```
class AppError extends Error {
  constructor(
    statusCode,
    message = "something went wrong",
    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 {AppError}
```

It looks like you're defining a custom error class for an API. 👍
Overall, this is **solid and usable**, but there are a few **small issues and best-practice improvements** you might want to consider.

What your code is doing well

- Extends the built-in `Error` class correctly
- Calls `super(message)` (important ✅)
- Stores HTTP `statusCode`, `data`, and a `success` flag
- Handles stack traces properly

Minor issues & improvements

1. Typo in default message

```
message = "something went wrong"
```

✅ Should be `"something went wrong"`

2. Redundant assignment of `message`

You already pass `message` to `super(message)`, which sets `this.message`.

So this line is unnecessary:

```
this.message = message
```

It won't break anything, but it's redundant.

3. Set the error name (best practice)

This helps with debugging and logging:

```
this.name = "AppError"
```

4. Optional: Default status code

If someone forgets to pass a status code, you might want a fallback:

```
constructor(
  statusCode = 500,
  message = "something went wrong",
  errors = [],
  stack = ""
)
```

5. Optional: Modern `cause` support (Node 16+)

If this error wraps another error:

```
constructor(statusCode, message, errors = [], cause) {
  super(message, { cause })
}
```

Cleaned up version (recommended)

```
class AppError extends Error {
  constructor(
    statusCode = 500,
    message = "Something went wrong",
    errors = [],
    stack = ""
  ) {
    super(message)

    this.name = "AppError"
    this.statusCode = statusCode
    this.success = false
    this.errors = errors
    this.data = null

    if (stack) {
      this.stack = stack
    } else {
      Error.captureStackTrace(this, this.constructor)
    }
  }
}

export { AppError }
```

Typical usage example

```
throw new AppError(404, "User not found", [{ field: "id", msg: "does not exist"}])
```

If you want, I can also:

- Add Express error handling middleware
- Show how to serialize this error for API responses
- Convert it to TypeScript

Just tell me 🙂

---

explain extend error work

Sure 🙂 Let's break down how `extends Error` works in JavaScript, step by step, in a clear way.

---

1. What is `Error`?

`Error` is a built-in JavaScript class used to represent runtime errors.

Example:

```
const err = new Error("something broken");
console.log(err.message); // "something broken"
console.log(err.stack); // stack trace
```

The `Error` class already provides:

- `message`
- `name`
- `stack`

---

2. What does `extends Error` mean?

When you write:

```
class AppError extends Error {
  ...
}
```

You are saying:

> `AppError` is a special type of `Error`.

This means:

- `AppError` inherits all properties and methods of `Error`
- `AppError` is also an instance of `Error`

Example:

```
const err = new AppError(404, "Not found");

err instanceof AppError // true
err instanceof Error    // true
```

---

3. Why do we call `super(message)`?

Inside a child class, you must call `super()` before using `this`.

```
constructor(message) {
  super(message);
}
```

What `super(message)` does:

- Calls the `Error` constructor
- Sets:
  - `this.message`
  - `this.stack`

If you don't call `super()`:

❌ JavaScript throws an error.

---

4. What happens internally?

Without extending `Error`

```
class AppError {
  constructor(message) {
    this.message = message;
  }
}
```

❌ Problems:

- No stack trace
- `instanceof Error` → `false`
- Not handled correctly by error tools

With `extends Error`

```
class AppError extends Error {
  constructor(message) {
    super(message);
  }
}
```

✅ Benefits:

- Proper stack trace
- Works with `try/catch`
- Recognized as an error everywhere

---

5. Why `Error.captureStackTrace`?

Node.js provides this method to control stack traces:

```
Error.captureStackTrace(this, this.constructor);
```

It:

- Removes constructor internals from the stack trace
- Makes debugging cleaner

Without it, stack traces are messy.

---

6. Full lifecycle example

```
class AppError extends Error {
  constructor(statusCode, message) {
    super(message); // Error constructor runs

    this.name = "AppError"; // ✅ custom name
    this.statusCode = statusCode; // ✅ custom data

    Error.captureStackTrace(this, this.constructor); // ✅ clean stack
  }
}

throw new AppError(404, "Unauthorized");
```

What JavaScript creates:

```
{
  name: "AppError",
  message: "Unauthorized",
  statusCode: 404,
  stack: "AppError: Unauthorized\n    at ..."
}
```

---

7. Why extend `Error` instead of using plain objects?

Feature	Plain Object	`extends Error`
Stack trace	❌	✅
`try/catch`	⚠	✅
`instanceof Error`	❌	✅
Debugging tools	❌	✅

---

8. Simple mental model 🧠

Think of it like:

- `Error` = base template
- `AppError` = `Error` + your API-specific data

If you want, I can explain:

- `super()` vs `this`
- Why extending built-ins used to be tricky
- How Express handles it in Express error middleware

Just ask 🙂

---

super activate error constructor and this: It denoted current instance

---

Yes — your understanding is **basically correct** 👍
Let me restate it clearly and precisely, and fix the wording a bit.

Correct explanation (clean version)

`super()` calls (activates) the parent (`Error`) constructor, and `this` refers to the current instance of the child class (`AppError`).

