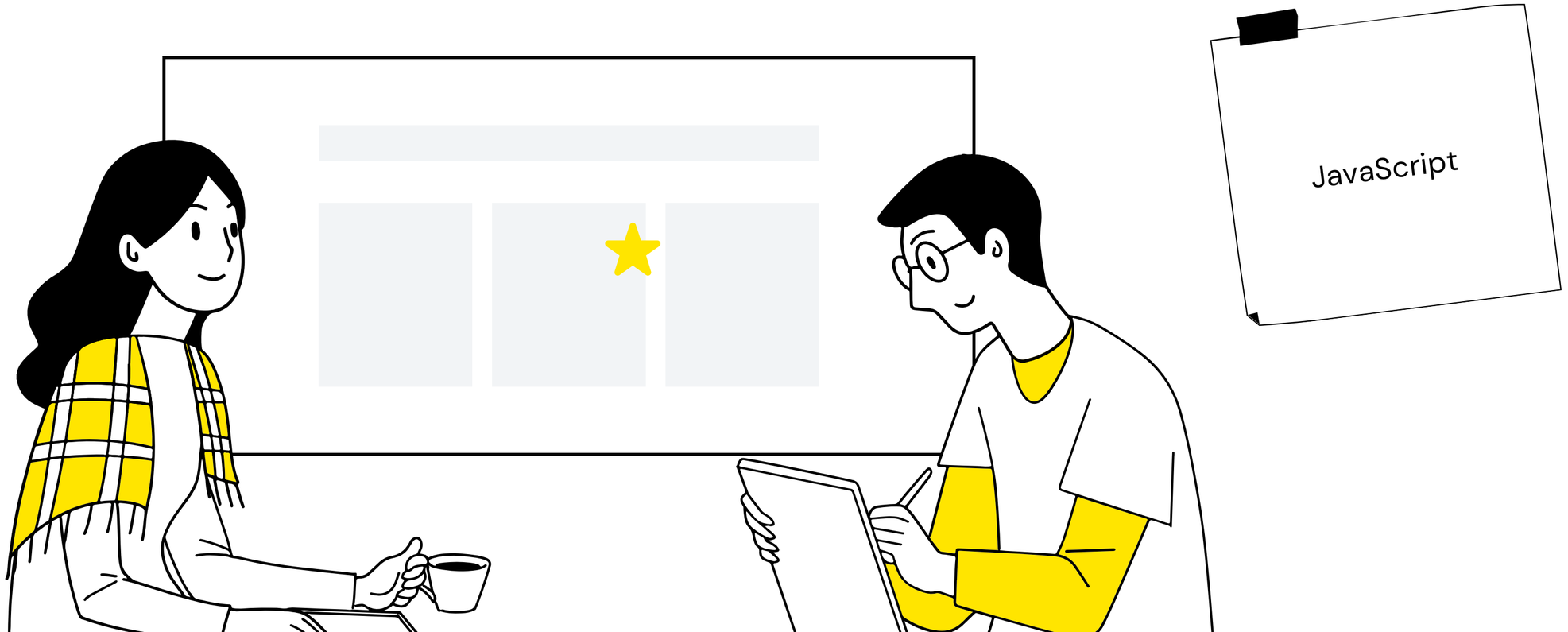
JavaScript



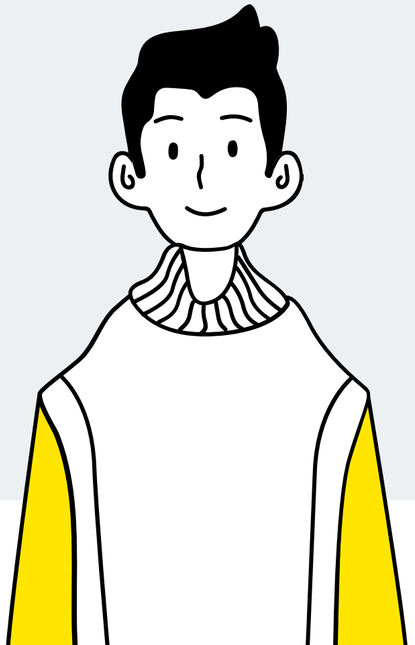
План

1. Методы примитивов
2. Числа, строки
3. Массивы
4. Методы массивов

Методы примитивов

JavaScript позволяет нам работать с примитивными типами данных – строками, числами и т.д., как будто они являются объектами. У них есть и методы.

