Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ |

|  |  |
| --- | --- |
| Специальность:  2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Скриптовые языки программирования» |

**Лабораторная работа № 22**

**Инструкционно-технологическая карта**

Тема: Обработка ошибок. Оператор try-catch синтаксис и семантика

Цель: Научиться обрабатывать ошибки при помощи оператора try-catch

Время выполнения: 2 часа

# Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.
2. Реализовать алгоритм решения задачи.
3. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.
4. Ответить на контрольные вопросы.

# Теоретические сведения

Ошибки могут возникать из-за неожиданного ввода пользователя, неправильного ответа сервера и по другим многочисленным причинам.

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 {

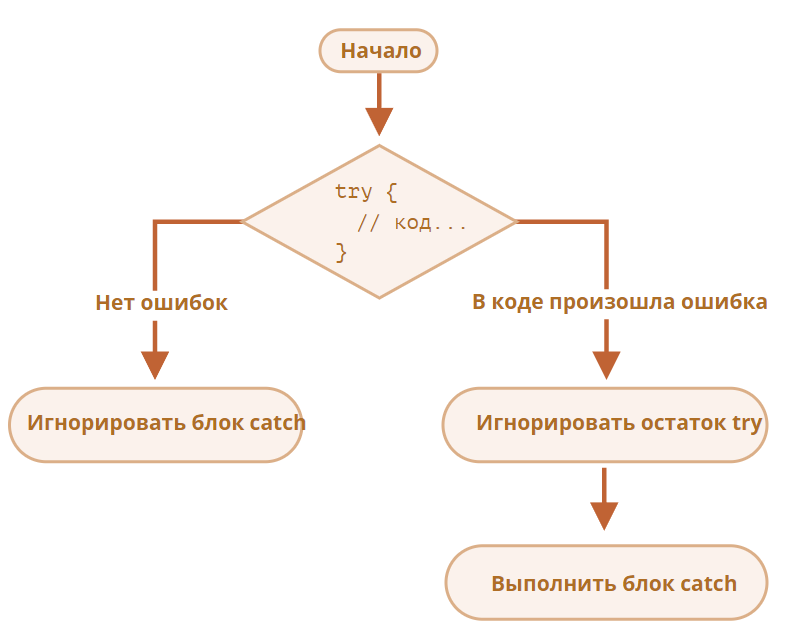
    // код...

} catch (err) {

    // обработка ошибки

}

Она работает следующим образом:

1. Первым 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 появляется возможность обработать ошибку внутри **catch**.

К примеру:

Данный 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. То есть, 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», или «**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)

    // Можно также просто вывести ошибку целиком

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

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

}

#### Блок «catch» без переменной

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

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

try {

    // ...

} catch { //  <-- без (err)

    // ...

}

## Использование «try…catch»

Нужно рассмотреть реальные случаи использования **try…catch**.

Уже говорилось, что JavaScript поддерживает метод **JSON.parse(str)** для чтения JSON.

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

Получение данных и вызов **JSON.parse** осуществляется следующим образом:

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

let user = JSON.parse(json); // преобразование текстового представления в JS-объект

// теперь user - объект со свойствами из строки

alert(user.name); // John

alert(user.age);  // 30

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

Данный исход не подходит для решения!

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.

Стоит использовать **try…catch** для обработки ошибки:

let json = "{ некорректный JSON }";

try {

    let user = JSON.parse(json); // <--на данном этапе возникает ошибка...

    alert(user.name); // не сработает

} catch (e) {

    // ...выполнение переходит сюда

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

    alert(e.name);

    alert(e.message);

}

В данном примере используется блок **catch** только для вывода сообщения, но также можно сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования, и так далее.

## Генерация собственных исключений

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

Например:

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, используется оператор **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 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); // оповещение об ошибке

Для рассмотрения ошибки, которую генерирует **JSON.parse** представлен пример:

try {

    JSON.parse("{ bad json o\_O }");

} catch (e) {

    alert(e.name); // SyntaxError

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

}

Это и есть **SyntaxError**.

