Lecture 10.1

Error Handling

Error Handling

Error Handling

No matter how great we are at programming, sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user input, an erroneous server response, and for a thousand other reasons.

Usually, a script “dies” (immediately stops) in case of an error, printing it to console.

But there’s a syntax construct try..catch that allows us to “catch” errors so the script can, instead of dying, do something more reasonable.

try..catch

The try..catch syntax

The try..catch construct has two main blocks: try, and then catch:

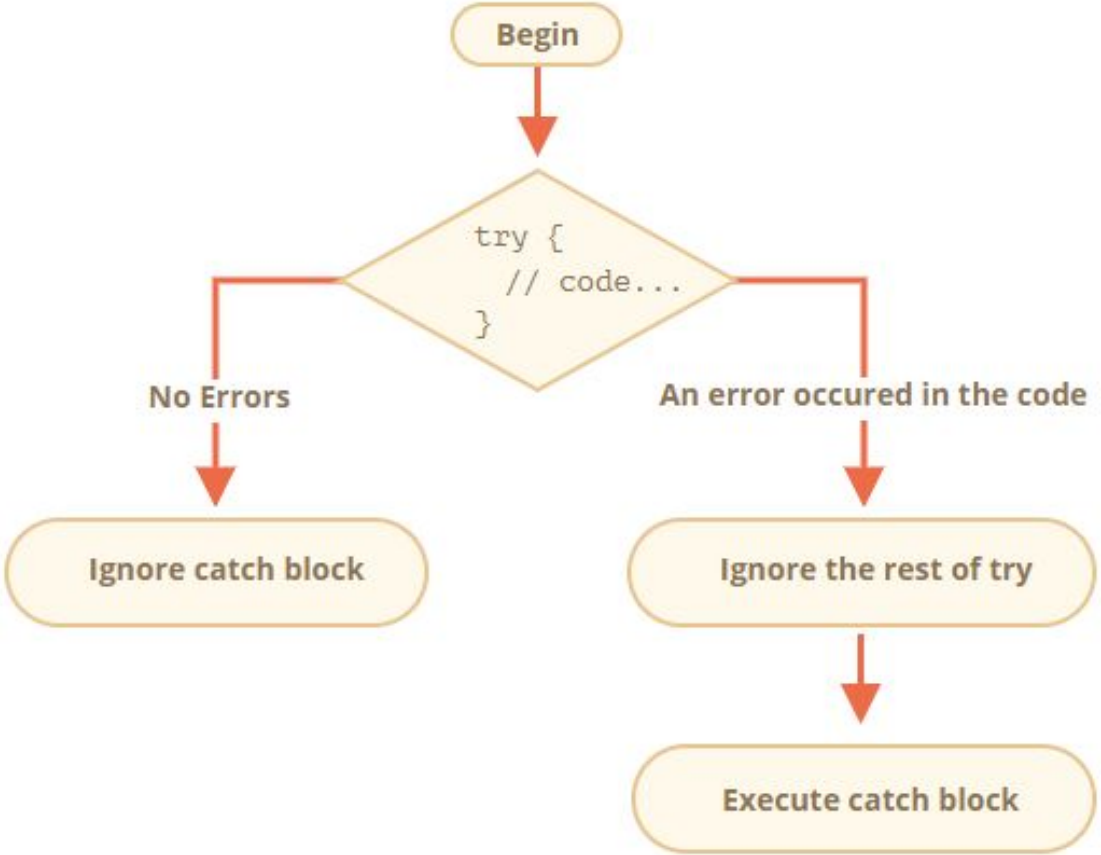
```
try {

  // code...

} catch (err) {

  // error handling

}
```

It works like this:

1. First, the code in try {...} is executed.

2. If there were no errors, then catch(err) is ignored: the execution reaches the end of try and goes on, skipping catch.

3. If an error occurs, then the try execution is stopped, and control flows to the beginning of catch(err). The err variable (we can use any name for it) will contain an error object with details about what happened.



So, an error inside the try {...} block does not kill the script – we have a chance to handle it in catch.