Одна из лучших особенностей объектов – это то, что мы можем хранить функцию как одно из свойств объекта.



Методы примитивов

В JavaScript примитивные типы данных (числа, строки, булевы значения и т.д.) не являются объектами и поэтому не могут хранить дополнительные данные или свойства, как это могут делать объекты.

Например, если вы попытаетесь добавить свойство к строке (которая является примитивом), это не сработает:

```
let str = "Привет";
str.newProperty = "Значение"; // Попытка добавить новое свойство
console.log(str.newProperty); // undefined
```

В этом примере мы пытаемся добавить свойство newProperty к строке str. Однако, когда мы пытаемся вывести это свойство в консоль, мы получаем undefined, потому что примитивы не могут хранить дополнительные свойства или методы.
Это отличает примитивы от объектов в JavaScript, поскольку объекты могут хранить дополнительные свойства и методы

Методы примитивов

```
let obj = {}; // Создаем новый объект
obj.newProperty = "Значение"; // Добавляем свойство к объекту
console.log(obj.newProperty); // "Значение"
```

В этом примере мы добавляем свойство newProperty к объекту obj и затем успешно выводим его в консоль. Это возможно, потому что obj является объектом, а не примитивом.

Методы примитивов

Примитив как объект

Вот парадокс, с которым столкнулся создатель JavaScript:

- Есть много всего, что хотелось бы сделать с примитивами, такими как строка или число. Было бы замечательно, если бы мы могли обращаться к ним при помощи методов.
- Примитивы должны быть лёгкими и быстрыми насколько это возможно.

Выбранное решение, хотя выглядит оно немного неуклюже:

1. Примитивы остаются примитивами. Одно значение, как и хотелось.
2. Язык позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов.
3. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который предоставляет нужную функциональность, а после удаляется.

Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean, Symbol и BigInt. Таким образом, они имеют разный набор методов.

Методы примитивов

1. Строка str – примитив. В момент обращения к его свойству, создаётся специальный объект, который знает значение строки и имеет такие полезные методы, как toUpperCase().
2. Этот метод запускается и возвращает новую строку (показывается в alert).
3. Специальный объект удаляется, оставляя только примитив str.

Получается, что примитивы могут предоставлять методы, и в то же время оставаться «лёгкими».

```
let str = "Привет";

alert( str.toUpperCase() ); // ПРИВЕТ
```

Методы примитивов

1. Примитивы в JavaScript: это значения "примитивных" типов, таких как строка (string), число (number), булево значение (boolean), символ (symbol), null, undefined и bigint.
2. Примитивы могут предоставлять методы, несмотря на то что они не являются объектами. Это достигается с помощью временных объектов-оберток, которые создаются при вызове метода и удаляются после его выполнения.
3. Например, у строки есть метод toUpperCase(), который возвращает новую строку с преобразованными в верхний регистр символами.
4. Однако, несмотря на то что примитивы могут вести себя как объекты, они не могут хранить дополнительные данные. Если вы попытаетесь добавить новое свойство к примитиву, это не сработает.
5. Важно помнить, что null и undefined не имеют своих методов в JavaScript.

Числа

Тип данных Number: В JavaScript все числа представлены в 64-битном формате IEEE-754, который также называют "double precision". Это означает, что мы можем безопасно хранить числа от -(2^53 - 1) до 2^53 - 1.

```
let bigNumber = 9007199254740991; // это максимальное безопасное число в JavaScript
```

Infinity и -Infinity

В JavaScript, Infinity и -Infinity являются специальными числовыми значениями, которые представляют положительную и отрицательную бесконечность соответственно. Они могут возникнуть в результате определенных математических операций, которые не имеют конечного результата.

```
console.log(1 / 0); // Infinity
console.log(-1 / 0); // -Infinity
console.log("не число" / 2); // NaN, так как деление не числовой строки на число дает NaN
```

isNaN и isFinite

Функции isNaN и isFinite: isNaN проверяет, является ли значение NaN, а isFinite проверяет, является ли значение конечным числом.

isNaN: Эта функция проверяет, является ли значение NaN (Not a Number). Это полезно, потому что NaN не равно ни одному другому значению, включая само NaN. То есть, вы не можете просто использовать === или == для проверки на NaN.

