

Обработка исключений и замыкания



Владимир Чебукин



Владимир Чебукин

Frontend-разработчик



[Владимир Чебукин](#)



План занятия

1. [Вспомним то, что мы уже знаем](#)
2. [Создание собственных исключений](#)
3. [Перехват исключений](#)
4. [Консольные методы для удобной работы](#)
5. [Области видимости](#)
6. [Замыкания](#)
7. [Итоги](#)

Вспомним прошлый материал

Что будет выведено?

```
let value = 1;
function counterGenerator(){
  let value = 3;
  function showCounter(){
    console.log(value);
  }
  function increaseCounter(){
    value += 1;
  }
  return [showCounter, increaseCounter];
}
const [show, increase] = counterGenerator();
increase();
show();
```

Внутренние исключения выполнения

Откуда появляются **ошибки**?

Некоторые стандартные ситуации не должны происходить:

```
let user = null;  
let age = user.age;  
//Uncaught TypeError: Cannot read property 'age' of null?
```

Если интерпретатор не может выполнить команду, он выбрасывает **ошибку**



Создание собственных исключений

Свои исключения

А можем ли мы сами **выбрасывать исключения**? Если да, то как и когда?

```
const divider = (a, b) => a / b;  
console.log(divider(1, 0)); // Infinity;
```

Вопрос: как запретить деление на 0?

Свои исключения

Иногда возникают исключительные ситуации, которые с точки зрения интерпретатора **ошибкой не считаются**, но ошибка проявится **позже** — недополучены какие-то параметры с сервера.

Или может произойти ошибка со стороны **бизнес-логики** — недопустимое значение исходных данных



Свои исключения

С точки зрения **интерпретатора** такие случаи ошибками не считаются, но с точки зрения разработчика — являются.

В таких случаях требуется генерировать свои ошибки:

```
const divider = (a, b) => {  
  if (b == 0) throw "Ошибка деления на 0";  
  
  return a / b;  
};  
console.log(divider(1, 2)); // 0.5;  
console.log(divider(1, 0)); // "Ошибка деления на 0";
```

Объект ошибки

Примитивы вроде текста или чисел не используются как ошибки. Для ошибок существует класс `Error`:

```
const divider = (a, b) => {  
  if (b !== 0) {  
    const divideError = new Error("Ошибка деления на 0");  
    throw divideError;  
    // или throw new Error("Ошибка деления на 0");  
  }  
  
  return a / b;  
};  
console.log(divider(1, 2)); // 0.5;  
console.log(divider(1, 0)); // "Ошибка деления на 0";
```



Перехват исключений

Зачем перехватывать исключения?

Если вы скажете, что ошибки надо решать, а не прятать, я полностью с вами соглашусь. Однако всегда ли наличие ошибки зависит от вас?

Может ли быть ситуация, при которой **ошибка произойдет**, несмотря на то, что написанный вами код полностью корректен?



Зачем перехватывать исключения?

Например:

- **нельзя гарантировать**, что необходимые данные получены с сервера
- **нельзя гарантировать**, что сторонний сервис всегда доступен и корректно работает
- **нельзя гарантировать**, что автор библиотеки *или тот, кто использует вашу библиотеку*, так же добросовестно пишет код, как и вы
- **нельзя гарантировать**, что пользователь не сможет ввести некорректные данные, *хоть и надо к этому стремиться*

Зачем перехватывать исключения?

- При возникновении ошибки необходимо **культурно сообщить об этом пользователю**

В случае, если пользователь встретится с ошибкой скрипта, лучше ему сообщить об этом ошибкой «Что-то пошло не так», зафиксировать ошибку и дать возможность продолжить работу, а не оставить его один на один с непонятным поведением страницы

Зачем перехватывать исключения?

- При возникновении ошибки необходимо культурно сообщить об этом пользователю
- При возникновении ошибки необходимо получить расширенную информацию

При выполнении скрипта возникла ошибка. Не всегда очевидно, какие значения данных к этому приводят. В этом случае в блоке catch достаточно будет дописать вывод необходимых нам данных, приводящих к ошибке

Зачем перехватывать исключения?

