# 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 many 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 to “catch” errors and, instead of dying, do something more reasonable

# The try/catch Syntax

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

```
try {
    // code…

} catch (err) {
    // error handling
}
```

  - First, the code in try {...} is executed
  - If there were no errors, then catch(err) is ignored: the execution reaches the end of try and then jumps over catch
  - If an error occurs, then try execution is stopped, and the control flows to the beginning of catch(err)
    - The err variable (can use any name for it) contains an error object with details about what's happened

# try/catch Example

- An example for a runtime error that is caught in the catch block:

```
try {
    alert('Start of try runs');
    lalala; // error, variable is not defined!
    alert('End of try (never reached)');
} catch (err) {
    alert('Error has occurred!');
}

alert('...Then the execution continues');
```

- try..catch only works for runtime errors or “exceptions”
  - It won’t work if the code is syntactically wrong, e.g. it has unmatched curly braces:

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

# 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:

```
try {
    // ...
} catch (err) { // <-- the "error object", could use another word instead of err
    // ...
}
```

- The error object has the following properties:
  - **name** – the error name, e.g., “SyntaxError”, “ReferenceError”, “TypeError”
  - **message** – textual message about error details
  - **stack** – current call stack: a string with information about the sequence of nested calls that led to the error
    - Used for debugging purposes

# Error Object Example

```
try {
    lalala; // error, variable is not defined!
} catch (err) {
    alert(err.name); // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack); // ReferenceError: lalala is not defined at ...

    // Can also show an error as a whole
    // The error is converted to string as "name: message"
    alert(err); // ReferenceError: lalala is not defined
}
```

# Using try/catch

- Let's explore a real-life use case of try..catch
- As we already know, JavaScript supports the **JSON.parse**(str) method to read JSON-encoded values
  - Usually it's used to decode data received over the network, from the server or another source
- We receive it and call JSON.parse, like this:

```
let json = '{"name": "John", "age": 30}'; // data from the server

let user = JSON.parse(json); // convert the text representation to JS object

// now user is an object with properties from the string
alert(user.name); // John
alert(user.age);  // 30
```

- If json is malformed, JSON.parse generates an error, and the script "dies"
  - This way, if something's wrong with the data, the visitor will never know that (unless he opens developer console)

# Using try/catch

- Let's use try..catch to handle the error:

```
let json = "{ bad json }";

try {
    let user = JSON.parse(json); // <-- when an error occurs...
    alert(user.name); // doesn't work
} catch (e) {
    // ...the execution jumps here
    alert("Our apologies, the data has errors, we'll try to request it once more.");
    alert(e.name); // SyntaxError
    alert(e.message); // Unexpected token o in JSON at position 0
}
```

  - Here we use the catch block only to show an error message, but we can do much more: send a new network request, suggest an alternative to the visitor, send information about the error to a logging facility, etc.

# Throwing Errors

- We can throw our own errors
- The throw operator generates an error
- The syntax is:

```
throw <error object>
```

- We can throw anything as an error object
- That may be even a primitive, like a number or a string, but it's better to use objects, preferrably with name and message properties
- We can also throw one of JavaScript built-in error objects
- Besides the generic Error constructor, there are seven other core error constructors: SyntaxError, TypeError, EvalError, InternalError, RangeError, ReferenceError, URIError
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

# Throwing Errors Example

- Let's say that we get a json object that is syntactically correct, but doesn't have a required name property
- We can treat the absence of name as a syntax error
- So let's throw a SyntaxError exception:

```
let json = '{ "age": 30 }'; // incomplete data

try {
    let user = JSON.parse(json); // <-- no errors

    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name"); // passing the message to the constructor
    }
    alert(user.name);
} catch (e) {
    alert("JSON Error: " + e.message); // JSON Error: Incomplete data: no name
}
```

# Rethrowing Errors

- In the example above we use try..catch to handle incorrect data
- But is it possible that *another unexpected error* occurs within the try {...} block
- **Catch should only process errors that it knows and "rethrow" all others**
- The "rethrowing" technique can be explained in more detail as:
  - Catch gets all errors
  - In catch(err) {...} block we analyze the error object err
  - If we don't know how to handle it, then we do throw err