isFinite: Эта функция проверяет, является ли значение конечным числом. Это полезно для обнаружения значений Infinity и -Infinity, которые могут возникнуть в результате таких операций, как деление на ноль.

```
console.log(isNaN( number: "не число")); // true
console.log(isFinite( number: "123")); // true
```

Обе эти функции полезны при работе с числами, особенно когда вы обрабатываете ввод пользователя или данные, которые могут не быть числами. Они позволяют вам обнаружить и корректно обработать особые числовые значения, предотвращая возможные ошибки и неожиданное поведение вашего кода.

parseInt и parseFloat

Функции parseInt и parseFloat в JavaScript используются для преобразования строк в числа. Они полезны, когда вам нужно работать с числовыми данными, которые были получены в виде строк, например, из текстовых полей формы, файлов CSV или JSON-ответов от сервера.

parseInt и parseFloat

parseInt: Эта функция преобразует строку в целое число. Если в строке есть нечисловые символы, parseInt преобразует столько символов в начале строки, сколько может, и игнорирует остальные. Если первый символ не может быть преобразован в число, parseInt возвращает NaN.

```
console.log(parseInt( string: '123')); // 123
console.log(parseInt( string: '123abc')); // 123
console.log(parseInt( string: 'abc123')); // NaN
```

parseInt и parseFloat

parseFloat: Эта функция преобразует строку в число с плавающей точкой (то есть, число, которое может иметь десятичные знаки). Как и parseInt, parseFloat преобразует столько символов в начале строки, сколько может, и игнорирует остальные.

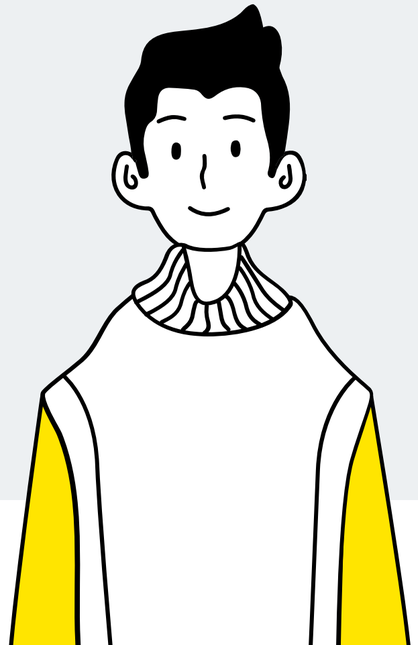
```
console.log(parseFloat( string: '123.45')); // 123.45
console.log(parseFloat( string: '123.45abc')); // 123.45
console.log(parseFloat( string: 'abc123.45')); // NaN
```

parseInt и parseFloat

Обе эти функции игнорируют начальные пробелы в строке, поэтому они могут безопасно использоваться с данными, которые могут иметь дополнительные пробелы в начале или в конце.
Важно отметить, что parseInt и parseFloat возвращают NaN, если они не могут преобразовать строку в число. Это может быть полезно для обнаружения некорректного ввода или ошибок в данных.

toFixed

Методы toFixed и toPrecision в JavaScript используются для форматирования чисел. Они оба возвращают строковое представление числа, но делают это немного по-разному.



toFixed

toFixed: Этот метод форматирует число, используя фиксированное количество цифр после десятичной точки. Вы можете указать количество цифр в качестве аргумента метода. Если аргумент не указан, число округляется до ближайшего целого.

```
let num = 123.456;
console.log(num.toFixed()); // "123"
console.log(num.toFixed( fractionDigits: 2)); // "123.46"
console.log(num.toFixed( fractionDigits: 5)); // "123.45600"
```

Округление чисел

В JavaScript существуют несколько методов для округления чисел:

1. Math.round()
2. Math.floor():
3. Math.ceil()
4. Math.trunc()

Math.round()

Math.round(): Этот метод округляет число до ближайшего целого числа.
Если дробная часть числа равна 0.5 или больше, число округляется вверх.

```
console.log(Math.round( x: 3.5)); // 4
console.log(Math.round( x: 3.4)); // 3
```

Math.floor()

Math.floor(): Этот метод округляет число вниз до ближайшего меньшего целого числа.

```
console.log(Math.floor( x: 3.9)); // 3
console.log(Math.floor( x: 3.1)); // 3
```

Math.ceil()

Math.ceil(): Этот метод округляет число вверх до ближайшего большего целого числа.

```
console.log(Math.ceil( x: 3.1)); // 4
console.log(Math.ceil( x: 3.9)); // 4
```

Math.trunc()

Math.trunc(): Этот метод просто отбрасывает все после десятичной точки, без округления.

```
console.log(Math.trunc( x: 3.1)); // 3
console.log(Math.trunc( x: 3.9)); // 3
```

Округление чисел

Важно отметить, что все эти методы возвращают целые числа. Если вы хотите округлить число до определенного количества знаков после запятой, вы можете использовать метод toFixed(), который возвращает строку.