- При возникновении ошибки необходимо **культурно сообщить об этом пользователю**
- При возникновении ошибки необходимо **получить расширенную информацию**
- Случившаяся ошибка **не должна прерывать выполнение** дальнейшего кода

Подключение виджета для отображение погоды на сайте может прекратить выполнение дальнейшего кода. Лучше заранее такое предусмотреть

Как перехватывать исключения?

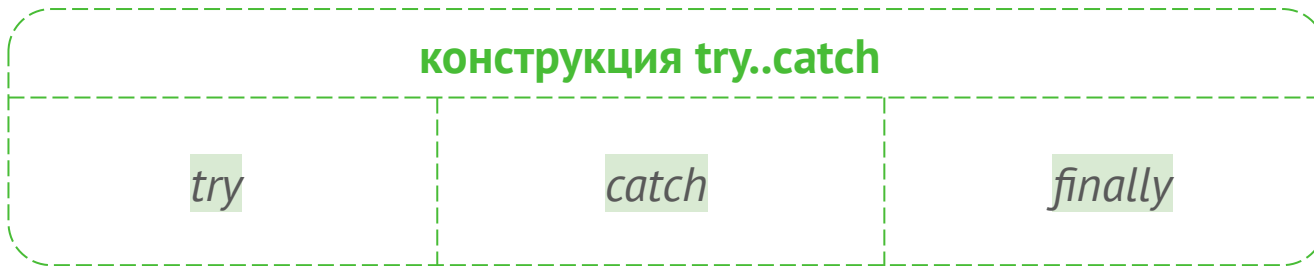
Конструкция `try..catch` служит для того, чтобы браузер попытался интерпретировать код.

Однако если **выполнить код не удастся**, то можно поймать ошибку или промежуточные данные, обработать её и затем безопасно выполнять код дальше



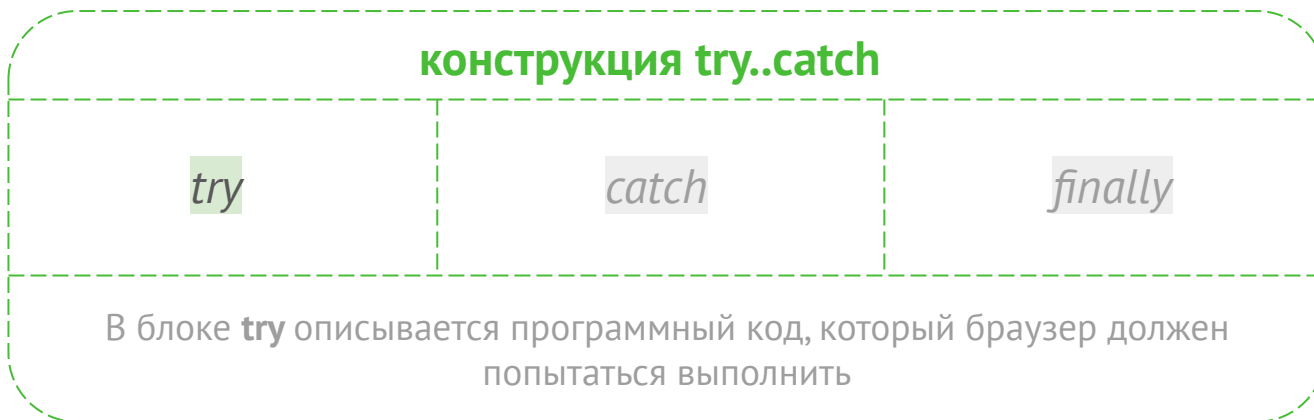
Как перехватывать исключения?

Конструкция `try..catch` состоит из блоков:



Описание блоков конструкции `try..catch`

Конструкция `try..catch` состоит из блоков:



