В ЭТОЙ ГЛАВЕ:

- используем массивы для обработки списков данных;
- научимся выполнять распространенные задачи с помощью различных свойств массива.

13

МАССИВЫ

Давайте представим, что вы хотите составить список на листке бумаги. Назовем его *продукты*. Теперь запишите в нем пронумерованный список, начинающийся с нуля, и перечислите все, что вам нужно (рис. 13.1).

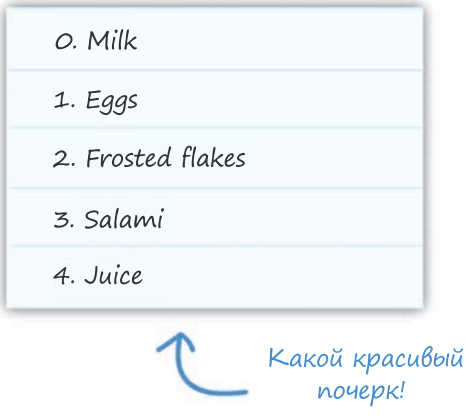

РИС. 13.1.
Список продуктов

Написав простой список, вы получили пример массива из реальной жизни. Листок бумаги, проименованный как *продукты*, это и есть ваш массив. Предметы же, которые вы хотите купить, — это значения массива.

В этом уроке вы не только узнаете, какие продукты я предпочитаю покупать, но и познакомитесь с очень распространенным типом — *массивом*.

Поехали!

Создание массива

Сейчас для создания массивов крутые чуваки используют открывающиеся и закрывающиеся квадратные скобки. Ниже приведена переменная `groceries` (продукты), инициализированная как пустой массив:

```
let groceries = [];
```

Такой скобочный способ создания массива больше известен как *литеральная нотация массива*.

Как правило, вы будете создавать массив, изначально содержащий определенные элементы. Для этого просто поместите нужные элементы в скобки и разделите их запятыми:

```
let groceries = ["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
```

Обратите внимание, что теперь массив содержит `Milk` (молоко), `Eggs` (яйца), `Frosted Flakes` (глазированные хлопья), `Salami` (салями) и `Juice` (сок). Считаю необходимым напомнить о важности запятых.

Теперь, когда вы научились объявлять массив, давайте взглянем на то, как его можно использовать для хранения данных и работы с ними.

Обращение к значениям массива

Одна из прелестей массивов в том, что вы имеете легкий доступ не только к ним самим, но и к их значениям, аналогично выделению одного из продуктов в вашем списке (рис. 13.2).

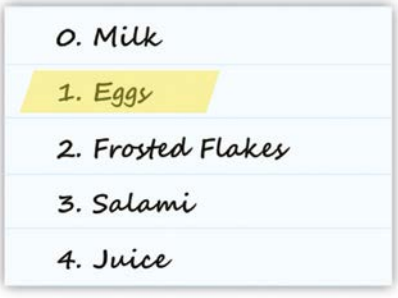

РИС. 13.2.
Массивы позволяют выборочно обращаться к отдельным элементам

Для этого вам достаточно знать простую процедуру обращения к отдельному элементу.

Внутри массива каждому элементу присвоен номер, начиная с нуля. На рис. 13.2 `Milk` имеет значение `0`, `Eggs` — `1`, `Frosted Flakes` соответствует значение `2` и т. д. Формально эти номера называются значением индекса (индексами).

В данном случае наш массив `groceries` объявлен следующим образом:

```
let groceries = ["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
```

Если мне понадобится обратиться к одному из элементов, то все, что потребуется, — это передать значение его индекса:

```
groceries[1]
```

Значение индекса передается массиву внутри квадратных скобок. В текущем примере мы обращаемся к значению `Eggs`, так как именно этому элементу соответствует позиция индекса `1`. Если передать `2`, то вернется `Frosted Flakes`. Вы можете продолжать передавать значения индекса, пока они не закончатся.

Диапазон чисел, которые вы можете использовать в качестве значений индекса, на одно меньше, чем длина самого массива. Причина в том, что индексы начинаются с `0`. Если в массиве есть пять элементов, то попытка отобразить `grocery[6]` или `grocery[5]` приведет к появлению сообщения `undefined`.

Идем дальше. В большинстве реальных сценариев вам понадобится программно перебирать весь массив вместо обращения к каждому элементу отдельно.

