

Функции

Функции — это механизм для многократного использования частей кода. Они позволяют запускать один и тот же код из разных мест программы без необходимости его копировать. Кроме того, если вы «спрячете» сложные фрагменты кода внутри функций, вам будет легче сосредоточиться на проектировании программы — так вы будете налаживать взаимодействие между функциями, а не барахтаться в мелких деталях, из которых состоит код этих фрагментов. Организация кода в виде небольших, легко контролируемых частей позволяет видеть общую картину и думать о строении программы на более высоком уровне.

Функции очень удобны, когда нужно многократно выполнять в программе некие расчеты или другие действия. Мы уже пользовались готовыми функциями, такими как `Math.random`, `Math.floor`, `alert`, `prompt` и `confirm`. В этой работе мы научимся создавать свои функции.

Базовое устройство функции

Строение функции указано ниже. Код внутри фигурных скобок называется телом функции — аналогично циклам, где код в фигурных скобках зовется телом цикла. Тело функции записывается в фигурных скобках

```
function () {  
    console.log("Делаем что-то");  
}
```

Давайте создадим простую функцию
Пусть она печатает фразу «Привет, мир!».

Задание

1. Создайте переменную `ourFirstFunction`
2. Присвойте ей значение `function () {};`
3. Внутри тела функции отдайте браузеру команду поприветствовать Мир записью в консоль `console.log("Привет, мир!");`

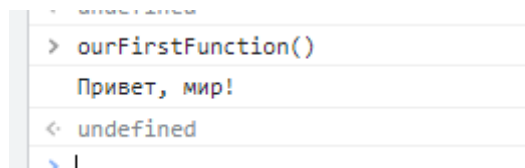
Вызов функции

Теперь наш код создает новую функцию, сохраняя ее в переменной `ourFirstFunction`. Следующий шаг - запустить код функции (то есть ее тело), а для этого нужно эту функцию вызвать. Делается все довольно просто - укажите ее имя, а следом — открывающую и закрывающую скобки, вот так:

```
ourFirstFunction(); // Привет, мир!
```

При вызове `ourFirstFunction` выполняется ее тело, то есть команда `console.log("Привет, мир!");`, и текст, который мы таким образом выводим, появляется в консоли на следующей строке: "Привет, мир!".

Однако, вызвав эту функцию из браузера, можно заметить в консоли еще одну строчку — с маленькой, указывающей влево стрелкой



```
> ourFirstFunction()
Привет, мир!
< undefined
> |
```

Это значение, которое **возвращает** функция.

Возвращаемое значение — это значение, которое функция выдает наружу, чтобы потом его можно было использовать где угодно в программе. В данном случае это `undefined`, поскольку мы не указывали возвращаемое значение в теле функции, мы лишь дали команду вывести текст в консоль. Функция всегда будет возвращать `undefined`, если в теле функции нет указания вернуть что-нибудь другое. К этому мы вернемся чуть позже.

Передача аргументов в функцию

Наша функция `ourFirstFunction` выводит одну и ту же строку при каждом вызове, однако хотелось бы, чтобы поведением функции можно было управлять. Чтобы функция могла изменять поведение в зависимости от значений, нам понадобятся аргументы. Список аргументов указывается в скобках после имени функции — как при ее создании, так и при вызове.

Допустим, мы все еще продолжаем работу над своим сайтом и как порядочные владельцы - собираемся приветствовать каждого своего пользователя по имени. Использование функции приветствия будет как нельзя кстати. Представим, что у нас есть какое то количество зарегистрированных пользователей, имена которых нам уже известны. Собственно аргументом нашей функции и станет вызов функции с этим аргументом. Создадим нашу функцию

Задание

1. Объявите функцию `sayHelloTo`
2. Чтобы поздороваться с человеком, имя которого передано в аргументе, присвойте ей значение `function (name) {};`
3. В тело функции вложите запись в консоль (`"Привет, " + name + "!"`);
};

Имя аргумента

Здесь мы создали функцию и сохранили ее в переменной `sayHelloTo`. При вызове функция печатает строку "Привет, " + `name` + "!", заменяя `name` на значение, переданное в качестве аргумента.

```
function ( argument ) {  
  console.log("Передан аргумент: " + argument);  
}
```

Аргумент можно использовать в теле функции. **И оно не равно переменной - это именно аргумент.**

Вызывая функцию, которая принимает аргумент, введите значение, которое вы хотите использовать в качестве этого аргумента в скобках после имени функции. Например, чтобы поздороваться с Михаилом, можно ввести:

```
sayHelloTo("Михаил"); //Привет, Михаил!
```

А с Анной поздороваться можно так:

```
sayHelloTo("Анна"); //Привет, Анна!
```

Каждый раз при вызове функции переданный нами аргумент `name` подставляется в строку, которую печатает функция. Поэтому, когда мы передаем значение "Михаил", в консоли появляется "Привет, Михаил!", а когда пишем "Анна", мы видим в консоли "Привет, Анна!".