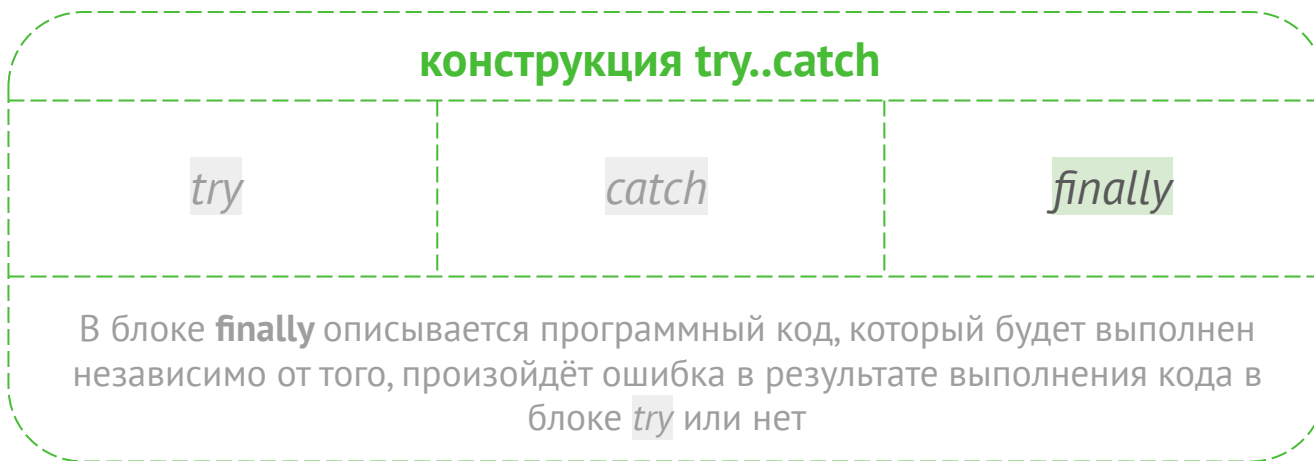
Описание блоков конструкции `try..catch`

Конструкция `try..catch` состоит из блоков:



Описание блоков конструкции `try..catch`

Конструкция `try..catch` состоит из блоков:



Перехват исключений в действии: демо

- `try..catch` в действии:

```
try {  
    // .. код, который может выполняться неверно  
} catch (error) {  
    // .. код, который в этом случае выполнится  
}
```

Перехват исключений в действии: демо

- `try..catch` в действии
- `try..finally` в действии:

```
try {  
    // .. код, который может выполняться неверно  
} finally {  
    // .. код, который выполнится в любом случае  
}
```

Перехват исключений в действии: демо

- `try..catch` в действии
- `try..finally` в действии
- `try..catch..finally` в действии:

```
try {  
    // .. код, который может выполняться неверно  
} catch (error) {  
    // .. код, который в этом случае выполнится  
} finally {  
    // .. код, который выполнится в любом случае  
}
```


Ограничения для `try...catch`

Перехват ошибки **не сработает**:

- если есть **синтаксическая ошибка**:

```
try{  
    console.log(Ошибка не произошла!);  
}catch(error){  
    console.log('Ошибка произошла!');  
}  
// Uncaught SyntaxError: missing ) after argument list
```

В этом случае `try...catch` **не будет выполняться**, интерпретатор сообщит о синтаксической ошибке

Ограничения для `try...catch`

Перехват ошибки **не работает**:

- если код, в котором произошла ошибка, работает **асинхронно** по отношению к `try...catch`:

```
try {
  setTimeout(() => {
    console.log(null.unknown_property);
  }, 200)
}catch(error){
  console.log('Ошибка произошла!');
}
// Uncaught ReferenceError: Invalid left-hand side in assignment
```

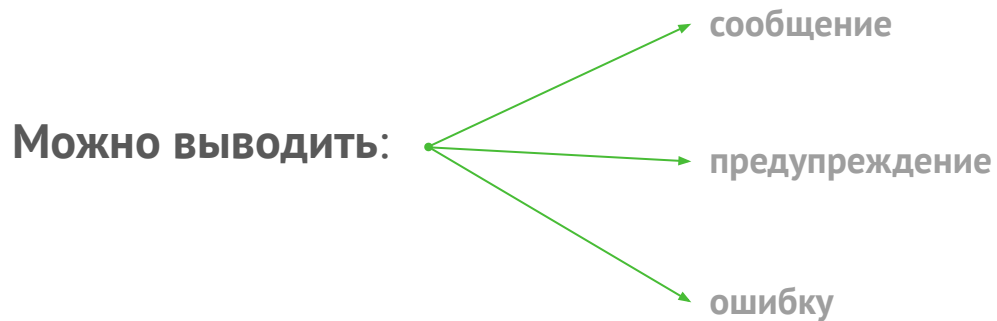
Асинхронность будет изучаться в одной из следующих лекций



Консольные методы для удобной работы

Вывод на консоль

Всем известен вывод на консоль с помощью `console.log`, используемый для вывода. При этом можно выводить не только сообщение.



