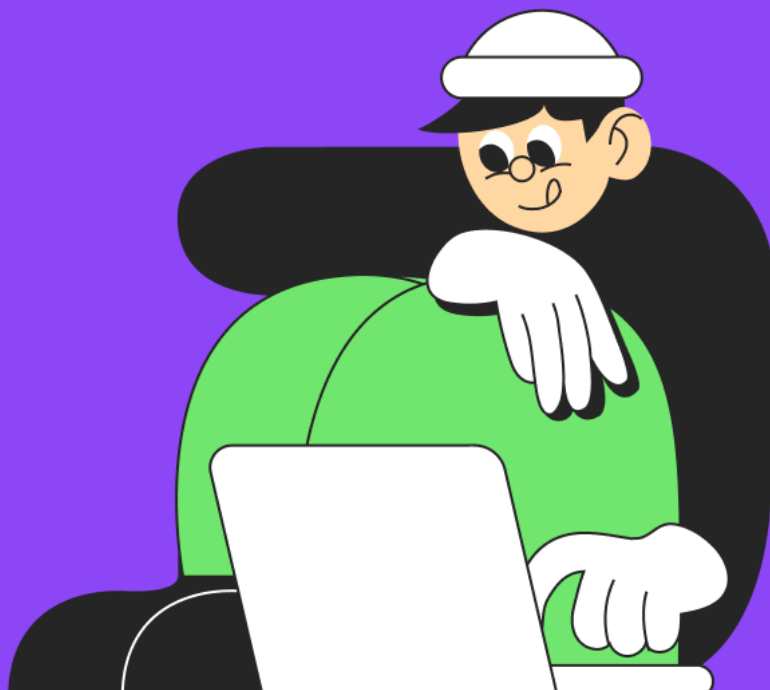
GeekBrains

Функциональный JavaScript

Курс JavaScript про ECMAScript



Оглавление

Spread, rest operator

Со стандартом ES2015 нам стали доступны очень полезные инструменты для работы с массивами - это spread и rest операторы, а также деструктуризация.

Spread operator

Spread (от английского расширить) - оператор расширения, или по другому распространения данных из массива в атомарные элементы. Т.е. мы можем взять массив, и вытащить все его элементы как отдельные переменные, это порой необходимо, когда мы хотим передать множество аргументов в функцию или хотим перенести элементы одного массива в другой, для этого необходимо перед массивом поставить многоточие (оператор spread). Давайте посмотрим на примерах:

```
const studentsGroup1PracticeTime = [
  {
    firstName: "Ivanov",
    practiceTime: 56
  },
  {
    firstName: "Petrov",
    practiceTime: 120
  },
  {
    firstName: "Sidorov",
    practiceTime: 148
  },
  {
    firstName: "Belkin",
    practiceTime: 20
  },
  {
    firstName: "Avdeev",
    practiceTime: 160
  }
];

const studentsGroup2PracticeTime = [
  {
    firstName: "Mankov",
    practiceTime: 87
```

```
  },
  {
    firstName: "Kistin",
    practiceTime: 133
  },
  {
    firstName: "Kotlyarov",
    practiceTime: 140
  },
  {
    firstName: "Peskov",
    practiceTime: 10
  },
];

// Напишем не очень удобную, но показательную функцию, которая
умеет принимать неограниченное число аргументов, и находить
максимум среди них. Функция должна вызываться подобным образом:
const maximum = findMax(4, 7, 10);
function findMax() {
  const values = arguments; // arguments - переменная доступная
внутри каждой функции, которая содержит в себе все аргументы,
переданные в функцию. Является псевдомассивом.
  let maxValue = -Infinity;
  // Так как arguments является псевдомассивом, мы не можем
применять к нему новые методы массивов такие как forEach или
reduce, поэтому будем итерировать по старинке.
  for (let index = 0; index < values.length; index++) {
    if (values[index] > maxValue) maxValue = values[index];
  }
  return maxValue;
};
// Мы должны передавать в нашу функции только числа, а в наших
массивах содержаться объекты, поэтому сначала создадим массивы
только со значениями времени отработанными студентами.
const group1PracticeTime =
studentsGroup1PracticeTime.map((student) =>
student.practiceTime);
const group2PracticeTime =
studentsGroup2PracticeTime.map((student) =>
student.practiceTime);
// Теперь можем вызывать нашу функцию поиска максимального
значения. Она принимает множество числовых аргументов, а у нас
есть только массив, вот тут нам и поможет оператор spread.
```

```
const maxTimeFromGroup1 = findMax(...group1PracticeTime); //
...group1PracticeTime - вытянет из массива все элементы и
передаст их в функцию как отдельные переменные.
// Это аналогично страшной и неудобной записи:
// findMax(group1PracticeTime[0], group1PracticeTime[1],
group1PracticeTime[2], group1PracticeTime[3],
group1PracticeTime[4])

console.log(maxTimeFromGroup1); // 160

const maxTimeFromGroup2 = findMax(...group2PracticeTime);
console.log(maxTimeFromGroup2); // 140

// Давайте также найдем максимально отработанное время среди двух
групп. Мы можем сделать это передав данные обоих массивов в
функцию таким образом:
// findMax(...group1PracticeTime, ...group2PracticeTime);
// А можем объединить два массива в один - это очень частая
операция и оператор расширения (spread) очень в этом помогает.

const bothGroupsTime = [...group1PracticeTime,
...group2PracticeTime];
// Для объединения двух массивов нам нужно вытащить их элементы в
один общий массив, поэтому мы объявляем новый массив, а в
качестве его элементов делаем расширение элементов первого и
второго массива. Также мы могли бы добавить в него и другие
элементы.

const maxTimeBothGroups = findMax(...bothGroupsTime);
console.log(maxTimeBothGroups); // 160
```