Задание

1. Поприветствуйте своего одноклассника передав в аргумент его имя
2. Вызовите функцию еще раз, вложив в аргумент имя любимого киногероя

Печатаем котиков!

Мы рассмотрели функцию с одним аргументом. Но, функция может иметь множество аргументов, отвечающих за самые разные атрибуты работы самой функции. Например, переданным в функцию аргументом можно указывать, сколько раз требуется что-то сделать. Создадим функцию `drawCats`, она будет выводить в консоль смайлы — кошачьи мордочки (вот такие: `=^.^=`). Задавая аргумент `howManyTimes`, мы будем сообщать ей, сколько таких смайлов нужно напечатать. Телом функции будет знакомый вам цикл `for`, который повторяется столько раз, сколько указано в аргументе `howManyTimes` (поскольку переменная `i` сначала равна 0, а затем возрастает до значения `howManyTimes - 1`). На каждом повторе цикла функция выводит в консоль строку `i + « ^=.^=»`.

Задание

1. Создайте функцию `drawCats` с аргументом `howManyTimes`
2. В тело функции добавьте цикл `for` с условием `(var i = 0; i < howManyTimes; i++) {}`
3. Тело цикла должно выполнять запись строки в консоль - порядковый номер итерации конкатенированный со смайликом “кошачье мордочки”

Передача в функцию нескольких аргументов

Если все выполнено верно, вызвав эту функцию со значением 5 в качестве аргумента `howManyTimes` мы увидим следующее:

```
drawCats(5);  
0 =^.^=  
1 =^.^=  
2 =^.^=  
3 =^.^=  
4 =^.^=
```

Теперь чтобы напечатать 100 кошачьих мордочек нам нужно задать `howManyTimes` значение 100, и мы в любой момент можем гибко менять значение!

В функцию можно передать больше одного значения, задав несколько аргументов. Для этого перечислите аргументы в скобках после имени функции, разделив их запятыми.

```
function (argument1, argument2) {  
  console.log("Первый аргумент: " + argument1);  
  console.log("Второй аргумент: " + argument2);  
}
```

В теле функции можно использовать оба аргумента. Имена аргументов пишутся через запятую.

Выяснилось, что рисовать только котиков - давно не актуально. Вы решаете что пора показать себя настоящим олдфагом всея интернета, а для этого нужно вспомнить ASCII-смайлики и то, как вы умеете их использовать. Для этого усовершенствуем функцию `drawCats`, создав ее обновленную версию.

Задание

1. Создадим функцию `printMultipleTimes`, копию функции `drawCats`
2. Первое отличие - добавим еще один аргумент с именем `whatToDraw`.
3. Второе отличие - изменим тело цикла. Запись в консоль теперь будет состоять из порядкового номера итерации и аргумента `whatToDraw`
4. Для проверки вызовите `printMultipleTimes` с аргументами `(5, (*^_^^*))`

Операция “Тростниковая гибкость”

Функция `printMultipleTimes` печатает строку, переданную в аргументе `whatToDraw` столько раз, сколько указано в аргументе `howManyTimes`. Второй аргумент сообщает функции, что печатать, а первый — сколько раз это нужно печатать. Вызывая функцию с несколькими аргументами, перечислите нужные вам значения через запятую в скобках после имени функции. Конечный вид функции сейчас:

```
var printMultipleTimes = function (howManyTimes, whatToDraw) {  
  for (var i = 0; i < howManyTimes; i++) {  
    console.log(i + " " + whatToDraw);  
  }  
};
```

Теперь, чтобы напечатать кошачьи мордочки с помощью функции `printMultipleTimes`, вызывайте ее так:

```
printMultipleTimes(5, "=^.^=");
```

А чтобы четыре раза напечатать смайлик `^_^`, вызывайте `printMultipleTimes` так:

```
printMultipleTimes(4, "^_^");
```

Здесь при вызове `printMultipleTimes` мы указали значение 4 для аргумента `howManyTimes` и строку `«^_^»` для аргумента `whatToDraw`. В результате цикл выполнил четыре повтора (переменная `i` менялась от 0 до 3), каждый раз печатая `i + " " + "^ _ ^"`. Чтобы дважды напечатать `(>_<)`, введите:

```
printMultipleTimes(2, ">_<");
```

На этот раз мы передали число 2 для аргумента `howManyTimes` и строку `«(>_<)»` для `whatToDraw`.