Очистка консоли

После многих сообщений в выводе можно запутаться. Для очистки всего вывода можно использовать `console.clear()`



Форматированный вывод

Вы можете напечатать очень хорошую таблицу с объектами, которые выводите, используя `console.table()`



Замер времени выполнения кода

Быстро ли выполняется ваша программа? Это сложно понять, если не производить **замер времени выполнения** отдельных функций.

Для замера времени выполнения используются методы `console.time()` и `console.timeEnd()`. *В метод передается id таймера.*

Таким образом можно получить время выполнения цикла из 10 000 итераций



Области видимости

Глобальная область видимости

Сделаем программу, которая выводит приветственное сообщение пользователю. Реализуем полноценный пример. Для этого нам понадобится **HTML** и **JavaScript-код**:

```
let firstName = "Олег";  
function showGreeting(person){  
    console.log(`С днём рождения, ${person}!`);  
}  
  
showGreeting(firstName);
```

Манипуляции с консолью

Теперь откроем страницу и **включим консоль**. Введём в консоль:

```
firstName;
```

И увидим значение, которое **хранится на настоящий момент** в нашей переменной:

```
>  
С днём рождения, Олег!  
> firstName  
=> 'Олег'
```

Манипуляции с консолью

Более того, если в консоли ввести **друг за другом команды**:

```
firstName = "Иван";  
  
showGreeting(firstName);
```

То получим текст поздравления с тем именем, которое мы установили в консоли!

```
> firstName = 'Иван';  
=> 'Иван'  
> showGreeting( firstName );  
С днём рождения, Иван!
```

Пошалим?

Проделаем еще одну шалость и **попробуем изменить код функции** `showGreeting`.

Запишем в консоль:

```
showGreeting = function(person){  
  console.log(`Вас взломали, ${person}!`);  
};
```

Пошалим? Шалость удалась

Прделаем ещё одну шалость и **попробуем изменить код функции** `showGreeting`.

Запишем в консоль:

```
showGreeting = function(person){  
  console.log(`Вас взломали, ${person}!`);  
};
```

И **следом** запустим:

```
showGreeting(firstName);
```

```
✧ showGreeting = function( person ) { console.log( `Вас взломали, ${person}!` ) }  
=> [Function: showGreeting]  
✧ showGreeting( firstName );  
Вас взломали, Иван!
```

Содержимое файлов `main.js` и `index.html` при таком способе не меняется. После обновления страницы значение `name` будет снова `Олег`, а функция `showGreeting` будет снова работать, как надо

Проблемы безопасности

Очевидно, что если бы наша программа была чуть серьёзнее и хранила данные о кредитных картах или пароли пользователей, **возможность изменения** нашего кода делала бы владельца любого проекта беспомощным против хакеров.

Для решения этой проблемы нам необходимо **ограничить доступ** к переменной и функции



Глобальная область видимости

До настоящего времени мы писали код в **глобальной области видимости** — месте, в котором можно изменить любую функцию или переменную, и на них не накладывается никаких ограничений.

Все функции **имеют доступ** к переменным и функциям глобальной области видимости:

```
// эта информация доступна любой функции в программе
let secret = "Ленин - гриб!";

function showSecret(){
  // любая функция имеет доступ к глобальной области видимости
  console.log(secret); // выведет содержимое secret
}

console.log(secret); // тоже выведет содержимое "Ленин - гриб!"
showSecret();
```

Функциональная область видимости

Функциональная область видимости ограничивает доступность переменных и функций **телом функции**. Всё, что находится в функции, должно остаться только в ней.

Функция ограничивает доступ к своим переменным:

```
function keepSecret(){
  let secret = "Ленин - гриб!";
  console.log(secret); // Выведет «Ленин - гриб!»
}

// Выведет ошибку «Can't find variable: secret»
console.log(secret);
```


Границы областей видимости

Переменные и функции, созданные в **функциональной области видимости** `keepSecret`, доступны **только** в рамках созданной `keepSecret`:

```
> let shared = 'Медведев - шмель';  
  
function keepSecret() {  
  let secret = 'Ленин - гриб';  
  console.log(secret);  
}  
  
keepSecret();  
// Ошибка доступа к переменной (которой не существует в глобальной области видимости)  
console.log(secret);
```

Ленин - гриб

✖ ▶ Uncaught ReferenceError: secret is not defined
at <anonymous>:10:13

глобальная область

keepSecret

Переменные и функции, созданные в **глобальной области видимости**, доступны **везде**

Использование функций

Один из способов решения проблемы с безопасностью — использование **функциональной области видимости**.