Rest operator

Rest оператор (от английского остальные, оставшиеся) - позволяет собрать оставшиеся аргументы функции в массив. Звучит немного странно, но этот оператор позволяет не перечислять все аргументы функции, как отдельные переменные, а получить их скопом в один массив. Для его использования необходимо в функции, которая принимает несколько аргументов перечислить необходимые аргументы, а все оставшиеся, которые мы хотим собрать в один массив записать как ...<имя массива>. Часто пишут ...rest. Давайте перепишем наш предыдущий пример, используя rest оператор и тем самым избавившись от псевдомассива arguments.

```
const studentsGroup1PracticeTime = [
  {
    firstName: "Ivanov",
    practiceTime: 56
  },
  {
    firstName: "Petrov",
    practiceTime: 120
  },
  {
    firstName: "Sidorov",
    practiceTime: 148
  },
  {
    firstName: "Belkin",
    practiceTime: 20
  },
  {
    firstName: "Avdeev",
    practiceTime: 160
  }
];

const studentsGroup2PracticeTime = [
  {
    firstName: "Mankov",
    practiceTime: 87
  },
  {
    firstName: "Kistin",
    practiceTime: 133
  },
  {
    firstName: "Kotlyarov",
    practiceTime: 140
  },
  {
    firstName: "Peskov",
    practiceTime: 10
  },
];

// Напишем не очень удобную, но показательную функцию, которая
умеет принимать неограниченное число аргументов, и находить
максимум среди них. Функция должна вызываться подобным образом:
```

```
const maximum = findMax(4, 7, 10);
function findMax(...values) { // тут мы принимаем все переданные
аргументы и с помощью rest оператора упаковываем их в массив
values.
  // На этот раз values уже настоящий массив и мы можем
использовать reduce для итерации по нему и нахождения
максимального числа.
  return values.reduce((acc, value) => {
    if (value > acc) return value;
    return acc;
  }, -Infinity);
};
// Создадим массивы только со значениями времени отработанными
студентами.
const group1PracticeTime =
studentsGroup1PracticeTime.map((student) =>
student.practiceTime);
const group2PracticeTime =
studentsGroup2PracticeTime.map((student) =>
student.practiceTime);

// Вызовем нашу функцию поиска максимума, используя оператор
spread.
const maxTimeFromGroup1 = findMax(...group1PracticeTime);
console.log(maxTimeFromGroup1); // 160

const maxTimeFromGroup2 = findMax(...group2PracticeTime);
console.log(maxTimeFromGroup2); // 140

// Давайте также найдем максимально отработанное время среди двух
групп.
const bothGroupsTime = [...group1PracticeTime,
...group2PracticeTime];

const maxTimeBothGroups = findMax(...bothGroupsTime);
console.log(maxTimeBothGroups); // 160
```

Давайте посмотрим на еще один пример:

```
const saveFullNameInDB = (firstName, lastName, ...additionals) =>
{
  saveFirstName(firstName);
  saveLastName(lastName);
  saveAdditionals(additionals); // Благодаря rest оператору мы
```

```
смогли собрать все дополнительные данные, которые были переданы
для сохранения в базе данных, и можем передать их одним массивом
в функцию сохранения дополнительных данных.
}
```

Чистые функции, иммутабельность

Чистые функции и иммутабельность - это понятия из функционального программирования. Они позволяют писать более понятный и надежный код.