Неточные вычисления

В JavaScript и многих других языках программирования существует проблема с неточными вычислениями чисел с плавающей точкой. Это связано с тем, как числа представлены внутри компьютера. Все числа в JavaScript хранятся в формате 64-битного числа с плавающей точкой (также известного как "double precision floating point format"). Этот формат может точно представлять целые числа и числа с плавающей точкой до определенного предела, но когда числа становятся слишком большими или слишком маленькими, точность может страдать.

Неточные вычисления

В JavaScript и многих других языках программирования существует проблема с неточными вычислениями чисел с плавающей точкой. Это связано с тем, как числа представлены внутри компьютера. Все числа в JavaScript хранятся в формате 64-битного числа с плавающей точкой (также известного как "double precision floating point format"). Этот формат может точно представлять целые числа и числа с плавающей точкой до определенного предела, но когда числа становятся слишком большими или слишком маленькими, точность может страдать.

Неточные вычисления

В этом примере, хотя математически 0.1 + 0.2 равно 0.3, JavaScript возвращает 0.30000000000000004 из-за неточности представления чисел с плавающей точкой.

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

Это важно учитывать при работе с числами с плавающей точкой в JavaScript, особенно при сравнении на равенство. Вместо прямого сравнения чисел с плавающей точкой, часто лучше сравнивать их с некоторым малым "допуском" или "погрешностью".

Неточные вычисления

Решение.

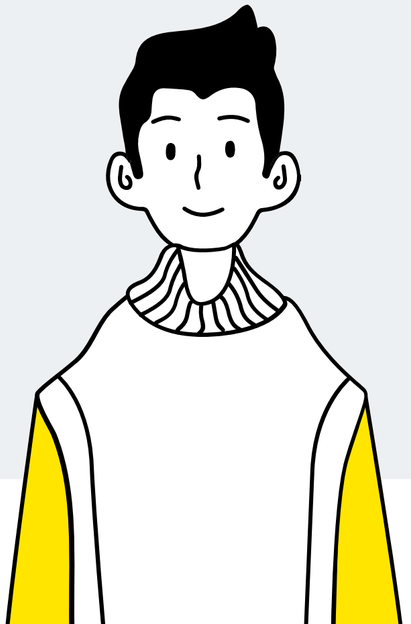
1. Использование библиотек для точных вычислений: Существуют специализированные библиотеки, такие как decimal.js или BigNumber.js, которые предоставляют функции для точного выполнения математических операций.
2. Использование целых чисел вместо чисел с плавающей точкой: Если возможно, можно избегать использования чисел с плавающей точкой, выполняя все вычисления с использованием целых чисел. Например, вместо того чтобы работать с долларами и центами как с числами с плавающей точкой, можно работать только с центами как с целыми числами.
3. Округление результатов до нужного количества знаков после запятой: Если неточности возникают из-за большого количества знаков после запятой, можно округлить результаты до нужного количества знаков после запятой с помощью методов toFixed или toPrecision.

Строки

Строки в JavaScript являются неизменяемыми. Это означает, что после создания строки вы не можете изменить ее содержимое. Вместо этого, когда вы выполняете операции, которые кажутся изменяющими строку, на самом деле создается новая строка.

```
let str = 'Hi';

str[0] = 'h'; // ошибка
alert( str[0] ); // не работает
```



Строки

Можно создать новую строку и записать её в ту же самую переменную вместо старой.

```
let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi
```

Строки

Строки можно сравнивать:
Строки можно сравнивать с помощью операторов больше/меньше или равно.