Создадим **функцию** `init` и сразу же её вызовем:

main.js

```
function init(){
  let firstName = "Олег";
  function showGreeting(person){
    console.log(`С днём рождения, ${person}!`);
  }

  showGreeting(firstName);
}

init();

console.log(firstName);
```

Использование функций

Один из способов решения проблемы с безопасностью — использование **функциональной области видимости**.

Создадим **функцию** `init` и сразу же её вызовем:

main.js

```
function init() { ... }  
  
init();  
  
console.log(firstName);
```

Результат вызова:

```
>  
ReferenceError: Can't find variable: firstName  
global code@https://replbox.repl.it/data/web_hosting_1/neizerth/ElderlyContentDos/script.js:12:23  
С днём рождения, Олег!  
>
```

Использование функций

Результат: мы получили **сообщение** с поздравлением:

```
С днём рождения, Олег!
```

И **ошибку**, что возникла из-за **попытки обращения** к переменной `firstName`, которая находится в **функциональной области видимости** и доступна только функции `init`, но не глобальной переменной:

```
ReferenceError: Can't find variable: firstName
global code@https://replbox.repl.it/data/web_hosting_1/neizerth/ElderlyContentDos/script.js:12:23
```

Замыкания. Окружение

Предположим, что у нас есть некая нами написанная **функция** `tick`, которая должна каждый раз при вызове **выводить в консоль** число на единицу больше (своеобразный таймер):

```
function tick(){  
    // какой-то код, который мы еще напишем  
}  
  
tick(); // 1  
tick(); // 2  
tick(); // 3
```

Задача: напишите функцию `tick`

Уничтожение

Попробуем создать в `tick` переменную и сразу же увеличить ее на 1:

```
function tick(){  
  let start = 1;  
  console.log(start++);  
}
```

```
tick(); // 1  
tick(); // 1  
tick(); // 1
```

Почему это не работает? Дело в том, что `start` создается и уничтожается с **каждым новым вызовом** `tick`

Переменные в глобальной области видимости

Но мы можем использовать **глобальную область видимости**:

```
let start = 1;

function tick(){
  console.log(start++);
}

tick(); // 1
tick(); // 2
tick(); // 3
```

В таком случае **start** **уничтожается** только после того, как посетитель закроет вкладку

Переменные в глобальной области видимости

Иными словами, переменные в **глобальной области видимости** не уничтожаются из памяти.

Переменные **функциональной области видимости** доступны только на время выполнения функции





Замыкания

Что такое замыкание?

Замыкания — это **функция** вместе со всеми внешними переменными, которые ей доступны.

Функция запоминает окружение, в котором она была **создана**:

```
let start = 1;

function tick(){
  console.log(start++);
}
```

В нашем случае замыканием является функция *tick* и переменная *start*

Добавляем условие

Если нам потребуется создать функцию `tick10`, которая бы считала с 10, нам потребуется **доработать** программу:

```
let start = 1;

function tick(){
  console.log(start++);
}
tick(); // 1
tick(); // 2

let start10 = 10;
function tick10(){
  console.log(start10++);
}

tick10(); // 10
tick10(); // 11
```

Можно ли объединить этот код?



Избавляемся от переменных в глобальной области видимости

Как и говорилось ранее, `return` может возвращать любое выражение, в том числе и функцию.