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

Нужно сгенерировать 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 использовался **try…catch** для обработки некорректных данных. 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); // <-- не был добавлен "let" перед user

    // ...

} catch (err) {

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

    // (не JSON ошибка на самом деле)

}

В данном 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.

Однако можно выяснить, какая ошибка вернулась, например, по её свойству **name**:

try {

    user = { /\*...\*/ };

} catch (e) {

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

}

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) {...}** анализируется объект ошибки **err**.
3. Если способ обработки неизвестен, то можно сделать **throw err**.

В коде ниже используется 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 знание как это обработать)

        }

    }

}

try {

    readData();

} catch (e) {

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

}

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

#### try…catch…finally

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 {

    ... попытка выполнения кода...

} catch (e) {

    ... обработка ошибки ...

} finally {

    ... выполняется всегда ...

}

Для примера нужно запустить следующий 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 ответ на вопрос: «Сгенерировать ошибку?» утвердительный, то **try → catch → finally**.
2. Если ответ отрицательный, то **try → finally**.

Секция **finally** часто используется, когда начинается некое действие и требуется завершить это вне зависимости от того, будет ошибка или нет.

Например, требуется измерить время, которое занимает функция чисел Фибоначчи **fib(n)**. Естественно, можно 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`);

Это можно проверить, запустив этот код и введя 35 в **prompt** – код завершится нормально, **finally** выполнится после **try**. А затем ввести -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

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. Она применяется, когда здесь обрабатывать ошибки не нужно, но нужно, чтобы 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.

# Контрольные вопросы

1. Привести пример использования конструкции **try…catch**.
2. Какие свойства содержат объекты ошибок?
3. Как можно пропустить объект ошибки, если он не нужен?
4. Что такое проброс исключения?

# Задания

1. Сравнить два фрагмента кода.

Первый использует **finally** для выполнения кода после **try…catch**:

try {

    начать работу

    работать

} catch (e) {

    обработать ошибку

} finally {

    очистить рабочее пространство

}

Второй фрагмент просто ставит очистку после try..catch:

try {

    начать работу

    работать

} catch (e) {

    обработать ошибку

}

очистить рабочее пространство

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 дать пример, когда оно проявляется.

1. Написать функцию, которая будет извлекать корень из числа и при этом выбрасывать исключение, если корень извлекается из отрицательного числа.
2. Пусть JSON выдается следующим образом:

let json = `[

    {

        "name": "user1",

        "age": 25,

        "salary": 1000

    },

    {

        "name": "user2",

        "age": 26,

        "salary": 2000

    },

    {

        "name": "user3",

        "age": 27,

        "salary": 3000

    }

]`;

Проверить этот JSON на общую корректность при разборе, а после разбора проверить, что в результате получается массив, а не что-то другое. Если в результате получается не массив ‑ выбросить исключение.

1. Пользователь вводит три числа. Если одно из чисел меньше ноля, вернуть и обработать первое исключение, если одно из чисел равно нолю, вернуть и обработать второе исключение, если пользователь не ввёл одно или несколько чисел, вернуть и обработать третье исключение. По завершении работы в любом случае вывести сообщение о завершении работы алгоритма.

# Литература

**Диков, А.В.** Клиентские технологии веб программирования: JavaScript и DOM: учебное пособие / А.В. Диков. – СПб: Лань, 2020 – 124 с.

**Читанамбри, Кирупа.** Изучаем React / Кирупа Читанамбри – 2-е изд. – М.: Эксмо, 2019. 368 с.

**Хавербеке, Марейн.** Выразительный JavaScript. Современное веб-программирование. / Марейн Хавербеке – 3-е изд. – СПб.: Питер, 2019. – 480 с.:

Преподаватель Рогалевич А.В.

Рассмотрено на заседании цикловой

комиссии программного обеспечения

информационных технологий №10

Протокол № \_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Председатель ЦК\_\_\_\_\_\_\_\_ В.Ю.Михалевич