Курс: Javascript. Событийно-ориентированное программирование

Тема 5. Массивы.

Теоретические сведения

Строки и массивы имеют некоторые совпадающие характеристики и множество различающих. Строка представляет собой последовательность символов, а массив — последовательность значений произвольного типа. И строки, и массивы позволяют обращаться к своим частям с помощью оператора индексации, индексы отсчитываются от 0. Строки и массивы имеют свойство length, содержащее длину строки или количество элементов массива. Литерал строки ограничивается кавычками, а литерал массива — квадратными скобками.

```
const first = 'Elias';
const last = 'Goss';
const person = [first, last];
console.log(person[0]); // Elias
console.log(first[0]); // E
console.log(person.length); // 2
console.log(first.length); // 5
```

Индексируемые структуры — традиционный материал для обработки с помощью циклов со счётчиком:

И строки, и массивы, в отличие от объектов, реализуют протокол итерации (перебираемости).

Это значит, что их значения можно перебрать, не обращаясь к индексам, с помощью инструкции for .. of:

```
const first = 'Elias';
const last = 'Goss';
for (const t of first) console.log(t);
for (const t of last) console.log(t);
```

Вместо императивных структур где возможно удобно пользоваться функциональными:

```
const person = [first, last];
person.forEach((x, i) => console.log(`${i} ${x}`));
// 0 Elias
// 1 Goss
```

В отличие от строк, массивы наследуют метод forEach, который принимает коллбэк, вызываемый для каждого элемента массива. Этот метод возвращает undefined, не

поддерживает таким образом цепочку методов и не мутирует (сам по себе) перебираемый массив.
В отличие от иммутабельных строк (нельзя изменить какую-то часть строки, как бы подставив вместо одного символа другой, можно только создать новую строку по исходной), массивы, естественно, поддерживают возможность изменять себя по частям. В ряде случаев это как раз нежелательно, поэтому нужно обращать внимание на то, как работает тот или иной метод.

В отличие от строк, являющихся примитивным типом данных, массивы относятся к ссылочному типу. Массивы являются подвидом объектов, о которых в соответствующей теме курса.

Какова связь ссылочных типов данных и так называемой реактивности?

```
1   let arrayOne = [1, 2];
2   // создаём синоним:
3   const refToArrayOne = arrayOne;
4   console.log(refToArrayOne[0]); // 1
5   console.log(arrayOne[0]); // 1
6   arrayOne[0] = 3;
7   console.log(refToArrayOne[0]); // 3
8   console.log(arrayOne[0]); // 3
9   // разрываем синонимичность:
10  arrayOne = [11, 12];
11  console.log(refToArrayOne[0]); // 3
12  console.log(arrayOne[0]); // 11
```

Ссылочность

Начиная со строки 3 и далее до строки 10 имена arrayOne и refToArrayOne ассоциированы с одной и той же областью памяти. Поэтому когда мы изменяем содержимое этой области в строке 6, имя refToArrayOne отражает это изменение.
Но в строке 10 связь имён arrayOne и refToArrayOne нарушилась, т.к. правостороннее выражение [11, 12] создаёт новую область памяти и связывает имя arrayOne с ней. А имя refToArrayOne остаётся связанным со старой областью памяти. Если бы имя refToArrayOne не хранило ссылку на эту область, то после строки 10 у интерпретатора появился бы повод передать массив [1, 2] в руки коллектора мусора.
Если бы в строке 3 создавалась «реактивная» переменная, то она была бы не синонимом имени для той же области памяти, а как бы полным дубликатом, повторителем оригинальной переменной (переменной-источника) и изменялась бы вместе с ней. Можно сказать и иначе: синонимы статичны, они привязывается к области памяти в момент присвоения, а реактивные переменные динамичны, они привязываются не к области памяти, а к связи оригинальной переменной и привязанной к ней области памяти.

Свойство length у массивов является динамическим, оно может быть изменено и оказывает влияние на структуру массива. Оно содержит значение, большее, чем самый большой индекс.

const arr = ['a', 'b', 'c'];

```
console.log(arr.length); // 3
arr.length = 4;
console.log(arr); // [ 'a', 'b', 'c', <1 empty item> ]
console.log(arr[3]); // undefined
console.log(arr[33]); // undefined
```

Разрежённым (sparse) называется массив, в котором есть «пустые» ячейки. Такое достигается применением delete к элементу массива:

```
const arr = ['a', 'b', 'c', 'd'];
delete arr[3];
console.log(arr); //  'a', 'b', 'c', <1 empty item> ]
```

или созданием массива с пропусками:

```
const arr = ['a', 'b', 'c', ,];
Unexpected comma in middle of array. (no-sparse-arrays)
console.log(arr); //  'a', 'b', 'c', <1 empty item> ]
```

Разреженный массив

Линтер в примере выше выражает недовольство наличием разрежённого массива. Обратите внимание на две запятые в конце. Именно отсутствие чего бы то ни было между ними создаёт эффект дыры в массиве.