# Rethrowing Errros Example

```
let json = '{ "age": 30 }'; // incomplete data
try {
    let user = JSON.parse(json);
    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name");
    }
    blabla(); // unexpected error
    alert(user.name);
} catch (e) {
    if (e instanceof SyntaxError) {
        alert("JSON Error: " + e.message);
    } else {
        throw e; // rethrow the error (*)
    }
}
```

- The error throwing on line (*) from inside the catch block “falls out” of try..catch and can be either caught by an outer try..catch construct (if it exists), or it kills the script

# try/catch/finally

- 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
- The extended syntax looks like this:

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

- The finally clause is often used when we start doing something before try..catch and want to finalize it in any case of outcome

# try/catch/finally Example

- For instance, let's say we want to measure the time that a Fibonacci numbers function **fib**(n) takes
- We can start measuring before it runs and finish afterwards
- But what if there's an error during the function call? e.g., if the function receives negative or non-integer numbers
- The finally clause is a great place to finish the measurements no matter what
- In the code on the next slide **finally** guarantees that the time will be measured correctly in both situations – in case of a successful execution of **fib** and in case of an error in it

# try/catch/finally Example

```
let num = +prompt("Enter a positive integer number?", 35);
let diff, result;

function fib(n) {
    if (n < 0 || Math.trunc(n) != n) {
        throw new Error("Must not be negative, and also an integer.");
    }
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
    result = fib(num);
} catch (e) {
    result = 0;
} finally {
    diff = Date.now() - start;
}

alert(result || "error occured");
alert(`execution took ${diff}ms`);
```

# finally and return

- The finally clause works for *any* exit from try..catch
  - That includes an explicit return
- In the example below, there's a return in try
  - In this case, finally is executed just before the control returns to the outer code

```
function func() {
    try {
        return 1;
    } catch (e) {
        /* ... */
    } finally {
        alert('finally');
    }
}

alert(func()); // first works alert from finally, and then this one
```