Строки

```
console.log( 'Z' > 'A' ); // true, потому что код символа 'Z' больше, чем 'A'
console.log( 'Glow' > 'Glee' ); // true, потому что 'o' > 'e'
console.log( 'Bee' > 'Be' ); // true, потому что 'Bee' длиннее, чем 'Be'
```

Сравнение строк в JavaScript происходит посимвольно и использует коды символов Unicode.
Вот как это работает:

1. Сначала сравниваются первые символы строк.
2. Если первые символы идентичны, то сравниваются вторые символы, и так далее.
3. Если все символы в строках идентичны, то строки считаются равными.
4. Если одна строка является подстрокой другой, то короткая строка считается меньшей.
5. Сравнение продолжается до тех пор, пока не будет найдено различие или пока не закончатся символы в обеих строках.

Строки

Важно отметить, что при сравнении строк JavaScript использует "лексикографический" или "словарный" порядок. Также стоит учесть, что регистр символов имеет значение, так как коды символов для строчных и прописных букв различаются. Например, 'a' > 'A' будет true, потому что код символа 'a' больше, чем 'A'.

Методы строк

В JavaScript есть множество встроенных методов для работы со строками, включая toLowerCase(), toUpperCase(), indexOf(), lastIndexOf(), slice(), substring(), substr(), replace() и многие другие.

Методы строк

1. charAt(index): Возвращает символ в указанной позиции.

```
let str = "Hello, world!";
console.log(str.charAt(0)); // Выводит "H"
```

2. indexOf(substring, position): Возвращает позицию первого вхождения подстроки, начиная с указанной позиции. Если подстрока не найдена, возвращает -1.

```
let str = "Hello, world!";
console.log(str.indexOf("world")); // Выводит 7
```

3. lastIndexOf(substring, position): Возвращает позицию последнего вхождения подстроки, начиная с указанной позиции и двигаясь назад. Если подстрока не найдена, возвращает -1.

```
let str = "Hello, world! world!";
console.log(str.lastIndexOf("world")); // Выводит 14
```

Методы строк

4. startsWith(substring) и endsWith(substring): Возвращают true, если строка начинается или заканчивается указанной подстрокой.

```
let str = "Hello, world!";
console.log(str.startsWith("Hello")); // Выводит true
console.log(str.endsWith("!")); // Выводит true
```

5. toLowerCase() и toUpperCase(): Возвращают новую строку, где все символы преобразованы в нижний или верхний регистр.

```
let str = "Hello, World!";
console.log(str.toLowerCase()); // Выводит "hello, world!"
console.log(str.toUpperCase()); // Выводит "HELLO, WORLD!"
```

6. trim(): Возвращает новую строку, где удалены пробелы с обоих концов строки.

```
let str = "   Hello, world!   ";
console.log(str.trim()); // Выводит "Hello, world!"
```

Методы строк

7. replace(oldSubstring, newSubstring): Возвращает новую строку,
где первое вхождение oldSubstring заменено на newSubstring.

```
let str = "Hello, world!";
console.log(str.replace( searchValue: "world",  replaceValue: "JavaScript")); // Выводит "Hello, JavaScript!"
```

8. split(delimiter): Разбивает строку на массив подстрок,
используя указанный разделитель.

```
let str = "Hello, world!";
console.log(str.split( separator: " ")); // Выводит ["Hello,", "world!"]
```

Это не все методы, которые предоставляет JavaScript для работы со строками,
но это наиболее распространенные и полезные в повседневной работе.

Методы строк

Подстрока — это часть строки, которая может начинаться и заканчиваться в любом месте внутри строки.
В других словах, подстрока — это набор символов, взятых из строки без изменения их порядка.
Например, в строке "Hello, World!", подстроками могут быть "Hello", "World", "llo, Wo", и даже пустая строка "".
В JavaScript есть несколько методов для работы с подстроками, включая:

- slice(start, end): Возвращает подстроку, начинающуюся с позиции start и заканчивающуюся (но не включая) позицию end.
- substring(start, end): Похож на slice(), но substring() не принимает отрицательные индексы.
- substr(start, length): Возвращает подстроку, начинающуюся с позиции start и имеющую длину length.
- indexOf(substring): Возвращает позицию первого вхождения подстроки в строку или -1, если подстрока не найдена.
- lastIndexOf(substring): Возвращает позицию последнего вхождения подстроки в строку или -1, если подстрока не найдена.
- replace(oldSubstring, newSubstring): Заменяет первое вхождение oldSubstring на newSubstring в строке и возвращает новую строку.