Кстати, интересно, что, если оставить в конце одну запятую, линтер недоволен как наличием пробела в конце, так и отсутствием:

```
const arr = ['a', 'b', 'c', ];
There should be no space before ']'. (array-bracket-spacing)
console.log(arr); //  [ 'a', 'b', 'c' ]
const arr = ['a', 'b', 'c',];
A space is required after ','. (comma-spacing)
console.log(arr); //  [ 'a', 'b', 'c' ]
```

Проблема пробела в синтаксисе литерала массива

Уменьшение свойства length вызывает усечение массива.

```
const arr = ['a', 'b', 'c'];
console.log(arr.length); // 3
arr.length = 2;
console.log(arr); // [ 'a', 'b']
```

Если мы присвоим значение

```
const arr = ['a', 'b', 'c'];
console.log(arr.length); // 3
arr[5] = 'λ';
console.log(arr); // [ 'a', 'b', 'c', <2 empty items>, 'λ' ]
console.log(arr.length); // 6
```

Задачи с массивами

Конвертация из массива в строку и обратно может быть осуществлена с помощью взаимно-обратных методов join и split.

Если поиск в строке очень удобно осуществлять с помощью регулярных выражений, то поиск в массиве благодаря es2015 стал удобнее с помощью методов find и findIndex. Ранее многие задачи было удобнее решать с помощью аналогичных методов сторонних библиотек, таких как lodash.

До ES6 существовало два основных способа создания массивов:

- литералы;
- конструктор Array.

Оба подхода требовали перечислить элементы будущего массива по отдельности. Чтобы упростить создание массивов JavaScript, в ECMAScript 6 были добавлены методы Array.of() и Array.from().

Упражнение

Используя метод Array.from и одну лямбду, сгенерируйте массив возрастающих значений от 1 до n

Решение

```
Array.from({ length: 5 }, (v, i) => ++i)
```

Также ES6 добавляет новые методы в каждый массив (т.е. в Array.prototype).

- find и findIndex (для работы над массивами с любыми значениями);
- fill и copyWithin (по аналогии с типизированными массивами).

Ещё ES6 добавил метод String.prototype.includes для проверки вхождения подстроки в строку. Чтобы обеспечить единообразие функциональных возможностей строк и массивов, метод Array.prototype.includes был добавлен в редакции ES7 (ECMAScript 2016).

Иными словами, после ES5 в 2015 и 2016 г. работа с массивами в JavaScript обогатилась пятью методами.

В ES2015 также введены типизированные массивы. Обычно принцип инкапсуляции при реализации типов данных не позволяет обращаться к отдельным частям значения данного типа. Например, мы не можем обратиться отдельно к 3-му байту из 8 в значении типа Double (IEEE754). Типизированные массивы предоставляют такую возможность. Наряду с собственно типизированными массивами в клиентский JavaScript привносится понятие буфера, которое ранее было введено в платформе Node.

С помощью методов map и reduce производится мэппинг и свёртка массива. Метод map создаёт новый массив, сопоставляя с помощью функции элементам исходного массива элементы нового массива:

```
const exps = [0, 1, 2, 3];
const degrs = exps.map(x => x ** 2);
console.log(degrs[3]); // 9
```

В этом примере каждому элементу исходного массива сопоставляется его квадрат в новом массиве.

Метод reduce помещает первые два значения массива в аргументы лямбды (здесь x, y), производит указанное к теле лямбды действие, и помещает в x результат, а в y третий элемент массива и так далее до исчерпания массива, тем самым сводя (редуцируя) массив до единственного оставшегося значения.
x + y
1 + 2 = 3

x + y
3 + 4 = 7

и т.п.

```
const nums = Array.from({ length: 100 }, (v, i) => ++i);
const sum = nums.reduce((x, y) => x + y);
console.log(sum); // 5050
```

В методе filter лямбда служит предикатом, позволяющим «отсеять» часть исходных значений в новый массив: если она возвращает истину, то значение будет передано в новый массив. В методе sort аналогичным образом решается вопрос о взаимном расположении элементов с целью их переупорядочения.