Для осуществления этого вы можете использовать цикл `for`:

```
for (let i = 0; i < groceries.length; i++) {
  let item = groceries[i];
}
```

Помните, что диапазон цикла начинается с `0` и заканчивается на одно значение раньше полной длины массива (возвращаемой как свойство `length`). Все работает именно так по уже описанной мной причине — значения индекса начинаются с `0` и заканчиваются на одно значение раньше, чем возвращаемая длина массива. При этом свойство `length` возвращает точное число элементов.

Добавление элементов

Ваши массивы будут редко сохранять свое изначальное состояние, так как вы, скорее всего, будете добавлять в них элементы. Для этого используется метод `push`:

```
groceries.push("Cookies");
```

Метод `push` вызывается непосредственно для массива, при этом в него передаются добавляемые данные. В итоге вновь добавленные элементы всегда оказываются в конце массива.

Например, если выполнить этот код для изначального массива, вы увидите, что элемент `Cookies` (печенье) добавлен в его конец (рис. 13.3).

РИС. 13.3.
Теперь массив расширен добавленным в конец элементом Cookies

Если же вы хотите добавить данные в начало, используйте метод `unshift`:

```
groceries.unshift("Bananas");
```

При добавлении данных в начало массива значение индекса каждого из существующих в нем элементов увеличивается с учетом вновь появившихся данных (рис. 13.4).

РИС. 13.4.
Только что добавленный элемент вставлен в начало

Причина в том, что первый элемент массива всегда будет иметь значение индекса `0`. Поэтому элемент, изначально занимающий позицию значения `0`, вынужденно смещается, смещая и все следующие за ним элементы, освобождая тем самым место для добавляемых данных.

При использовании методы `push` и `unshift` помимо добавления элементов также возвращают новую длину массива:

```
console.log(groceries.push("Cookies")); // возвращает 6
```

Я не уверен, полезно ли это, но на всякий случай имейте это в виду.

Удаление элементов

Для удаления можно использовать методы `pop` и `shift`. `Pop` удаляет последний элемент и возвращает его:

```
let lastItem = groceries.pop();
```

Метод `shift` делает то же самое, но с обратной стороны массива, то есть вместо удаления и возвращения последнего элемента он проделывает это с первым:

```
let firstItem = groceries.shift();
```

При удалении элемента из начала массива позиции индексов остальных уменьшаются на `1`, заполняя тем самым появившийся пропуск (рис. 13.5).

РИС. 13.5.
Что происходит при удалении элементов из массива

Обратите внимание, что при добавлении элементов с помощью `unshift` или `push` значение, возвращаемое при вызове этих методов, является новой длиной массива. Но при использовании методов `pop` или `shift` происходит не то же самое. В данном случае при удалении элементов значение, возвращаемое при вызове метода, является самим удаляемым элементом.

Поиск элементов в массиве

Для поиска элементов внутри массива существует несколько методов: `indexOf`, `lastIndexOf`, `includes`, `find`, `findIndex` и `filter`. Во избежание усложнения мы пока что сконцентрируемся на `indexOf` и `lastIndexOf`.

Работа этих двух индексов заключается в сканировании массива и возвращении индекса совпадающего элемента.

Метод `indexOf` возвращает первый найденный индекс искомого элемента:

```
let groceries =["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
let resultIndex = groceries.indexOf("Eggs",0);

console.log(resultIndex); // 1
```

Обратите внимание, что переменная `resultIndex` содержит результат вызова `indexOf` для массива `groceries`. Для использования `indexOf` я передаю ему искомый элемент вместе с индексом, с которого следует начать:

```
groceries.indexOf("Eggs", 0);
```

В данном случае `indexOf` вернет значение `1`.

Метод `lastIndexOf` похож на `indexOf` в использовании, но отличается тем, *что* возвращает при обнаружении элемента. Если `indexOf` находит первый индекс искомого элемента, то `lastIndexOf` находит и возвращает последний индекс этого элемента.

Если же искомый элемент в массиве отсутствует, оба этих метода возвращают `-1`.

Слияние массивов

Последнее, что мы рассмотрим, — это слияние двух раздельных массивов для создания нового. Предположим, у вас есть два массива `good` (хорошие) и `bad` (плохие):

```
let good = ["Mario", "Luigi", "Kirby", "Yoshi"];
let bad = ["Bowser", "Koopa Troopa", "Goomba"];
```

Чтобы совместить их, используйте метод `concat` для массива, который вы хотите расширить, и передайте в него второй массив в виде аргумента. В итоге будет возвращен новый массив, содержащий и `good`, и `bad`:

```
let goodAndBad = good.concat(bad);
console.log(goodAndBad);
```