How it works step by step

1️⃣ **When you create an instance**

```
new AppError(404, "Not found")
```

JavaScript does this:

- Creates a new empty object
- That object will become the current instance

2️⃣ **`super(message)` runs first**

```
super(message)
```

This:

- Calls the `Error` constructor
- Initializes:
  - `this.message`
  - `this.stack`
  - internal error properties

⚠ **Important rule:**
You cannot use `this` before calling `super()` in a subclass.

This is why this would crash:

```
this.statusCode = statusCode // ❌ before super()
super(message)
```

3️⃣ **`this` becomes usable**

After `super()` finishes:

```
this.name = "AppError"
```

Now you can safely add:

```
this.statusCode = statusCode
this.success = false
```

Visual mental model 🧠

```
new AppError()
  |
  v
create empty object = this
  |
call error constructor via super()
  |
attach error properties (message, stack)
  |
return to AppError constructor
  |
add AppError properties
```

Short precise definitions

- `super()`
  - Calls the parent class constructor
  - Initializes inherited properties
  - Unlocks `this` in child constructor
- `this`
  - Refers to the current instance being created
  - In your case an `AppError` object

One-line takeaway

`super()` initializes the `Error` part of the object, and `this` represents the same object as it is being built by the child class.

If you want, I can explain this using **prototype chains** or show what breaks if `super()` is missing.

---

what inside the constructor of error

Good question 👍
Let's look at what the `Error` constructor **actually does** (conceptually), because its real implementation is hidden inside the JavaScript engine.

Short answer

The `Error` constructor mainly does **three things**:

1. Sets the error message
2. Sets the error name
3. Captures the stack trace (where the error happened)

What happens when you run:

```
new Error("something went wrong")
```

Internally (conceptually), JavaScript does something like this:

```
function Error(message) {
  this.name = "Error";
  this.message = message || "";
  this.stack = captureStackTrace();
}
```

⚠ This is not the real source code, but it's a very close mental model.

1 `this.message`

```js
this.message = message;
```

So when you write:

```js
super("Not Found");
```

You get:

```js
error.message === "Not Found"
```

---

2 `this.name`

By default:

```js
this.name = "Error";
```

That's why normal errors look like:

```txt
Error: Something went wrong
```

When you extend it:

```js
this.name = "ApiError";
```

It becomes:

```txt
ApiError: Not Found
```

---

3 `this.stack`

The engine records:

- File name
- Line number
- Call stack (function calls)

Example:

```txt
ApiError: Not Found
  at loginController (auth.js:12)
  at router (index.js:5)
```

This is **automatically created** when `Error` is constructed.

Why `super(message)` is so important

If you **don't** call `super(message)`:

❌ No `message`
❌ No `stack`
❌ Not treated as a real error

Example (broken):

```js
class ApiError extends Error {
  constructor(message) {
    // super(message) ❌ missing
    this.message = message; // crash
  }
}
```

JavaScript throws:

```txt
ReferenceError: Must call super constructor before accessing 'this'
```

What `Error.captureStackTrace` adds (Node.js)

```js
Error.captureStackTrace(this, this.constructor);
```

It:

- Removes the constructor itself from the stack
- Makes the stack cleaner and more useful

Without it, you might see internal constructor lines.

What `Error` does NOT do

- ❌ It does not log anything
- ❌ It does not stop the program
- ❌ It does not send HTTP responses

It only **creates an error object**.

Throwing the error is separate:

```js
throw new Error("Boom");
```

Final mental model 🧠

> **`Error` constructor = initialize message + name + stack trace**

Your subclass just **adds more information** on top of that.

If you want next, I can explain:

- How stack traces are generated
- Difference between `Error`, `TypeError`, `ReferenceError`
- Why `instanceof Error` works

Just tell me 😊

---

mean super activate error class constructor and error constructor allow us to set this.message this.name etc

---

Yes — that statement is **correct** 👍
Let me rewrite it in **clear, correct English**, then explain it briefly.

Corrected sentence

> **`super()` activates (calls) the `Error` class constructor, and the `Error` constructor initializes properties like `this.message`, `this.name`, and `this.stack`.**

Why this is true

When you write:

```js
class ApiError extends Error {
  constructor(message) {
    super(message);
  }
}
```

What happens:

1. `super(message)` calls the `Error` **constructor**
2. The `Error` constructor:
   - Sets `this.message = message`
   - Sets `this.name = "Error"`
   - Creates `this.stack`
3. Control returns to `ApiError`
4. `this` now refers to the **current ApiError instance**

After `super()`, you can customize

```js
this.name = "ApiError";
this.statusCode = 404;
```

Now the object contains:

```js
{
  name: "ApiError",
  message: "Not Found",
  statusCode: 404,
  stack: "..."
}
```

Important rule to remember ⚠️

> **You cannot use `this` before calling `super()` in a subclass constructor.**

This is because `this` does not exist until the parent constructor runs.

One-line takeaway

> **`super()` runs the `Error` constructor, which sets up the error object, and then the child class extends it with extra properties.**

Your understanding is exactly right.
If you want, I can explain this using **prototypes** or show what happens internally if `super()` is missing.

Ask anything
Voice
ChatGPT can make mistakes. Check important info. See Cookie Preferences.