Chapter 97: Error Handling

Section 97.1: Error objects

Runtime errors in JavaScript are instances of the `Error` object. The `Error` object can also be used as-is, or as the base for user-defined exceptions. It's possible to throw any type of value - for example, strings - but you're strongly encouraged to use `Error` or one of its derivatives to ensure that debugging information -- such as stack traces -- is correctly preserved.

The first parameter to the `Error` constructor is the human-readable error message. You should try to always specify a useful error message of what went wrong, even if additional information can be found elsewhere.

```
try {
   throw new Error('Useful message');
} catch (error) {
   console.log('Something went wrong! ' + error.message);
}
```

Section 97.2: Interaction with Promises

Version ≥ 6

Exceptions are to synchronous code what rejections are to promise-based asynchronous code. If an exception is thrown in a promise handler, its error will be automatically caught and used to reject the promise instead.

```
Promise.resolve(5)
    .then(result => {
        throw new Error("I don't like five");
    })
    .then(result => {
        console.info("Promise resolved: " + result);
    })
    .catch(error => {
        console.error("Promise rejected: " + error);
    });
```

```
Promise rejected: Error: I don't like five
```

Version > 7

The async functions proposal—expected to be part of ECMAScript 2017—extends this in the opposite direction. If you await a rejected promise, its error is raised as an exception:

```
async function main() {
  try {
    await Promise.reject(new Error("Invalid something"));
  } catch (error) {
    console.log("Caught error: " + error);
  }
}
main();
```

```
Caught error: Invalid something
```

Section 97.3: Error types

There are six specific core error constructors in JavaScript:

- **`EvalError`** - creates an instance representing an error that occurs regarding the global function `eval()`.
- **`InternalError`** - creates an instance representing an error that occurs when an internal error in the JavaScript engine is thrown. E.g. "too much recursion". (Supported only by **Mozilla Firefox**)
- **`RangeError`** - creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range.
- **`ReferenceError`** - creates an instance representing an error that occurs when dereferencing an invalid reference.
- **`SyntaxError`** - creates an instance representing a syntax error that occurs while parsing code in `eval()`.
- **`TypeError`** - creates an instance representing an error that occurs when a variable or parameter is not of a valid type.
- **`URIError`** - creates an instance representing an error that occurs when `encodeURI()` or `decodeURI()` are passed invalid parameters.

If you are implementing error handling mechanism you can check which kind of error you are catching from code.

```
try {
    throw new TypeError();
}
catch (e){
    if(e instanceof Error){
        console.log('instance of general Error constructor');
    }

    if(e instanceof TypeError) {
        console.log('type error');
    }
}
```

In such case `e` will be an instance of `TypeError`. All error types extend the base constructor `Error`, therefore it's also an instance of `Error`.

Keeping that in mind shows us that checking `e` to be an instance of `Error` is useless in most cases.

Section 97.4: Order of operations plus advanced thoughts

Without a try catch block, undefined functions will throw errors and stop execution:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Will throw an error and not run the second line:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

You need a try catch block, similar to other languages, to ensure you catch that error so code can continue to

execute:

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Now, we've caught the error and can be sure that our code is going to execute

```
// An error occurred! ReferenceError: undefinedFunction is not defined(...)
// The code-block has finished
// I will run because we caught the error!
```

What if an error occurs in our catch block!?

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    otherUndefinedFunction("Uh oh... ");
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

We won't process the rest of our catch block, and execution will halt except for the finally block.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(...)
```

You could always nest your try catch blocks.. but you shouldn't because that will get extremely messy..

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    try {
        otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
        console.log("Too much nesting is bad for my heart and soul...");
    }
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Will catch all errors from the previous example and log the following:

```
//Too much nesting is bad for my heart and soul...
//An error occurred! ReferenceError: undefinedFunction is not defined(...)
//The code-block has finished
//I will run because we caught the error!
```

So, how can we catch all errors!? For undefined variables and functions: you can't.

Also, you shouldn't wrap every variable and function in a try/catch block, because these are simple examples that will only ever occur once until you fix them. However, for objects, functions and other variables that you know exist, but you don't know whether their properties or sub-processes or side-effects will exist, or you expect some error states in some circumstances, you should abstract your error handling in some sort of manner. Here is a very basic example and implementation.

Without a protected way to call untrusted or exception throwing methods:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
try {
    foo(1, 2, 3);
} catch(e) {
    try {
        foo(4, 5, 6);
    } catch(e2) {
        console.log("We had to nest because there's currently no other way...");
    }
    console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// Error: custom error!(...)
```

And with protection:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
function protectedFunction(fn, ...args) {
    try {
        fn.apply(this, args);
    } catch (e) {
        console.log("caught error: " + e.name + " -> " + e.message);
    }
}

protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

We catch errors and still process all the expected code, though with a somewhat different syntax. Either way will work, but as you build more advanced applications you will want to start thinking about ways to abstract your error handling.