Чистые функции

Чистые функции - это такие функции которые при вызове с одними и теми же параметрами всегда возвращают одинаковое значение, при этом такие функции оперируют данными только из полученных аргументов и никак не взаимодействуют с глобальными для них переменными. Чистые функции позволяют писать хорошо тестируемый код, т.к. они не зависят от глобальных переменных и всегда возвращают одинаковое значение, нам достаточно написать один тест для них, и мы можем быть уверены что они работают стабильно и правильно. Также чистые функции могут быть легко переиспользованы в другом коде, поэтому такие функции часто выносят как наборы утилит, которые используются всем проектом. Давайте рассмотрим несколько примеров:

```
const student = {
   firstName: "Ivan",
   age: 21,
};
// Функция вычисления года рождения. Принимает текущий год, и
вычисляет год рождения студента используя глобальные данные. Это
функция с побочными эффектами. Она сильно зависит от глобальной
переменной student.
const getYearOfBith = (currentYear) => {
   return currentYear - student.age;
}
console.log(getYearOfBith(2021)); // 2000
student.age = 25;
console.log(getYearOfBith(2021)); // 1996 - Мы вызывали функцию
дважды с одним и тем же параметром, но получили разный результат.
Это значит что мы не можем точно знать что вернет функция в тот
```

```
или иной момент работы программы, и мы не можем гарантировать что
код будет выполняться верно.
  // Чистая версия функции. Берет данные только из своих
аргументов.
const getYearOfBithPureVersion = (age, currentYear) => {
   return age - currentYear;
}
// Более сложный пример с мутацией (побочными эффектами), но
более частый на практике.
// Функция добавления нового ключа в объект. Принимает исходный
объект, имя ключа, и значение, которое надо добавить.
const addField = (object, key, value) => {
   object[key] = value;
   return object;
};
const updatedStudent = addField(student, 'lastName', 'Belkin');
console.log(student); // {firstName: "Ivan", age: 25, lastName:
"Belkin"} - вызвав нашу функцию добавления поля, мы изменили
начальный объект, что может привести к нежелательным последствиям
в остальном коде, которые порой очень сложно обнаружить. Например
дальше по коду может идти итерация объекта student и вывод только
начальных полей, но мы добавили в него третье поле, которое тоже
будет проитерировано.
console.log(updatedStudent); // {firstName: "Ivan", age: 25,
lastName: "Belkin"}
// Чистый вариант функции - нам нужно создать новый объект внутри
функции для изменения и возврата.
const addFieldPureVersion = (object, key, value) => {
   return { // возвращаем новый объект.
     ...object, // Воспользуемся оператором spread, для получения
копии свойств исходного объекта.
     [key]: value // Добавим новое свойство.
   }
};
const updatedStudentPure = addFieldPureVersion(student,
'practiceTime', 148);
console.log(student); // {firstName: "Ivan", age: 25, lastName:
"Belkin"} - на этот раз исходный объект не был изменен.
console.log(updatedStudentPure); // {firstName: "Ivan", age: 25,
lastName: "Belkin", practiceTime: 148}
```

Не всегда есть возможность писать только чистые функции. Часто нам нужно получить данные из базы, или сделать запрос на сервер, а также записать что-то на файловую систему, вывести данные в консоль - все это называется побочными

эффектами (side effect), которые очень нужны, но сильно усложняют тестирование кода и его предсказуемость. Рекомендуется разделять код на мелкие части, и большую часть писать чистыми функциями, и только маленькие кусочки оставлять с побочными эффектами.