Errorless Program

This errorless example will show alert (1) and (2):

```
try {

    alert('Start of try runs');  // (1) <--

    // ...no errors here

    alert('End of try runs');   // (2) <--

  } catch(err) {

    alert('Catch is ignored, because there are no errors'); // (3)

  }
```

Program with an Error

An example with an error: shows (1) and (3):

```
try {

    alert('Start of try runs');  // (1) <--

    // ...no errors here

    alert('End of try runs');   // (2) <--

  } catch(err) {

    alert('Catch is ignored, because there are no errors'); // (3)

  }
```

For try..catch to work, the code must be runnable. In other words, it should be valid JavaScript.

It won't work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
    {{{{{{{{{{{{
  } catch(e) {
    alert("The engine can't understand this code, it's invalid");
  }
```

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called "parse-time" errors and are unrecoverable (from inside that code). That's because the engine can't understand the code.

So, try..catch can only handle errors that occur in valid code. Such errors are called "runtime errors" or, sometimes, "exceptions".

Try..catch only works for runtime errors!

Try..catch works synchronously

If an exception happens in “scheduled” code, like in setTimeout, then try..catch won’t catch it:

```
try {
    setTimeout(function() {
      noSuchVariable; // script will die here
    }, 1000);
  } catch (e) {
    alert( "won't work" );
  }
```

That’s because the function itself is executed later, when the engine has already left the try..catch construct.

Try..catch works synchronously cont..

To catch an exception inside a scheduled function, try..catch must be inside that function::

```
setTimeout(function() {
  try {
    noSuchVariable; // try..catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

Optional “catch” binding

⚠️ **A recent addition**

This is a recent addition to the language. Old browsers may need polyfills.

If we don’t need error details, `catch` may omit it:

```
try {
  // ...
} catch { // <-- without (err)
  // ...
}
```

Error Object

Error Object

When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to `catch`:

The “error object”, (err in this example), can use another word instead of err.

```
try {
  // ...
} catch(err) {
  // ...
}
```

Error Object cont..

For all built-in errors, the error object has two main properties:

name

Error name. For instance, for an undefined variable that's "ReferenceError".

message

Textual message about error details.

There are other non-standard properties available in most environments. One of most widely used and supported is:

stack

Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

Throwing our own Errors

To unify error handling, have the throw operator. The throw operator generates an error.

```
throw <error object>
```

Technically, we can use anything as an error object. That may be even a primitive, like a number or a string, but it's better to use objects, preferably with name and message properties (to stay somewhat compatible with built-in errors).

Built-in Errors

JavaScript has many built-in constructors for standard errors: Error, SyntaxError, ReferenceError, TypeError and others. We can use them to create error objects as well.

```
let error = new SyntaxError(message);
let error = new ReferenceError(message);
```

For built-in errors (not for any objects, just for errors), the name property is exactly the name of the constructor. And message is taken from the argument.

```
let error = new Error("Things happen o_O");

alert(error.name); // Error
alert(error.message); // Things happen o_O
```

try..catch..finally

try..catch..finally

Wait, that's not all! The try..catch construct may have one more code clause: finally.

If it exists, it runs in all cases:

- after try, if there were no errors,
- after catch, if there were errors.

```
try {
   //... try to execute the code ...
} catch(e) {
   //... handle errors ...
} finally {
   //... execute always ...
}
```

Summary

The try..catch construct allows to handle runtime errors. It literally allows to “try” running the code and “catch” errors that may occur in it.

```
try {
    // run this code
  } catch(err) {
    // if an error happened, then jump here
    // err is the error object
  } finally {
    // do in any case after try/catch
  }
```

There may be no catch section or no finally, so shorter constructs try..catch and try..finally are also valid.

Error objects have following properties:

- message – the human-readable error message.
- name – the string with error name (error constructor name).
- stack (non-standard, but well-supported) – the stack at the moment of error creation.

Summary

Error objects have following properties:

- message – the human-readable error message.
- name – the string with error name (error constructor name).
- stack (non-standard, but well-supported) – the stack at the moment of error creation.

If an error object is not needed, we can omit it by using catch { instead of catch(err) {.

We can also generate our own errors using the throw operator. Technically, the argument of throw can be anything, but usually it's an error object inheriting from the built-in Error class. More on extending errors in the next chapter.

Rethrowing is a very important pattern of error handling: a catch block usually expects and knows how to handle the particular error type, so it should rethrow errors it doesn't know.

Even if we don't have try..catch, most environments allow us to setup a "global" error handler to catch errors that "fall out". In-browser, that's window.onerror.

Summary cont..