Задание

1. Напечатать смайлик `^_^` 4 раза
2. Напечатать смайлик `(>_<)` 2 раза

Возврат значения из функции

До сих пор все наши функции выводили текст в консоль с помощью `console.log`. Это простой и удобный способ отображения данных, однако мы не сможем потом взять это значение из консоли и использовать его в коде. Вот если бы наша функция выдавала значение так, чтобы его потом можно было использовать в других частях программы... Как уже говорилось, функции могут возвращать значение. Вызвав функцию, которая возвращает значение, мы можем затем использовать это значение в своей программе (сохранив его в переменной, передав в другую функцию или объединив с другими данными). Например, следующий код прибавляет 5 к значению, которое возвращает вызов `Math.floor(1.2345)`:

```
5 + Math.floor(1.2345); //6
```

`Math.floor` — функция, которая берет переданное ей число, округляет его вниз до ближайшего целого значения и возвращает результат. Глядя на вызов функции `Math.floor(1.2345)`, представьте, что вместо него в коде стоит значение, возвращаемое этой функцией, — в данном случае это число 1.

Теперь давайте создадим функцию, которая возвращает значение. Вот функция `double`, которая принимает аргумент `number` и возвращает произведение `number * 2`. Иными словами, значение, которое возвращает эта функция, вдвое больше переданного ей аргумента. Чтобы вернуть из функции значение, используйте оператор `return`, после которого укажите само это значение.

Задание

1. Создайте функцию `double` с аргументом `number`
2. В теле функции воспользуйтесь оператором `return` для возврата значения удвоения аргумента `number`. Внутренняя часть функции будет выглядеть как `return number * 2;`
3. Вызовите функцию с аргументом 3
4. А теперь с аргументом 6

Вызов функции в качестве значения

Мы воспользовались `return`, вернув из функции `double` число `number * 2`. Возвращаемое значение показано следом за вызовом функции. Но, хоть функции и способны принимать несколько аргументов, вернуть они могут лишь одно значение. А если вы не укажете в теле функции, что именно надо возвращать, она вернет `undefined`.

Когда функция вызывается из кода программы, значение, возвращаемое этой функцией, подставляется туда, где происходит вызов. Давайте воспользуемся функцией `double`, чтобы удвоить пару чисел и затем сложить результаты:

```
double(5) + double(6); //22
```

Здесь мы дважды вызвали функцию `double` и сложили значения, которые вернули эти два вызова. То же самое было бы, если бы вместо вызова `double(5)` стояло число 10, а вместо `double(6)` — число 12.

Также вызов функции можно указать в качестве аргумента другой функции, при вызове которой в аргумент попадет значение, возвращенное первой функцией. Чтобы понять такую вложенность попробуем усложнить. На пальцах - вызов `double(3)` даст 6, так что `double(double(3))` упрощается до `double(6)`, что, в свою очередь, упрощается до 12. Вот как JavaScript вычисляет это выражение:

```
double(double(3)) -> double( 3 * 2 ) -> double( 6 ) -> 6 * 2 -> 12
```

Тело функции `double` возвращает `number * 2`, поэтому первым шагом заменяется `double(3)` на `3 * 2`. Следующим шагом мы заменяем `3 * 2` на 6. Затем мы делаем то же, заменяя `double(6)` на `6 * 2`. И наконец, последним шагом мы можем заменить `6 * 2` числом 12.

Задание

1. Вызовите функцию `double()`, аргументом которой будет функция `double()`, в свою очередь имеющую аргумент равный 3;

Упрощаем код с помощью функций

В предыдущих работах мы использовали методы `Math.random` и `Math.floor`, чтобы выбирать случайные слова из массивов и генерировать предсказания магического шара. В этом разделе мы перепишем генератор предсказаний, упростив его с помощью функций.