Также важно следить за тем, чтобы в функции не изменять её аргументы. Возьмите за правило что аргументы функции должны использоваться только для чтения и будьте крайне осторожны при работе с объектами, которые приходят как аргументы в функцию. Т.к. объекты в функцию передаются по ссылке, а не по значению, очень легко присвоить эту ссылку новому объекту внутри функции и изменить исходный объект в дальнейшем. Баги связанные с таким мутированием объекта обнаружить порой очень тяжело. Для соблюдения этих правил есть полезные инструменты разработчика, такие как ESLint, который можно настроить на подсветку попыток изменения аргументов функции, а также специальные библиотеки для блокирования возможности править исходный объект или другие данные, например Immutable.

Замыкания

Представим что мы хотим создать на сайте счетчик нажатия на кнопку (например мы хотим отслеживать, сколько раз пользователь нажал на какую-нибудь кнопку). Для этого мы можем создать числовую переменную в коде и увеличивать её значение на единицу, каждый раз когда происходит нажатие на кнопку. Но название переменной может совпасть с названием другой переменной в другом скрипте, подключенному к странице, и этот другой скрипт может изменить данные в нашей переменной, чего нам не хотелось бы. То есть нам нужна защищенная переменная, чтобы доступ к ней имел только наш алгоритм, а также у нас может быть не одна кнопка, и тогда нам нужно несколько таких защищенных переменных. Лучшее решение для такой задачи спрятать эту переменную внутри функции - это называется замыкание.

Замыкание — это термин для механизма сохранения данных. Мы замыкаем данные внутри функции таким образом, чтобы к этим данным можно было обратиться и изменить их внутри этой функции, но при этом они были недоступны снаружи.

На этом механизме работают кэширование вычислений функций (разберем пример чуть ниже), скрытие переменных в модулях (когда несколько скриптов, подключенных к одной странице могут иметь одинаковые переменные и мешать работе друг друга), создание хранилищ данных, защищённых от доступа из внешнего кода.

Посмотрим, как организуется замыкание, на примере создания счетчика. В алгоритмах часто есть необходимость подсчитать какие-либо действия, для этого создается переменная счетчик, с первоначальным значением 0 и при необходимых действиях значение этого счетчика увеличивается на единицу. Это может быть например количество просмотров статьи, или количество нажатий на определенную кнопку на сайте, например на кнопку спасибо. Вот вариант создания счетчика через замыкание:

```
const createCounter = () => {
   const counter = 0;
   return () => {
       return ++counter;
   }
}


// Создаем счетчик.
const counter1 = createCounter();
counter1(); // 1
counter1(); // 2


//Создадим еще один счетчик. Каждый будет работать независимо.
const counter2 = createCounter();
counter2(); // 1
counter1(); // 3
```

Функция создания счётчика замыкает внутри себя значение счётчика. При вызове она возвращает нам новую функцию, которая может обращаться к этому значению, каждый раз увеличивая его на единицу. Благодаря созданию лексического окружения для каждого вызова функции мы можем создать два счётчика или более, и все они будут независимы.

Посмотрим на ещё один пример — создание функции с кэшированием результатов расчета. Бывают функцию, которые делают сложные и долгие расчеты, поэтому имеет смысл сохранять результат такого расчета с привязкой к аргументам, с которыми была вызвана функция, чтобы если функция будет вызвана с такими аргументами повторно, можно было взять уже готовый результат, а не рассчитывать его снова. Для примера возьмем простую функцию, которая вычисляет квадрат числа:

```
const closureFunction = () => {
```

```
    const cache = {};
    return (x) => {
        if (cache[x]) return cache[x];
        const result = x * x;
        cache[x] = result;
        return result;
    }
}

const chachedPow = closureFunction();
chachedPow(2); // 4
chachedPow(8); // 64
chachedPow(2); // 4 — тут функция возьмёт значение из кеша и не будет вычислять его заново. Это особенно эффективно работает, когда мы имеем дело со сложными и тяжёлыми вычислениями или, например, запросами каких-то ресурсов из базы данных или внешних источников. Тут нельзя забывать о валидации кеша. Он может стать неактуальным, если мы имеем дело с базой данных или внешними источниками данных.
```

В этом примере реализовано простое замыкание для сохранения результатов вычисления в кэш. Если в кеше результат уже есть, то мы возвращаем его. Если нет, то вычисляем и добавляем в кэш.