Предположим, что у нас есть мифическая функция `createCounter`, которая должна работать так:

Код на следующей странице

Избавляемся от переменных в глобальной области видимости

```
function createCounter(){  
    // какой-то код  
}  
  
let tick = createCounter(1), // начинаем считать с 1  
    tick10 = createCounter(10); // начинаем считать с 10  
  
tick(); // 1  
tick(); // 2  
tick(); // 3  
  
tick10(); // 10  
tick10(); // 11  
tick10(); // 12
```

Анализируем

Раз `tick` и `tick10` — функции, значит, `createCounter` должна возвращать функцию:

```
function createCounter(){
  /*
    Именно функция ниже сработает, когда мы будем вызывать tick или tick10.
    Иными словами, именно эта функция поместится в tick и tick10.
  */
  return function (){
    // какой-то код
  };
}
```

Анализируем

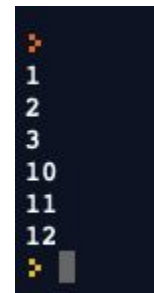
В `createCounter` мы передаем начальное значение счётчика:

```
function createCounter(start = 0){  
  return function () {  
    // какой-то код  
  };  
}
```

Как и в базовой версии `tick`, нам нужно использовать консоль и увеличение значения на один для работы функции:

```
function createCounter(start = 0){  
  return function () {  
    console.log(start++);  
  };  
}
```

Готово!



```
1  
2  
3  
10  
11  
12  
█
```

Замыкания. Пояснение

В нашем случае **замыканием** является вложенная в `createCounter` **функция**, которая использует аргумент `start`, находящийся в области видимости вне `createCounter`.

Как происходит **поиск** переменных и функций по областям видимости?

Код на следующей странице

Поиск в областях видимости

Если в текущей области видимости какая-то переменная не была найдена, она ищется в **области видимости выше** (включая глобальную):

```
let x = 8;
function findX(){
  function stillFindX(){
    function whereIsX(){
      console.log(`X = ${x}`); // X = 8
    }
    whereIsX();
  }
  stillFindX();
}
findX();
```

Остановка поиска

Поиск **останавливается** при нахождении переменной:

```
let x = 3;
console.log(`X = ${x}`); // X = 3;
function findX(){
  let x = 8;
  function stillFindX(){
    let x = 4;
    // тут поиск переменной x остановится
    function whereIsX(){
      console.log(`X = ${x}`); // X = 4
    }
    whereIsX();
  }
  stillFindX();

  console.log(`X = ${x}`); // X = 8;
}
findX();

console.log(`X = ${x}`); // X = 3;
```

Если переменная **не найдена** ни в одной области видимости, мы получим **ошибку**

Поиск в областях видимости

```
function leninHistory(){
  let leninName = "Владимир";
  console.log(`Меня зовут ${leninName}, у меня нет предшественников`);
  function stalinHistory(){
    let stalinName = "Иосиф";

    console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);

    function medvedevHistory(){
      let medvedevName = "Дмитрий";

      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников:
${stalinName}, ${leninName}`);
    }
    medvedevHistory();
  }
  // не забываем выполнить объявленную функцию
  stalinHistory();
}

leninHistory();
```

Поиск в областях видимости

Иллюстрация областей видимости:

```
> function leninHistory() {  
  let leninName = "Владимир";  
  console.log(`Меня зовут ${leninName}, у меня нет предшественников`);  
  function stalinHistory() {  
    let stalinName = "Иосиф";  
  
    console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);  
  
    function medvedevHistory() {  
      let medvedevName = "Дмитрий";  
  
      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);  
    }  
    medvedevHistory();  
  }  
  // не забываем выполнить объявленную функцию  
  stalinHistory();  
}  
  
leninHistory();
```

leninHistory

stalinHistory

medvedevHistory

global

Изменения в областях видимости

Будьте аккуратны, используя замыкания:

```
let firstName = "Олег";
function changeName(){
  firstName = "Иван";
}

console.log(firstName); // Олег
changeName();
console.log(firstName); // Иван
```

Для этого не забывайте ставить **let**:

```
let firstName = "Олег";
function changeName(){
  // обратите внимание на let
  let firstName = "Иван";
}

console.log(firstName); // Олег
changeName();
console.log(firstName); // Олег
```



Итоги



Чему мы научились?

- **Познакомились** с исключениями, узнали, как их генерировать и перехватывать
- **Изучили** новые интересные методы для работы с консолью
- **Узнали** об областях видимости
- **Познакомились** с концепцией замыканий

Домашнее задание

Давайте посмотрим ваше [домашнее задание](#):

- **вопросы** по домашней работе задаем в группе Slack;
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как приняты **все обязательные задачи**

**Задавайте вопросы и
пишите отзыв о лекции!**

Владимир Чебукин