Для этого нам понадобится функция для выбора случайного слова. Вот как мы выбирали случайное слово из массива:

```
randomWords[Math.floor(Math.random() * randomWords.length)];
```

Если поместить этот код в функцию, можно многократно вызывать его для получения случайного слова из массива — вместо того чтобы вводить тот же код снова и снова. Например, давайте определим такую функцию `pickRandomWord`:

```
var pickRandomWord = function (words) {  
  return words[Math.floor(Math.random() * words.length)];  
};
```

Все, что мы сделали, — поместили прежний код в функцию. Теперь можно создать массив `randomWords`, который будет содержать наши предсказания:

```
var randomWords = ["Бесспорно", "Вероятнее всего", "Пока не ясно,  
попробуй снова", "Даже не думай"];
```

С помощью функции `pickRandomWord` мы можем получить случайное слово из этого массива, вот так:

```
pickRandomWord(randomWords); // "Вероятнее всего"
```

При этом нашу функцию можно использовать с любым массивом. Например, получить случайное имя из массива имен:

```
pickRandomWord(["Лена", "Света", "Вика", "Вероника"]); //  
"Вероника"
```

Генератор случайных предсказаний

Теперь давайте перепишем генератор предсказаний, используя нашу функцию для выбора случайных слов. Для начала вспомним, как выглядел код ранее:

```
var randomPredictions = ["Бесспорно", "Вероятнее всего", "Пока не  
ясно, попробуй снова", "Даже не думай"];  
  
var randomPrediction = randomPredictions[Math.floor(Math.random()  
* randomPredictions.length)];  
  
console.log("Ответ на ваш вопрос: " + randomPrediction);
```

Для начала добавим немного астрологии, пусть шар обосновывает свое решение именно ей.

Задание

1. Создайте массив `randomPlanets` с именами планет нашей солнечной системы
2. Данному массиву необходим метод извлечения случайного элемента
3. Создайте массив созвездий `randomGalaxies`
4. И ему также свой способ извлечения случайного элемента
5. Строка ответа должна состоять из сбора всех элементов в единое предсказание
'Ответ на ваш вопрос: ' + randomPrediction + ", потому что "
+ randomPlanet + " находится в созвездии " + randomGalaxy и
выведена в консоль

Рефакторим Magic 8 Ball

Теперь наш код разросся до внушительных размеров. Итоговая часть на данный момент:

```
var randomPredictions = ["Бесспорно", "Вероятнее всего", "Пока не  
ясно, попробуй снова", "Даже не думай"];  
var randomPlanets = ["Меркурий", "Венера", "Земля", "Марс",  
"Юпитер", "Сатурн", "Уран", "Нептун"];  
var randomGalaxies = ["Овен", "Козерог", "Телец", "Рыбы",  
"Рак"];  
  
var randomPrediction = randomPredictions[Math.floor(Math.random()  
* randomPredictions.length)];  
var randomPlanet = randomPlanets[Math.floor(Math.random() *  
randomPlanets.length)];  
var randomGalaxy = randomGalaxies[Math.floor(Math.random() *  
randomGalaxies.length)];  
  
console.log('Ответ на ваш вопрос: ' + randomPrediction + ", потому  
что " + randomPlanet + " находится в созвездии " + randomGalaxy);
```

Как вы уже заметили - мы трижды вызываем одну и ту же конструкцию для выбора случайного элемента из массива. Дублирование кода у программистов чаще вызвано ленью и непрофессионализмом, поэтому срочно перепишем наш код с использованием ранее упомянутой функции `pickRandomWord`:

```
var pickRandomWord = function (words) {  
    return words[Math.floor(Math.random() * words.length)];  
};
```

Задание

1. Добавьте код функции перед финальным выводом предсказания в консоль
2. Удалите строки генерации случайного предсказания, планеты и созвездия
3. Соберите финальную фразу предсказания, заменив прежние, теперь несуществующие переменные, на вызов функции, собранной из ее имени и имени массива в качестве аргумента, например
`pickRandomWord(randomPredictions)`

Рефакторим Magic 8 Ball 2

Итак, что же мы сделали?

1. Мы использовали функцию `pickRandomWord` для выбора случайного слова из массива вместо того, чтобы каждый раз писать `words[Math.floor(Math.random()*length)]`.
2. Вместо того чтобы сохранять каждое случайное слово в переменной перед тем, как добавлять его к итоговой строке, мы сразу объединяем возвращаемые из функции значения, формируя таким образом строку.