Обратиться к переменной cache снаружи функции нельзя: она защищена (замкнута) внутри функции closureFunction.

Основной подход к созданию замыканий:

- создать функцию;
- внутри неё объявить переменные, которые мы хотим в ней замкнуть: спрятать, сохранить и использовать в дальнейшем;
- вернуть из неё другую функцию, которая уже выполняет какое-то конкретное действие и может использовать замкнутые (спрятанные) данные.

На самом деле любая функция в JavaScript — это замыкание. Внутри неё находятся замкнутые переменные, которые недоступны снаружи. Для эффективного использования этого механизма удобно создавать функцию, которая возвращает другую функцию, позволяющую взаимодействовать с данными.

Ещё один пример использования замыкания — это сокрытие переменных внутри подключаемых модулей. Мы можем подключить на страницу много внешних скриптов, и каждый такой скрипт будет определять свои переменные, которые в какой-то момент могут совпасть с переменными из другого скрипта, и всё сломается. Именно поэтому придумали скрывать необходимые скрипту переменные внутри модуля (замыкания). Вот так выглядит один из вариантов:

```
(function () {
   const sliderTexts = ['Promo', 'Brands', 'Best'];
   function showSlider(texts) {
       console.log(texts[0]);
       console.log(texts[1]);
       console.log(texts[2]);
   }

   showSlider(sliderTexts);
}());
```

Мы создаём функцию, которая тут же вызывается и исполняет код. При этом переменная `sliderTexts` скрыта от других скриптов и не может быть переопределена.

🔥 **Внимание!** Замыкания могут приводить к созданию огромного числа лексических контекстов, и такой код может потребовать больше памяти.

Недостатки замыканий

Так как все функции — это замыкания, то при вызове функции создаётся лексическое окружение, которое занимает место в памяти компьютера. В коде часто бывают моменты, когда какая-то функция начинает вызываться чрезмерно много раз. Чаще всего это происходит при работе с рендерингом элементов и их обновлении. Для функции создаётся чрезмерно много лексических окружений, которые начинают быстро съедать память, — это называется утечкой памяти во время исполнения программы.

Следует анализировать, где используется функция, может ли она быть вызвана в цикле. В случае замедления страницы использовать её профилирование на потребление памяти. Это приходит с опытом. Подробнее об утечках памяти можно прочитать в статье на habr.ru.

Лексический контекст

Замыкания работают благодаря такому механизму, как лексическое окружение. Именно лексическое окружение (или лексический контекст) позволяет хранить все

эти замкнутые данные и обращаться к ним при вызове функции, а также позволяет функции иметь доступ к внешним данным.

Лексический контекст или **лексическое окружение** — это механизм в JavaScript, который позволяет функции во время её вызова получать доступ к переменным, константам и всему, что ей нужно. Каждый раз при вызове функции создаётся что-то вроде объекта словаря, который записывает все значения переменных и констант внутри функции, а также тех переменных и констант вне функции, к которым она обращается. Посмотрим на примерах:

```
const lastName = "Petrov";
// lexical environment: { lastName: "Petrov" }
const getFullName = (firstName) => {
   // lexical environment: { lastName: "Petrov", firstName:
<определяется в момент вызова функции> }
   const fullName = firstName + ' ' + lastName;
   // lexical environment: { lastName: "Petrov", firstName:
<определяется в момент вызова функции>, fullName: <вычисляется в
момент вызова функции> }
   console.log(fullName);
   return energy;
};

getFullName("Ivan"); // Ivan Petrov
// lexical environment в момент вызова функции становится таким:
{ lastName: "Petrov", firstName: "Ivan", fullName: "Ivan Petrov"
}
```

Когда функция только создаётся в коде, в её лексическом контексте ещё не определены многие переменные, но для них оставлен задел. В момент вызова функции они будут определены и заполнены. В этом примере скорость света определена как глобальная константа и будет доступна в функции. Если бы на её месте была переменная, то она попала бы в лексический контекст только в момент вызова функции, и значение будет использоваться то, которое будет в этой переменной на момент вызова функции. Это очень важный момент, который позволяет понять, почему при вызове функции мы часто получаем не тот результат, который хотели бы. Остальные переменные, аргументы и объявленные в самой функции переменные и константы до её вызова не имеют значений. В момент её вызова они будут записаны и использованы.

