26th October 2019

Custom errors, extending Error

When we develop something, we often need our own error classes to reflect specific things that may go wrong in our tasks. For errors in network operations we may need `HttpError`, for database operations `DbError`, for searching operations `NotFoundError` and so on.

Our errors should support basic error properties like `message`, `name` and, preferably, `stack`. But they also may have other properties of their own, e.g. `HttpError` objects may have a `statusCode` property with a value like `404` or `403` or `500`.

JavaScript allows to use `throw` with any argument, so technically our custom error classes don't need to inherit from `Error`. But if we inherit, then it becomes possible to use `obj instanceof Error` to identify error objects. So it's better to inherit from it.

As the application grows, our own errors naturally form a hierarchy. For instance, `HttpTimeoutError` may inherit from `HttpError`, and so on.

Extending Error

As an example, let's consider a function `readUser(json)` that should read JSON with user data.

Here's an example of how a valid `json` may look:

```
let json = `{ "name": "John", "age": 30 }`;
```

Internally, we'll use `JSON.parse`. If it receives malformed `json`, then it throws `SyntaxError`. But even if `json` is syntactically correct, that doesn't mean that it's a valid user, right? It may miss the necessary data. For instance, it may not have `name` and `age` properties that are essential for our users.

Our function `readUser(json)` will not only read JSON, but check ("validate") the data. If there are no required fields, or the format is wrong, then that's an error. And that's not a `SyntaxError`, because the data is syntactically correct, but another kind of error. We'll call it `ValidationError` and create a class for it. An error of that kind should also carry the information about the offending field.

Our `ValidationError` class should inherit from the built-in `Error` class.

That class is built-in, but here's its approximate code so we can understand what we're extending:

```
// The "pseudocode" for the built-in Error class defined by JavaScript itself
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (different names for different built-in error cla
    this.stack = <call stack>; // non-standard, but most environments support
  }
}
```

Now let’s inherit `ValidationError` from it and try it in action:

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}

function test() {
  throw new ValidationError("Whoops!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Whoops!
  alert(err.name); // ValidationError
  alert(err.stack); // a list of nested calls with line numbers for each
}
```

Please note: in the line `(1)` we call the parent constructor. JavaScript requires us to call `super` in the child constructor, so that’s obligatory. The parent constructor sets the `message` property.

The parent constructor also sets the `name` property to `"Error"`, so in the line `(2)` we reset it to the right value.

Let’s try to use it in `readUser(json)`:

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

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }

  return user;
}

// Working example with try..catch

```

```
try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No field: name
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it (**)
  }
}
```

The `try..catch` block in the code above handles both our `ValidationError` and the built-in `SyntaxError` from `JSON.parse`.

Please take a look at how we use `instanceof` to check for the specific error type in the line `(*)`.

We could also look at `err.name`, like this:

```
// ...
// instead of (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

The `instanceof` version is much better, because in the future we are going to extend `ValidationError`, make subtypes of it, like `PropertyRequiredError`. And `instanceof` check will continue to work for new inheriting classes. So that's future-proof.

Also it's important that if `catch` meets an unknown error, then it rethrows it in the line `(**)`. The `catch` block only knows how to handle validation and syntax errors, other kinds (due to a typo in the code or other unknown ones) should fall through.

Further inheritance

The `ValidationError` class is very generic. Many things may go wrong. The property may be absent or it may be in a wrong format (like a string value for `age`). Let's make a more concrete class `PropertyRequiredError`, exactly for absent properties. It will carry additional information about the property that's missing.

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

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

```

```
// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Working example with try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No property: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

The new class `PropertyRequiredError` is easy to use: we only need to pass the property name: `new PropertyRequiredError(property)`. The human-readable `message` is generated by the constructor.

Please note that `this.name` in `PropertyRequiredError` constructor is again assigned manually. That may become a bit tedious – to assign `this.name = <class name>` in every custom error class. We can avoid it by making our own “basic error” class that assigns `this.name = this.constructor.name`. And then inherit all our custom errors from it.

Let’s call it `MyError`.

Here’s the code with `MyError` and other custom error classes, simplified:

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

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.property = property;
  }
}
```

```

// name is correct
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError
```

Now custom errors are much shorter, especially `ValidationError`, as we got rid of the `"this.name = ..."` line in the constructor.

Wrapping exceptions

The purpose of the function `readUser` in the code above is "to read the user data". There may occur different kinds of errors in the process. Right now we have `SyntaxError` and `ValidationError`, but in the future `readUser` function may grow and probably generate other kinds of errors.

The code which calls `readUser` should handle these errors. Right now it uses multiple `if` s in the `catch` block, that check the class and handle known errors and rethrow the unknown ones. But if the `readUser` function generates several kinds of errors, then we should ask ourselves: do we really want to check for all error types one-by-one in every code that calls `readUser`?

Often the answer is "No": the outer code wants to be "one level above all that", it just wants to have some kind of "data reading error" – why exactly it happened is often irrelevant (the error message describes it). Or, even better, it could have a way to get the error details, but only if we need to.

So let's make a new class `ReadError` to represent such errors. If an error occurs inside `readUser`, we'll catch it there and generate `ReadError`. We'll also keep the reference to the original error in its `cause` property. Then the outer code will only have to check for `ReadError`.

Here's the code that defines `ReadError` and demonstrates its use in `readUser` and `try..catch`:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Syntax Error", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Validation Error", err);
    } else {
      throw err;
    }
  }

}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // Original error: SyntaxError: Unexpected token b in JSON at position 1
    alert("Original error: " + e.cause);
  } else {
    throw e;
  }
}
```

In the code above, `readUser` works exactly as described – catches syntax and validation errors and throws `ReadError` errors instead (unknown errors are rethrown as usual).

So the outer code checks `instanceof ReadError` and that's it. No need to list all possible error types.

The approach is called "wrapping exceptions", because we take "low level exceptions" and "wrap" them into `ReadError` that is more abstract and more convenient to use for the calling code. It is widely used in object-oriented programming.

Summary

- We can inherit from `Error` and other built-in error classes normally. We just need to take care of the `name` property and don't forget to call `super`.
- We can use `instanceof` to check for particular errors. It also works with inheritance. But sometimes we have an error object coming from a 3rd-party library and there's no easy way to get its class. Then `name` property can be used for such checks.
- Wrapping exceptions is a widespread technique: a function handles low-level exceptions and creates higher-level errors instead of various low-level ones. Low-level exceptions sometimes become properties of that object like `err.cause` in the examples above, but that's not strictly required.

Tasks

Inherit from SyntaxError

importance: 5

Create a class `FormatError` that inherits from the built-in `SyntaxError` class.

It should support `message`, `name` and `stack` properties.

Usage example:

```
let err = new FormatError("formatting error");

alert( err.message ); // formatting error
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof FormatError ); // true
alert( err instanceof SyntaxError ); // true (because inherits from SyntaxErr
```

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)