Вызов функции можно рассматривать как значение, которое эта функция возвращает, поэтому все, что мы тут делаем, — это объединяем строки. Как видите, новую версию программы гораздо легче читать. Да и писать ее тоже было легче, поскольку часть повторяющегося кода мы вынесли в функцию. Но давайте еще больше упростим происходящее, для этого сделаем генератор предсказаний функцией! Для этого нам нужно обернуть все происходящее в функцию без аргументов (действия внутри самодостаточны), создать переменную в которую присвоим собираемую строку и собственно вернем значение нашей функции во вне!

Задание

1. Оберните всю нашу конструкцию в функцию `getFullPrediction() {}`
2. Значение генерируемой строки из `console.log` присвойте переменной `randomString`
3. В самом низу функции `getFullPrediction()` добавьте вывод переменной из функции, используя `return randomString;`
4. Теперь для получения рекомендации вызовите функцию `getFullPrediction`, сделайте это 3 раза

Ранний выход из функции по return

Наша новая функция `getFullPrediction()` представляет собой все тот же код, помещенный в тело функции без аргументов. Мы добавили лишь одну строку, содержащую `return`, где мы возвращаем сгенерированную строку `randomString`. Трижды вызвав функцию `getFullPrediction()`, мы каждый раз получали новое предсказание. Теперь весь код находится в функции, и это означает, что для генерации мы можем просто вызывать эту функцию, а не копировать в консоль один и тот же код каждый раз, когда понадобится кого-нибудь обрадовать предсказанием будущего.

Как только JavaScript, выполняя код функции, встречает оператор `return`, он завершает функцию, даже если после `return` еще остался какой-нибудь код. Оператор `return` часто используют, чтобы выйти из функции в самом начале, если какие-нибудь из переданных аргументов имеют некорректные значения — то есть если с такими аргументами функция не сможет правильно работать.

Для примера напишем следующую функцию. Она будет возвращать строку с информацией о пятой букве вашего имени. Если в имени, переданном в аргументе `name`, меньше пяти букв, будет выполнен `return`, чтобы сразу же выйти из функции. При этом оператор `return` в конце функции (тот, что возвращает сообщение о пятой букве) так и не будет выполнен.

Задание

1. Создайте функцию `fifthLetter` с аргументом `name`
2. В тело функции добавьте условное ветвление, которое будет проверять длину (менее 5 символов) введенного имени с помощью метода `.length`
3. Если количество символов имени менее 5, то и взять его у нас шансов нет. Следовательно нам необходимо добавить в тело условного ветвления срочный выход - `return`
4. В противном случае возвращаем из функции строку "Пятая буква вашего имени: " соединенную с 5 символом имени

Проверяем нашу функцию

Вид нашей функции теперь:

```
var fifthLetter = function (name) {  
  if (name.length < 5) {  
    return;  
  } else {  
    return "Пятая буква вашего имени: " + name[4];  
  }  
};
```

Сначала мы проверяем длину переданного имени — уж не короче ли оно пяти символов? Если это так, в строке мы выполняем `return`, чтобы незамедлительно выйти из функции. Давайте попробуем эту функцию в деле.

Задание

1. Вызовите функцию с аргументом "Николай"
2. Вызовите функцию с аргументом "Ник"

Многократное использование return

В имени Николай больше пяти букв, так что функция `fifthLetter` благополучно завершается, вернув пятую букву имени Николай, то есть **л**. Когда мы вызвали `fifthLetter` для имени Ник, функция распознала, что имя недостаточно длинное, и сразу завершилась, выполнив оператор `return` в строке `.`. Поскольку никакого значения после этого `return` не указано, функция вернула `undefined`.