Всё это сделано специально, чтобы каждый вызов функции исполнялся в своём отдельном мире, но мог взаимодействовать с внешним миром посредством

передачи значений в лексический контекст. Именно из него функция получает значения и может их использовать.

Разберём пример, как работает лексическое окружение, чтобы лучше понять его важность и проблематику его неправильного использования:

Напишем функцию, которая будет строить дома: создадим массив функций, при вызове функции из каждого элемента массива должен выводиться номер дома.

```
const houses = [];

let i = 0;
while (i < 10) {
   let house = function () { // функция «дом»
       console.log(i); // выводит номер дома
   };
   houses.push(house);
   i++;
}

houses[0](); // 10 — у нулевого дома номер 10
houses[7](); // 10 — и у седьмого дома номер 10
```

Как мы можем убедиться, все наши дома имеют номер 10, хотя в цикле мы давали им номера по порядку и они должны быть корректными. Так происходит из-за лексического контекста, который создаётся в момент вызова функции. Вызов функции, которая должна показать номер дома, происходит уже после выполнения цикла. Так как переменная `i` объявлена именно как переменная, то её значение на момент вызова функции становится равным 10, это результат выполнения цикла. Соответственно, при вызове функции выводится номер дома 10.

Вот так выглядит лексическое окружение в момент выполнения функции `house`:

```
let house = function () { // функция «дом»
   // лексическое окружение: { i: 10 }
   console.log(i); // выводит номер дома
};
```

Для исправления такой ситуации необходимо хранить номер дома внутри блока цикла с использованием `let`:

```
const houses = [];

let i = 0;
while (i < 10) {
   let houseNumber = i; // Здесь мы создаём блочную переменную,
которая сохранит значение i для конкретной итерации цикла, и
именно её значение попадёт в лексическое окружение функции house.
   let house = function () { // функция «дом»
       console.log(houseNumber); // выводит номер дома
   };
   houses.push(house);
   i++;
}

houses[0](); // 0 — у нулевого дома номер 0
houses[7](); // 7 — и у седьмого дома номер 7
```

Или внутри функции `house`:

```
const houses = [];

let i = 0;
while (i < 10) {
   let house = function () { // функция «дом»
       let houseNumber = i; // Здесь мы создаём блочную
переменную, которая сохранит значение i, и именно её значение
попадёт в лексическое окружение функции house.

       console.log(houseNumber); // выводит номер дома
   };
   houses.push(house);
   i++;
}


houses[0](); // 0 — у нулевого дома номер 0
houses[7](); // 7 — и у седьмого дома номер 7
```

Рекурсия

Рекурсия — подход к вычислению, который позволяет сделать сложный расчёт итеративным.

Рекурсия — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя. Термин «рекурсия» используется в различных специальных областях знаний — от лингвистики до логики, но наиболее широкое применение находит в математике и информатике («Википедия»).

Или более простыми словами: рекурсия - это объект определенной структуры, который может содержать в себе другой объект такой же структуры, который в свою очередь также может содержать объект такой же структуры и так далее.

💡 Частым примером рекурсии является фракталы, которые встречаются в природе. Например капуста “Романеско” (изображение из wikipedia).

В программировании рекурсия — это вызов функцией самой себя. Рекурсия очень часто нужна там, где есть какие-то вложенные структуры и их нужно обработать. Например, на основе них создать или отобразить какие-то данные или элементы, вычислить какое-то значение на основе всех этих данных. Чаще всего это вложенные каталоги или другие структуры данных. Такие структуры не получится обработать обычным циклом, так как в этом случае мы можем пройти только по первому уровню вложенности, а более глубокие уровни останутся

необработанными. Также рекурсию можно применять при математических вычислениях, когда сложное вычисление поддаётся разбиению типа:

сложное вычисление = простая операция, применённая к более простому вычислению.

Например, вычисление степени: $2^{10} = 2^9 \times 2$.

```
const exponentiation = (base, exponent) => {
   // Всегда проверяйте данные, которые к вам пришли.
   if (typeof base !== 'number' || typeof exponent !== 'number')
return NaN;
   // Если наша степень больше нуля, вызываем рекурсию, то есть
берём основание и умножаем на нашу же функцию, только с
аргументом степени, уменьшенным на единицу.
   if (exponent > 0) return base * exponentiation(base, exponent
- 1);
   // В противном случае делаем выход из рекурсии, просто
возвращая основания, так как любое число в нулевой степени равно
одному.
   return 1;
}

const result = exponentiation(4, 4); // 1024
```

Конечно, возвести в степень проще с использованием обычного цикла. Тогда выполнение алгоритма займёт меньше ресурсов.

