.address?.city` | ✅ Returns `undefined` safely | | | |
| Nullish Coalescing `??` | Fallback for `null` or `undefined` | `value ?? "default"` | ✅ Replaces `null`/`undefined` | | | |
| ` | | ` | Fallback for falsy (can be risky) | `value | "default"` | ⚠️ Replaces `0`, `""`, `false` |

---

### 🎯 Real-Life Analogy:

#### 🍽️ **Ordering Food Online**

- `try/catch`: You try to place an order – if payment fails, you get an error.
- Mandatory Check: The app won't proceed unless you add a delivery address.
- Optional Check: If you don't add special instructions, it just continues normally.
- `?.`: Check if `order?.specialInstructions` exists — avoid crashing.
- `??`: If `order?.notes ?? "No notes"` — fallback if user didn't write anything.

---

### 🧪 Combine Optional Check with Default Handling

```
function greet(user) {
  const name = user?.name ?? "Guest";
  console.log(`Hello, ${name}!`);
}

greet({ name: "Darshan" }); // Hello, Darshan!
greet(null);                // Hello, Guest!
```

✅ **Safe and readable** — no errors even when `user` is null.

---

### ✅ Best Practices for Optional & Mandatory Checks

| Guideline | Why it matters | | |
|---|---|---|---|
| 🔐 Use mandatory checks for critical values | Prevents broken app logic | | |
| 🔄 Use optional checks for non-critical chains | Prevents runtime crashes | | |
| 🎯 Prefer `??` over ` | | ` when checking for null | Avoids overriding values like `0` or `false` |
| 💡 Use `?.` for deep object/property access | Cleaner and safer code | | |

---

# 🛠️ Backend Error Handling in JavaScript (Express.js)

> ✨ "Clean, consistent, and centralized error management is a hallmark of production-grade backend systems."

---

## 🧩 Components of Error Handling

Your project uses a **modular utility-based** structure with these key components:

---

### 🟢 1. `ApiError` – Custom Error Class

A reusable error blueprint with status, message, data, and stack trace.

📄 **Code:**

```
class ApiError extends Error {
  constructor(statusCode, message = "Something Went Wrong", errors = [], stack =
"") {
    super(message);
    this.statusCode = statusCode;
    this.data = null;
    this.message = message;
    this.success = false;
    this.errors = errors;

    if (stack) this.stack = stack;
    else Error.captureStackTrace(this, this.constructor);
  }
}
export { ApiError };
```

🧠 **Analogy:**

> Like a custom pizza 🍕 — you control the ingredients: code, message, trace, etc.

✅ **Use:**

```
throw new ApiError(404, "User Not Found");
```

---

### ✅ 2. `ApiResponse` – Uniform Success Format

Standardized structure for success responses.

📄 **Code:**

```
class ApiResponse {
  constructor(statusCode, data, message = "Success") {
    this.statusCode = statusCode;
    this.data = data;
    this.message = message;
    this.success = statusCode < 400;
  }
}
export { ApiResponse };
```

✅ **Use:**

```
res.status(200).json(new ApiResponse(200, userData, "User fetched successfully"));
```

---

### 🌀 3. `asyncHandler` – Catch Async Errors Automatically

Wraps any async controller and auto-passes errors to `next()`.

📄 **Code:**

```
const asyncHandler = (requestHandler) => {
  return (req, res, next) => {
    Promise.resolve(requestHandler(req, res, next)).catch((err) => next(err));
  };
};
export { asyncHandler };
```

✅ **Use:**

```
const registerUser = asyncHandler(async (req, res) => {
  // some async logic
});
```

🧠 **Analogy:**

> Like an umbrella ☂ — it catches all raindrops (errors) from async routes!

---

## 🚨 Centralized Error Middleware

Don't forget the Express.js error middleware at the end of your route file:

```
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  res.status(statusCode).json({
    success: false,
    message: err.message || "Internal Server Error",
    errors: err.errors || [],
    stack: process.env.NODE_ENV === "development" ? err.stack : undefined,
  });
});
```

## 🧰 Optional & Mandatory Checks in Practice

### ✔ Mandatory Check Example

```
if (!req.body.email) {
  throw new ApiError(400, "Email is required!");
}
```

### ❓ Optional Check Example

```
const profilePic = req.body?.profile?.pic ?? "default.png";
```

## 🧪 Full Example Route Using All Concepts

```
import { asyncHandler } from "../utils/asyncHandler.js";
import { ApiError } from "../utils/ApiError.js";
import { ApiResponse } from "../utils/ApiResponse.js";

const getUserProfile = asyncHandler(async (req, res) => {
  const userId = req.params.id;

  if (!userId) throw new ApiError(400, "User ID is required");

  const user = await User.findById(userId);
  if (!user) throw new ApiError(404, "User not found");

  res.status(200).json(new ApiResponse(200, user, "User profile fetched"));
});
```

## 📌 Best Practices

| 🏁 **Practice** | ☑ **Why It's Good** |
|---|---|
| Use `ApiError` for all thrown errors | Uniform format across the app |
| Always wrap controllers in `asyncHandler` | Prevents repetitive try/catch |
| Structure success with `ApiResponse` | Predictable responses for frontend |

| 🏁 **Practice** | ✅ **Why It's Good** |
|---|---|
| Log errors in production | Use `winston`, `pino`, or log to file/monitoring services |
| Use environment-specific stacks | Hide `.stack` in production for security reasons |
| Validate request body/query params | Use libraries like `zod`, `joi`, or `express-validator` |

---

## 🚀 Summary

| **Component** | **Purpose** | **Usage** |
|---|---|---|
| `ApiError` | Custom error format | `throw new ApiError(...)` |
| `ApiResponse` | Standard success response | `res.json(new ApiResponse)` |
| `asyncHandler` | Catches all async route errors | `asyncHandler(async () => {})` |
| Error Middleware | Centralized handling & formatting | `app.use((err, req, res...` |

---