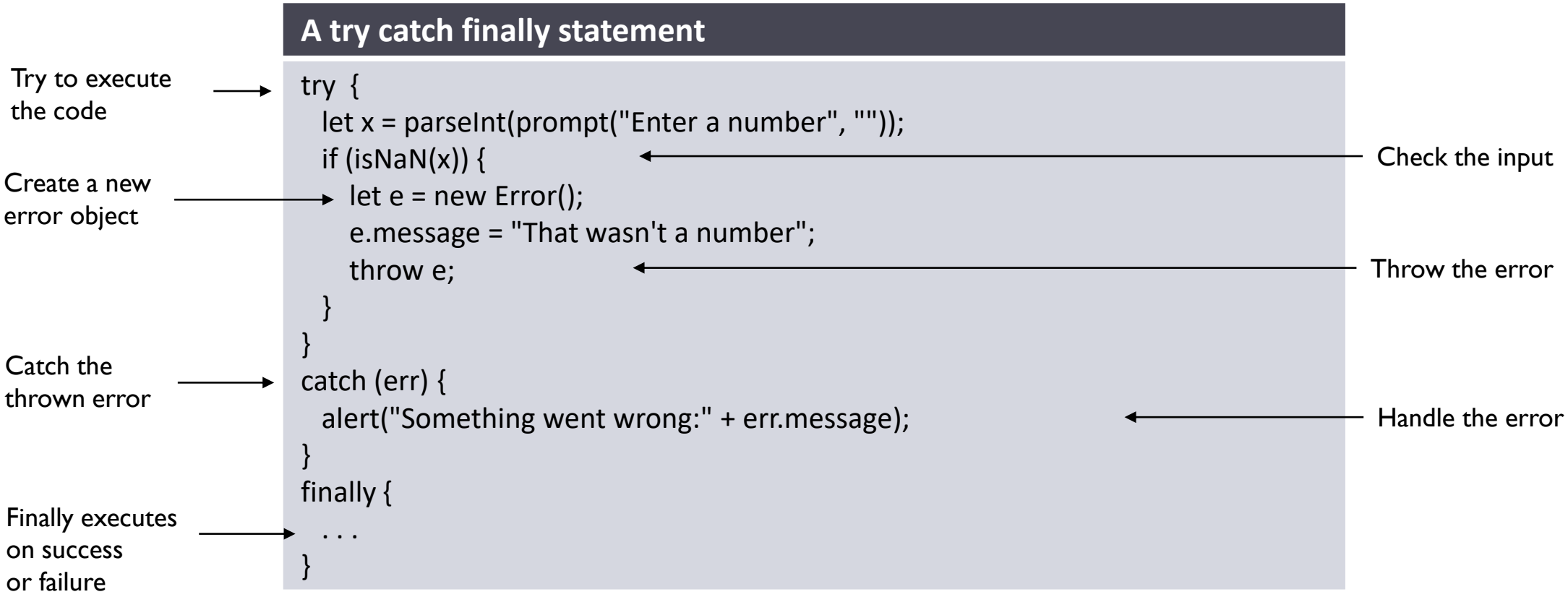
# Global Catch

- Let's imagine we've got a fatal error outside of try..catch, and the script died
- Is there a way to react on such occurrences? We may want to log the error, show something to the user, etc.
- If we run the JavaScript code in a browser, we can assign a function to a special **window.onerror** property, that will run in case of an uncaught error

```
window.onerror = function(message, url, line, col, error) {
    // ...
};
```

- There are also web-services that provide error-logging for such cases, like https://errorception.com or http://www.muscula.com
  - You register at the service and get a piece of JS (or a script URL) from them to insert on pages
  - That JS script has a custom window.onerror function
  - When an error occurs, it sends a network request about it to the service

# Error Handling – Summary



# Custom Errors

- When we develop an application, we often need our own error classes to reflect specific problems that may occur in our tasks
  - For errors in network operations we may need HttpError, for database operations DbError, etc.
- Our errors should support basic error properties like message, name and stack
- But they also may have other properties of their own
  - e.g. HttpError objects may have statusCode property with a value like 404 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 from Error, then it becomes possible to use obj instanceof Error to identify error objects. So it's better to inherit from it.
- As we build our application, 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
- Internally, it will 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
  - For instance, it may not have some required properties, such as name and age
- Our function readUser(json) will not only read JSON, but check ("validate") the data
- If there are no required fields, it will throw a ValidationError, which will carry the information about the offending field

# Extending Error

- Our ValidationError class should inherit from the built-in Error class
- The Error class's code looks something like this:

```
// 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 classes
        this.stack = <nested calls>; // non-standard, but most environments support it
    }
}
```

- Now we will inherit ValidationError from it:

```
class ValidationError extends Error {
    constructor(message) {
        super(message); // the parent constructor sets the message property
        this.name = "ValidationError"; // reset the name property to its right value
    }
}
```

# Extending Error

- Let's try to use it in readUser(json):

```
// 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()
- If it meets an unknown error, then it rethrows it, since the catch only knows how to handle validation and syntax errors, other kinds should fall through

# Exercise (31)

- 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)
- Create a more concrete class PropertyRequiredError, exactly for absent properties
- It should inherit from ValidationError and add the property “missingProperty” to it
- Set its error message to be “No property: [name of the missing property]”

```
class PropertyRequiredError extends ValidationError {
    // Your code here
}

function readUser(json) {
    let user = JSON.parse(json);
    if (!user.age) {
        throw new PropertyRequiredError("age");
    }
    if (!user.name) {
        throw new PropertyRequiredError("name");
    }
    return user;
}
```

```
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.missingProperty); // name
    } else if (err instanceof SyntaxError) {
        alert("JSON Syntax Error: " + err.message);
    } else {
        throw err; // unknown error, rethrow it
    }
}
```

# Promise

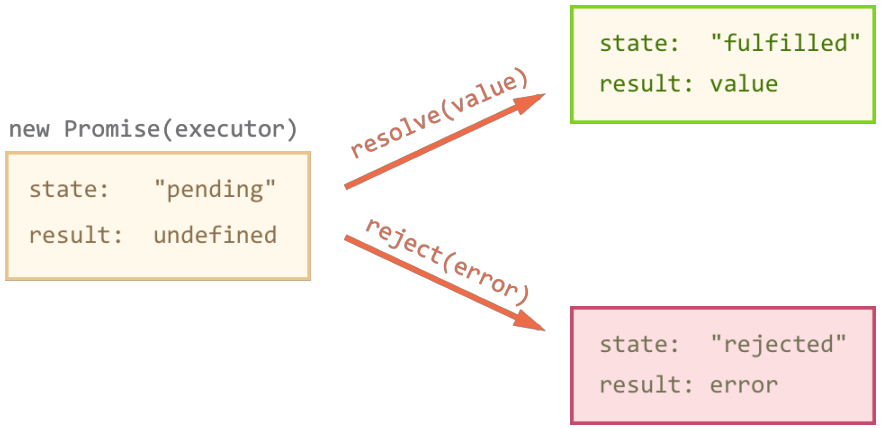
- Often in programming we have a “producing code” that does something that needs time (e.g., load a remote script) and a “consuming code” that wants the result when it’s ready
- A **promise** is a special JavaScript object that links them together
- The producing code creates the promise and gives to everyone who needs the result, so that they can subscribe for the result
- The constructor syntax for a promise object is:

```
let promise = new Promise(function (resolve, reject) {
    // executor (the producing code)
});
```

- The function passed to new Promise is called **executor**
  - When the promise is created, the executor is called immediately
  - It contains the producing code, that should eventually finish with a result

# Promise

- The resulting promise object has internal properties:
  - **state** – initially is “pending”, then changes to “fulfilled” or “rejected”
  - **result** – the result of the computation, initially undefined
- When the executor finishes its job, it should call one of:
  - **resolve(value)** – to indicate that the job finished successfully:
    - sets state to "fulfilled"
    - sets result to value
  - **reject(error)** – to indicate that an error occurred:
    - sets state to "rejected"
    - sets result to error



# Promise Example

- An example for a simple executor:

```
let promise = new Promise(function (resolve, reject) {
    // after 1 second signal that the job is done with the result "done!"
    setTimeout(() => resolve("done!"), 1000);
});
```

- And now an example where the executor rejects promise with an error:

```
let promise = new Promise(function (resolve, reject) {
    // after 1 second signal that the job is finished with an error
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

- There can be only one result or an error
  - The executor should call only one resolve or reject. The promise state change is final
- Technically we can call reject (just like resolve) with any type of argument
  - It's recommended to use Error objects in reject

# Consumers: ".then" and ".catch"

- A promise object serves as a link between the producing code (executor) and the consuming functions – those that want to receive the result/error
- Consuming functions can be registered using **promise.then()** and **promise.catch()**
- The syntax of **.then** is:

```
promise.then(
    function (result) { /* handle a successful result */ },
    function (error) { /* handle an error */ }
);
```

- Example:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
    result => alert(result), // shows "done!" after 1 second
    error => alert(error) // doesn't run
);
```

# Consumers: “.then” and “.catch”

- If we’re interested only in successful completions, then we can provide only one argument to .then:

```
let promise = new Promise(resolve => {
    setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

- If we’re interested only in errors, then we can use .then(null, function) or an “alias” to it: **.catch(function)**:

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

# Example: loadScript

- For instance, take a look at the function loadScript(src):

```
function loadScript(src) {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
}
```

  - The purpose of this function is to load a new script
  - When <script src="…"> is added to the document, the browser loads the script asynchronously and executes it
- We'd like to know when the script has finished loading, as to use new functions and variables from that script
- For that purpose, we can add a callback function as a second argument to loadScript() that should execute when the script loads:

# Example: loadScript

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);

    document.head.append(script);
}
```

- Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function () {
    // the callback runs after the script is loaded
    newFunction(); // so now it works
    ...
});
```

# Example: loadScript

- Let's rewrite it using promises
  - The new function loadScript() will not require a callback
  - Instead it will create and return a promise object that settles when the loading is complete

```
function loadScript(src) {
    return new Promise(function (resolve, reject) {
        let script = document.createElement('script');
        script.src = src;
        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error("Script load error: " + src));
        document.head.append(script);
    });
}

// Usage:
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js");
promise.then(
    script => alert(`${script.src} is loaded!`),
    error => alert(`Error: ${error.message}`)
);
promise.then(script => alert('One more handler to do something else!'));
```

# Callbacks vs. Promises

- We can immediately see few benefits of promises over the callback-based syntax:

| Callbacks | Promises |
| --- | --- |
| We must have a ready callback function when calling loadScript. In other words, we must know what to do with the result *before* loadScript is called. | Promises allow us to code things in the natural order.<br>First we run loadScript, and .then write what to do with the result. |
| There can be only one callback. | We can call .then on a promise as many times as we want, at any time later. |

# Exercise (32)

- The built-in function setTimeout() uses callbacks
- Create a promise-based alternative
- The function **delay(ms)** should return a promise
- That promise should resolve after ms milliseconds, so that we can add .then to it:

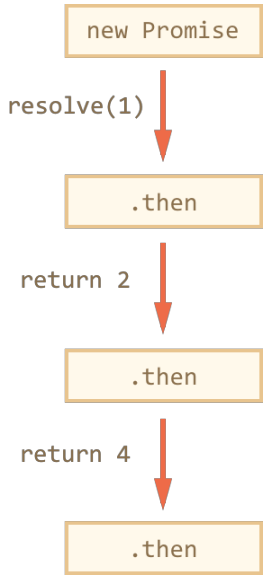
```
function delay(ms) {
    // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

# Promises Chaining

- **Promises chaining** allows you to have a sequence of asynchronous tasks to be done one after another (e.g., for loading scripts)

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
}).then(function (result) {
    alert(result); // 1
    return result * 2;
}).then(function (result) {
    alert(result); // 2
    return result * 2;
}).then(function (result) {
    alert(result); // 4
    return result * 2;
});
```




- The idea is that the result is passed through the chain of .then handlers
- This works because a call to promise.then() returns a promise, so that we can call the next .then() on it

# Returning Promises

- Normally, a value returned by a .then handler is immediately passed to the next handler
- However, if the returned value is a promise, then the further execution is suspended until it settles. After that, the result of that promise is given to the next .then handler

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
}).then(function (result) {
    alert(result); // 1

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function (result) {
    alert(result); // 2

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function (result) {
    alert(result); // 4
});
```

The output is again 1 → 2 → 4, but now with 1 second delay between alert calls

# Example: loadScript

- Let's use this feature with loadScript to load scripts one by one, in sequence:

```
loadScript("/promises/one.js")
    .then(function (script) {
        return loadScript("/promises/two.js");
    })
    .then(function (script) {
        return loadScript("/promises/three.js");
    })
    .then(function (script) {
        one();
        two();
        three();
    });
```

- Here each loadScript call returns a promise, and the next .then runs when it resolves
- Then it initiates the loading of the next script. So scripts are loaded one after another
- Note that the code is still "flat", it grows down, not to the right (as opposed to using callbacks)

# async/await

- The newest way to write asynchronous code in JavaScript (from ES7)
- Async/await simplify the process of working with promises
- **Async functions** always return a Promise
  - If the function throws an error, the Promise will be rejected
  - If the function returns a value, the Promise will be resolved

```
async function func() {
    return 1; // the same as: return Promise.resolve(1);
}

func().then(alert); // 1
```

  - The word “async” before a function means that the function always returns a promise
  - If the code has return <non-promise> in it, JS automatically wraps it into a resolved promise

# await

- The keyword await makes JS wait until the promise settles, and returns its result
  - That doesn't cost any CPU resources, because the engine can do other jobs meanwhile
  - It's just a more elegant syntax of getting the promise result than promise.then
- await can be used only inside async functions

```
async function f() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("done!"), 1000)
    });
    let result = await promise; // wait till the promise resolves

    alert(result); // "done!"
}

f();
```

# Async Methods

- A class method can also be async, just put async before it:

```
class Waiter {
    async wait() {
        return await Promise.resolve(1);
    }
}

new Waiter()
    .wait()
    .then(alert); // 1
```

- The meaning is the same: it ensures that the returned value is a promise and enables await

# Error Handling

- In case that the promise was rejected, await throws the error, just if there were a throw statement at that line

```
async function f() {
    await Promise.reject(new Error("Whoops!"));
    // the same as:
    // throw new Error("Whoops!");
}
```

- We can catch that error using try..catch, the same way as a regular throw:

```
async function f() {
    try {
        let response = await fetch('http://no-such-url');
    } catch (err) {
        alert(err); // TypeError: failed to fetch
    }
}
f();
```

# Error Handling

- If we don't have try..catch, then the promise generated by the call of the async function f() becomes rejected
- We can append .catch to handle it:

```
async function f() {
    let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch
```

- If we forget to add .catch there, then we get an unhandled promise error (and can see it in the console)
- Note that at the top level of the code, when we're outside of any async function, we're syntactically unable to use await, so it's a normal practice to add .then/catch to handle the final result or falling-through errors

# Exercise (33)

- Rewrite the following code using async/await instead of .then:

```
function resolveAfter2Seconds() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('resolved');
        }, 2000);
    });
}

function asyncCall() {
    console.log('calling');
    resolveAfter2Seconds().then(result => {
        console.log(result);
        // expected output: "resolved"
    });
}

asyncCall();
```