Второй пример с древовидной структурой полей формы:

```
// Типы полей в форме. Очень полезно выносить повторяющиеся
данные в справочники.
const fieldTypes = {
   text: 'textField',
   fieldSet: 'fieldSet',
};

// Тестовый объект полей формы, который может быть получен от
сервера (Backend).
const formData = [
   {
       fieldName: "First name",
       required: true,
       type: fieldTypes.text,
```

```
    },
    {
        fieldName: "Last name",
        required: false,
        type: fieldTypes.text,
    },
    {
        fieldName: "Address",
        required: true,
        type: fieldTypes.fieldSet,
        fields: [
            {
                fieldName: "State - Province",
                required: true,
                type: fieldTypes.text,
            },
            {
                fieldName: "Street",
                required: true,
                type: fieldTypes.text,
            },
            {
                fieldName: "House",
                required: true,
                type: fieldTypes.text,
            },
        ]
    },
];

// Наша функция, которая должна на основе этих данных построить
HTML-форму.
const getForm = (formStructure) => {
    // Всегда проверяйте данные, которые к вам пришли.
    if (!Array.isArray(formStructure)) return 'Wrong form
structure';
    let form = ''; // Это очень грубый пример. Мы будем создавать
форму просто как текст, а в реальности в этой переменной должны
быть узлы DOM или компоненты фреймворка (например, React.js).
    formStructure.forEach((element, index) => {
        // Если поле текстовое, то мы обработаем его сразу.
        if (element.type === fieldTypes.text) {
            form = form +
                `<div class="field-wrapper">
     <label>${element.fieldName}</label>${element.required ?
```

```
'<span class="required">*</span>' : ""}
    <input type="text">
  </div>`;
        }
        // Если это набор полей, то мы вызовем нашу функцию
рекурсивно для вложенного набора полей.
        if (element.type === fieldTypes.fieldSet) {
            form = form + `<div
class="fieldset">${getForm(element.fields)}</div>`;
        }
    });
    return form;
}

const result = getForm(formData);
console.log(result);
```

При использовании рекурсии главное — задать условия остановки, иначе можно получить бесконечный цикл вызовов функции. Когда у нас неправильно написано условие выхода из рекурсии, она может зациклиться до бесконечности, и такого лучше не допускать, так как выполняемый скрипт остановится с ошибкой, потому что движок JavaScript защищает нас от чрезмерной вложенности рекурсии. Еще бывают моменты когда у нас очень много данных для обработки и вложенность рекурсии превосходит лимит, установленный в движе JavaScript, тогда мы не можем использовать рекурсию, и можно попробовать переписать её на алгоритм с использованием циклов.

Вот пример неограниченной рекурсии и её результат:

```
const exponentiation = (base, exponent) => {
    if (typeof base !== 'number' || typeof exponent !== 'number')
return NaN;
    return base * exponentiation(base, exponent - 1);
 }
  const result = exponentiation(4, 4); // Uncaught RangeError:
Maximum call stack size exceeded
```

Домашнее задание

Задание 1

Дан массив `const arr = [1, 5, 7, 9]` с помощью `Math.min` и `spread` оператора, найти минимальное число в массиве, решение задание должно состоять из одной строки

Задание 2

Напишите функцию `createCounter`, которая создает счётчик и возвращает объект с двумя методами: `increment` и `decrement`. Метод `increment` должен увеличивать значение счетчика на 1, а метод `decrement` должен уменьшать значение счетчика на 1. Значение счетчика должно быть доступно только через методы объекта, а не напрямую.

Задание 3

Напишите рекурсивную функцию `findElementByClass`, которая принимает корневой элемент дерева DOM и название класса в качестве аргументов и возвращает первый найденный элемент с указанным классом в этом дереве.

Пример:

```
const rootElement = document.getElementById('root');
const targetElement = findElementByClass(rootElement,
'my-class');
console.log(targetElement);
```