**Лабораторная работа №17**

**Тема работы:** Разработка программ по реализации обработки исключительных ситуаций посредством операторов try catch.

**Цель работы:** закрепить навыки по обработке исключительных ситуаций.

**Теоретические сведения**

## Обработка ошибок, "try..catch"

Неважно, насколько мы хороши в программировании, иногда наши скрипты содержат ошибки. Они могут возникать из-за наших промахов, неожиданного ввода пользователя, неправильного ответа сервера и по тысяче других причин.

Обычно скрипт в случае ошибки «падает» (сразу же останавливается), с выводом ошибки в консоль.

Но есть синтаксическая конструкция try..catch, которая позволяет «ловить» ошибки и вместо падения делать что-то более осмысленное.

## Синтаксис «try…catch»

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try {

// код...

} catch (err) {

// обработка ошибки

}

Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

Таким образом, при ошибке в блоке try {…} скрипт не «падает», и мы получаем возможность обработать ошибку внутри catch.

Давайте рассмотрим примеры.

* Пример без ошибок: выведет alert (1) и (2):
* try {
* alert('Начало блока try'); // (1) <--
* // ...код без ошибок
* alert('Конец блока try'); // (2) <--
* } catch(err) {
* alert('Catch игнорируется, так как нет ошибок'); // (3)

}

* Пример с ошибками: выведет (1) и (3):
* try {
* alert('Начало блока try'); // (1) <--
* lalala; // ошибка, переменная не определена!
* alert('Конец блока try (никогда не выполнится)'); // (2)
* } catch(err) {
* alert(`Возникла ошибка!`); // (3) <--

}

**try..catch работает только для ошибок, возникающих во время выполнения кода**

Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код.

Он не сработает, если код синтаксически неверен, например, содержит несовпадающее количество фигурных скобок:

try {

{{{{{{{{{{{{

} catch(e) {

alert("Движок не может понять этот код, он некорректен");

}

JavaScript-движок сначала читает код, а затем исполняет его. Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код.

Таким образом, try..catch может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».

**try..catch работает синхронно**

Исключение, которое произойдёт в коде, запланированном «на будущее», например в setTimeout, try..catch не поймает:

try {

setTimeout(function() {

noSuchVariable; // скрипт упадёт тут

}, 1000);

} catch (e) {

alert( "не сработает" );

}

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию try..catch.

Чтобы поймать исключение внутри запланированной функции, try..catch должен находиться внутри самой этой функции:

setTimeout(function() {

try {

noSuchVariable; // try..catch обрабатывает ошибку!

} catch {

alert( "ошибка поймана!" );

}

}, 1000);

## Объект ошибки

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок catch:

try {

// ...

} catch(err) { // <-- объект ошибки, можно использовать другое название вместо err

// ...

}

Для всех встроенных ошибок этот объект имеет два основных свойства:

**name**

Имя ошибки. Например, для неопределённой переменной это "ReferenceError".

**message**

Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

**stack**

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Например:

try {

lalala; // ошибка, переменная не определена!

} catch(err) {

alert(err.name); // ReferenceError

alert(err.message); // lalala is not defined

alert(err.stack); // ReferenceError: lalala is not defined at (...стек вызовов)

// Можем также просто вывести ошибку целиком

// Ошибка приводится к строке вида "name: message"

alert(err); // ReferenceError: lalala is not defined

}

## Блок «catch» без переменной

**Новая возможность**

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Если нам не нужны детали ошибки, в catch можно её пропустить:

try {

// ...

} catch { // <-- без (err)

// ...

}

## Использование «try…catch»

Давайте рассмотрим реальные случаи использования try..catch.

Как мы уже знаем, JavaScript поддерживает метод [JSON.parse(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse) для чтения JSON.

Обычно он используется для декодирования данных, полученных по сети, от сервера или из другого источника.

Мы получаем их и вызываем JSON.parse вот так:

let json = '{"name":"John", "age": 30}'; // данные с сервера

let user = JSON.parse(json); // преобразовали текстовое представление в JS-объект

// теперь user - объект со свойствами из строки

alert( user.name ); // John

alert( user.age ); // 30

Вы можете найти более детальную информацию о JSON в главе [Формат JSON, метод toJSON](https://learn.javascript.ru/json).

**Если json некорректен, JSON.parse генерирует ошибку, то есть скрипт «падает».**

Устроит ли нас такое поведение? Конечно нет!

Получается, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает. А люди очень не любят, когда что-то «просто падает» без всякого сообщения об ошибке.

Давайте используем try..catch для обработки ошибки:

let json = "{ некорректный JSON }";

try {

let user = JSON.parse(json); // <-- тут возникает ошибка...

alert( user.name ); // не сработает

} catch (e) {

// ...выполнение прыгает сюда

alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз." );

alert( e.name );

alert( e.message );

}

Здесь мы используем блок catch только для вывода сообщения, но мы также можем сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования, … Всё лучше, чем просто «падение».