Вместо конструкции `if... else` можно многократно использовать `return` внутри разных конструкций `if`, чтобы возвращать из функции разные значения в зависимости от входных данных. Предположим, вы пишете игру, в которой игроки награждаются медалями согласно набранному очкам. Счету меньше трех очков соответствует бронзовая медаль, счету от трех до шести — серебряная, а счету от семи и выше — золотая.

```
var medalForScore = function (score) {  
  if (score < 3) {  
    return "Бронзовая";  
  }  
  if (score < 7) {  
    return "Серебряная";  
  }  
  return "Золотая";  
};
```

В строке `.` мы возвращаем значение "Бронзовая" и выходим из функции, если счет меньше трех очков. Если мы достигли следующую "ветку" - `(score < 7)`, значит, счет как минимум равен трем очкам, поскольку, будь он меньше трех, мы бы уже вышли из функции (выполнив `return` в первом операторе `if`). И наконец, если мы достигли строки `return "Золотая";`, значит, на счету как минимум семь очков, проверять больше нечего и можно спокойно вернуть значение "Золотая". Хотя мы проверяем здесь несколько условий, необходимости использовать цепочку конструкций `if... else` нет. Мы используем `if... else`, когда хотим убедиться, что будет выбран лишь один из вариантов.

Однако если в каждом варианте выполняется `return`, это также гарантирует однозначный выбор (поскольку выйти из функции можно лишь один раз).

Сокращенная запись при создании функций

Есть длинный и короткий способы записи функций. Часто используют длинную запись, поскольку она наглядно демонстрирует, что функция хранится в переменной. Тем не менее вам стоит знать и о короткой записи, поскольку ее используют многие JavaScript-разработчики. Возможно, и вы сами, достаточно поработав с функциями, предпочтете короткую запись.

Вот пример длинной записи:

```
var double = function (number) {  
    return number * 2;  
};
```

Короткая запись той же функции выглядит так:

```
function double(number) {  
    return number * 2;  
}
```

Как видите, при длинной записи мы явно создаем переменную и сохраняем в ней функцию, так что имя `double` записывается прежде ключевого слова `function`. На техническом это называется **Function Expression** (функциональное выражение).

Напротив, при короткой записи сначала идет ключевое слово `function`, а затем название функции. В этом случае JavaScript создает переменную `double` неявным образом. На техническом это называется **Function Declaration** (объявление функции)

«Классическим» объявлением функции является укороченный вариант, вида

```
function имя(параметры) {  
    ...  
}
```

Итого:

- **Function Declaration** – функция, объявленная в основном потоке кода.
- **Function Expression** – объявление функции в контексте какого-либо выражения, например присваивания.

На техническом сленге длинная запись называется функциональным выражением, а короткая — объявлением функции и несмотря на немного разный вид, по сути две эти записи делают одно и то же

Задание

1. Перепишите функцию `fifthLetter` в классическую укороченную форму **Function Declaration**

Итого

Функции позволяют повторно использовать фрагменты кода. Они могут работать по-разному в зависимости от переданных аргументов и могут возвращать значение в то место кода, откуда они были вызваны. Также функции дают возможность называть фрагменты кода понятными именами, чтобы, глядя на название, мы могли сразу понять, что функция делает.

1. Математические расчеты и функции

Создайте две функции, `add` и `multiply`; пусть каждая принимает по два аргумента.

Функция `add` должна складывать аргументы и возвращать результат, а функция `multiply`

- перемножать аргументы. С помощью только этих двух функций вычислите

следующее несложное выражение:

$36325 * 9824 + 777$

2. Опросник

Напишите функцию `askSomeQuestion`. Она должна иметь 3 аргумента:

1. `question` - строка-вопрос, выполняемая с помощью функции `confirm()`
2. `yes` - функция, выполняющая `alert("Вы согласились.");`
3. `no` - функция, выполняющая `alert("Вы не согласны.");`

Вызовите функцию `askSomeQuestion` с любым вопросом в аргументе на ваш выбор.

В зависимости от ответа - она должна показать результат выполнения одной из функций - `yes` или `no`

3. Совпадают ли массивы?

Напишите функцию `areArraysSame`, которая принимает два массива с числами в качестве аргументов. Она должна возвращать `true`, если эти массивы одинаковые (то есть содержат одни и те же числа в одном и том же порядке), или `false`, если массивы различаются. Вам понадобится перебрать все значения из первого массива в цикле `for` и убедиться, что они совпадают со значениями из второго массива. Вы можете вернуть `false` прямо из тела `for`, если обнаружите несовпадающие значения. Вы можете сразу выйти из функции, пропустив цикл `for`, если у массивов разная длина.

Убедитесь, что ваша функция работает правильно, запустив такой код:

```
areArraysSame([1, 2, 3], [4, 5, 6]); // false
areArraysSame([1, 2, 3], [1, 2, 3]); // true
areArraysSame([1, 2, 3], [1, 2, 3, 4]); // false
```