Эти методы позволяют извлекать, находить и заменять подстроки в строках.

includes

Метод includes() в JavaScript используется для проверки, содержит ли строка указанную подстроку. Этот метод возвращает true или false.

str.includes(searchString, position)

- searchString – подстрока, которую мы хотим найти в строке.
- position – необязательный параметр, указывающий позицию в строке, с которой начинается поиск. Если позиция не указана, поиск начинается с начала строки.

```
let str = "Hello, world!";
console.log(str.includes("world")); // Выводит true
console.log(str.includes("JavaScript")); // Выводит false
console.log(str.includes("world", 8)); // Выводит true
console.log(str.includes("world", 9)); // Выводит false
```

includes

В этом примере первый вызов includes("world") возвращает true, потому что "world" присутствует в строке. Второй вызов includes("JavaScript") возвращает false, потому что "JavaScript" отсутствует в строке. Третий вызов includes("world", 8) возвращает true, потому что "world" присутствует в строке, начиная с 8-го индекса. Четвертый вызов includes("world", 9) возвращает false, потому что "world" отсутствует в строке, начиная с 9-го индекса.

Массивы

Массивы – это одна из основных структур данных в JavaScript, которая используется для хранения упорядоченных коллекций значений.

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.
Но довольно часто мы понимаем, что нам необходима упорядоченная коллекция данных, в которой присутствуют 1–й, 2–й, 3–й элементы и т.д.
Например, она понадобится нам для хранения списка чего–либо: пользователей, товаров, элементов HTML и т.д.
В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Массивы

Массивы и объекты в JavaScript имеют многие общие черты: они оба являются типами данных, которые могут хранить множественные значения, и они оба обрабатывают значения как свойства или элементы с доступом по ключу или индексу.

Массивы

Однако есть несколько ключевых отличий:

1. Упорядоченность: Массивы являются упорядоченными коллекциями данных, а объекты – нет. В массиве каждому элементу присваивается числовой индекс, начиная с 0, и элементы упорядочены по этим индексам. В объекте элементы или свойства идентифицируются по ключам, и нет гарантированного порядка, в котором свойства объекта возвращаются в циклах или при сериализации объекта.
2. Методы: Массивы в JavaScript имеют множество встроенных методов, которые упрощают обработку и манипулирование массивами, такие как push(), pop(), shift(), unshift(), splice(), slice(), map(), reduce(), и т.д. Хотя объекты также имеют встроенные методы, они более общие и не связаны напрямую с структурой данных объекта.
3. Использование: Объекты часто используются для создания сложных структур данных, где каждое значение имеет уникальное имя или "ключ". Объекты хорошо подходят для моделирования реальных объектов, где каждое свойство имеет свое уникальное имя. Массивы, с другой стороны, идеально подходят для хранения упорядоченных коллекций, где порядок имеет значение, например, список покупок, очередь задач и т.д.

В целом, выбор между массивами и объектами зависит от конкретной ситуации и того, какие структуры данных лучше всего подходят для решения конкретной задачи.

Массивы

Создание массивов: Массивы можно создать с помощью квадратных скобок [] или через конструктор new Array(). Например:

```
let array1 = [1, 2, 3, 4, 5]; // создание массива с помощью квадратных скобок
let array2 = new Array( items: 1, 2, 3, 4, 5); // создание массива с помощью конструктора
```

Массивы

Индексация массивов: Массивы в JavaScript являются индексированными, начиная с нуля. Это означает, что первый элемент массива имеет индекс 0, второй – 1, и так далее. Индексы используются для доступа к элементам массива:

```
let array = [1, 2, 3, 4, 5];
console.log(array[0]); // выводит 1
console.log(array[1]); // выводит 2
```

Массивы

Длина массива: Вы можете узнать длину массива (количество его элементов) с помощью свойства .length:

```
let array = [1, 2, 3, 4, 5];
console.log(array.length); // выводит 5
```

Массивы

1. Методы массива: Массивы в JavaScript имеют множество полезных методов для манипулирования данными. Например, push() для добавления элемента в конец массива, pop() для удаления последнего элемента, shift() для удаления первого элемента, unshift() для добавления элемента в начало массива, splice() для добавления, удаления или замены элементов и многие другие.
2. Массивы могут содержать разные типы данных: Массивы в JavaScript могут хранить любые типы данных, включая строки, числа, булевы значения, объекты и даже другие массивы.
3. Массивы являются объектами: Несмотря на то, что мы называем их "массивами", в JavaScript массивы являются на самом деле объектами с некоторыми дополнительными возможностями.

Thank you!