## Генерация собственных ошибок

Что если json синтаксически корректен, но не содержит необходимого свойства name?

Например, так:

let json = '{ "age": 30 }'; // данные неполны

try {

let user = JSON.parse(json); // <-- выполнится без ошибок

alert( user.name ); // нет свойства name!

} catch (e) {

alert( "не выполнится" );

}

Здесь JSON.parse выполнится без ошибок, но на самом деле отсутствие свойства name для нас ошибка.

Для того, чтобы унифицировать обработку ошибок, мы воспользуемся оператором throw.

### Оператор «throw»

Оператор throw генерирует ошибку.

Синтаксис:

throw <объект ошибки>

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки.

Их синтаксис:

let error = new Error(message);

// или

let error = new SyntaxError(message);

let error = new ReferenceError(message);

// ...

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство name – это в точности имя конструктора. А свойство message берётся из аргумента.

Например:

let error = new Error(" Ого, ошибка! o\_O");

alert(error.name); // Error

alert(error.message); // Ого, ошибка! o\_O

Давайте посмотрим, какую ошибку генерирует JSON.parse:

try {

JSON.parse("{ bad json o\_O }");

} catch(e) {

alert(e.name); // SyntaxError

alert(e.message); // Unexpected token b in JSON at position 2

}

Как мы видим, это SyntaxError.

В нашем случае отсутствие свойства name – это ошибка, ведь пользователи должны иметь имена.

Сгенерируем её:

let json = '{ "age": 30 }'; // данные неполны

try {

let user = JSON.parse(json); // <-- выполнится без ошибок

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени"); // (\*)

}

alert( user.name );

} catch(e) {

alert( "JSON Error: " + e.message ); // JSON Error: Данные неполны: нет имени

}

В строке (\*) оператор throw генерирует ошибку SyntaxError с сообщением message. Точно такого же вида, как генерирует сам JavaScript. Выполнение блока try немедленно останавливается, и поток управления прыгает в catch.

Теперь блок catch становится единственным местом для обработки всех ошибок: и для JSON.parse и для других случаев.

## Проброс исключения

В примере выше мы использовали try..catch для обработки некорректных данных. А что, если в блоке try {...} возникнет другая неожиданная ошибка? Например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными.

Пример:

let json = '{ "age": 30 }'; // данные неполны

try {

user = JSON.parse(json); // <-- забыл добавить "let" перед user

// ...

} catch(err) {

alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined

// (не JSON ошибка на самом деле)

}

Конечно, возможно все! Программисты совершают ошибки. Даже в утилитах с открытым исходным кодом, используемых миллионами людей на протяжении десятилетий – вдруг может быть обнаружена ошибка, которая приводит к ужасным взломам.

В нашем случае try..catch предназначен для выявления ошибок, связанных с некорректными данными. Но по своей природе catch получает все свои ошибки из try. Здесь он получает неожиданную ошибку, но всё также показывает то же самое сообщение "JSON Error". Это неправильно и затрудняет отладку кода.

К счастью, мы можем выяснить, какую ошибку мы получили, например, по её свойству name:

try {

user = { /\*...\*/ };

} catch(e) {

alert(e.name); // "ReferenceError" из-за неопределённой переменной

}

Есть простое правило:

**Блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.**

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.
2. В блоке catch(err) {...} мы анализируем объект ошибки err.
3. Если мы не знаем как её обработать, тогда делаем throw err.

В коде ниже мы используем проброс исключения, catch обрабатывает только SyntaxError:

let json = '{ "age": 30 }'; // данные неполны

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени");

}

blabla(); // неожиданная ошибка

alert( user.name );

} catch(e) {

if (e.name == "SyntaxError") {

alert( "JSON Error: " + e.message );

} else {

throw e; // проброс (\*)

}

}

Ошибка в строке (\*) из блока catch «выпадает наружу» и может быть поймана другой внешней конструкцией try..catch (если есть), или «убьёт» скрипт.

Таким образом, блок catch фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные.

Пример ниже демонстрирует, как такие ошибки могут быть пойманы с помощью ещё одного уровня try..catch:

function readData() {

let json = '{ "age": 30 }';

try {

// ...

blabla(); // ошибка!

} catch (e) {

// ...

if (e.name != 'SyntaxError') {

throw e; // проброс исключения (не знаю как это обработать)

}

}

}

try {

readData();

} catch (e) {

alert( "Внешний catch поймал: " + e ); // поймал!

}

Здесь readData знает только, как обработать SyntaxError, тогда как внешний блок try..catch знает, как обработать всё.

## try…catch…finally

Подождите, это ещё не всё.

Конструкция try..catch может содержать ещё одну секцию: finally.

Если секция есть, то она выполняется в любом случае:

* после try, если не было ошибок,
* после catch, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

try {

... пробуем выполнить код...

} catch(e) {

... обрабатываем ошибки ...

} finally {

... выполняем всегда ...

}

Попробуйте запустить такой код:

try {

alert( 'try' );

if (confirm('Сгенерировать ошибку?')) BAD\_CODE();

} catch (e) {

alert( 'catch' );

} finally {

alert( 'finally' );

}

У кода есть два пути выполнения:

1. Если вы ответите на вопрос «Сгенерировать ошибку?» утвердительно, то try -> catch -> finally.
2. Если ответите отрицательно, то try -> finally.

Секцию finally часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

Например, мы хотим измерить время, которое занимает функция чисел Фибоначчи fib(n). Естественно, мы можем начать измерения до того, как функция начнёт выполняться и закончить после. Но что делать, если при вызове функции возникла ошибка? В частности, реализация fib(n) в коде ниже возвращает ошибку для отрицательных и для нецелых чисел.

Секция finally отлично подходит для завершения измерений несмотря ни на что.

Здесь finally гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения fib и в случае ошибки:

let num = +prompt("Введите положительное целое число?", 35)

let diff, result;

function fib(n) {

if (n < 0 || Math.trunc(n) != n) {

throw new Error("Должно быть целое неотрицательное число");

}

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 || "возникла ошибка");

alert( `Выполнение заняло ${diff}ms` );

Вы можете это проверить, запустив этот код и введя 35 в prompt – код завершится нормально, finally выполнится после try. А затем введите -1 – незамедлительно произойдёт ошибка, выполнение займёт 0ms. Оба измерения выполняются корректно.

Другими словами, неважно как завершилась функция: через return или throw. Секция finally срабатывает в обоих случаях.

**Переменные внутри try..catch..finally локальны**

Обратите внимание, что переменные result и diff в коде выше объявлены до try..catch.

Если переменную объявить в блоке, например, в try, то она не будет доступна после него.

**finally и return**

Блок finally срабатывает при любом выходе из try..catch, в том числе и return.

В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

function func() {

try {

return 1;

} catch (e) {

/\* ... \*/

} finally {

alert( 'finally' );

}

}

alert( func() ); // сначала срабатывает alert из finally, а затем этот код

**try..finally**

Конструкция try..finally без секции catch также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.

function func() {

// начать делать что-то, что требует завершения (например, измерения)

try {

// ...

} finally {

// завершить это, даже если все упадёт

}

}

В приведённом выше коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отрабатывает до того, как поток управления выйдет из функции.

## Глобальный catch

**Зависит от окружения**

Информация из данной секции не является частью языка JavaScript.

Давайте представим, что произошла фатальная ошибка (программная или что-то ещё ужасное) снаружи try..catch, и скрипт упал.

Существует ли способ отреагировать на такие ситуации? Мы можем захотеть залогировать ошибку, показать что-то пользователю (обычно они не видят сообщение об ошибке) и т.д.

Такого способа нет в спецификации, но обычно окружения предоставляют его, потому что это весьма полезно. Например, в Node.js для этого есть process.on("uncaughtException"). А в браузере мы можем присвоить функцию специальному свойству window.onerror, которая будет вызвана в случае необработанной ошибки.

Синтаксис:

window.onerror = function(message, url, line, col, error) {

// ...

};

**message**

Сообщение об ошибке.

**url**

URL скрипта, в котором произошла ошибка.

**line, col**

Номера строки и столбца, в которых произошла ошибка.

**error**

Объект ошибки.

Пример:

<script>

window.onerror = function(message, url, line, col, error) {

alert(`${message}\n В ${line}:${col} на ${url}`);

};

function readData() {

badFunc(); // Ой, что-то пошло не так!

}

readData();

</script>

Роль глобального обработчика window.onerror обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как [https://errorception.com](https://errorception.com/) или [http://www.muscula.com](http://www.muscula.com/).

Они работают так:

1. Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
2. Этот JS-скрипт ставит свою функцию window.onerror.
3. Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
4. Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

**Порядок выполнения работы**

### Finally или просто код?

Сравните два фрагмента кода.

1. Первый использует finally для выполнения кода после try..catch:
2. try {
3. начать работу
4. работать
5. } catch (e) {
6. обработать ошибку
7. } finally {
8. очистить рабочее пространство

}

1. Второй фрагмент просто ставит очистку после try..catch:
2. try {
3. начать работу
4. работать
5. } catch (e) {
6. обработать ошибку
7. }

очистить рабочее пространство

Нам определённо нужна очистка после работы, неважно возникли ошибки или нет.

Есть ли здесь преимущество в использовании finally или оба фрагмента кода одинаковы? Если такое преимущество есть, то дайте пример, когда оно проявляется.

### Eval-калькулятор с ошибками

важность: 5

Напишите интерфейс, который принимает математическое выражение (в prompt) и выводит результат его вычисления через eval.

**При ошибке нужно выводить сообщение и просить переввести выражение**.

Ошибкой считается не только некорректное выражение, такое как 2+, но и выражение, возвращающее NaN, например 0/0.