Булевы значения

Теперь поговорим о булевых значениях. В сущности, есть лишь два варианта таких значений — это либо true (истина), либо false (ложь). Например, вот простое выражение с булевым значением:

```
var javascriptIsCool = true;
javascriptIsCool;
true
```

Здесь мы создали новую переменную с именем javascriptIsCool и присвоили ей булево значение true. Следующей строкой мы запросили содержимое javascriptIsCool и, разумеется, получили true.

Логические операции

Подобно тому как числа можно объединять с помощью математических операторов (+, —, * / и других), булевы значения можно объединять посредством булевых (логических) операторов. Результатом выражения, составленного из булевых значений и булевых операторов, всегда будет другое булево значение (либо true, либо false).

Три основных булевых оператора — это &&, || и !. Давайте познакомимся с ними поближе.

&&(И)

Оператор && означает «и». Вслух его называют «и», либо «и-и», либо «амперсанд-амперсанд» (амперсандом называется символ &). Используйте оператор && с двумя булевыми значениями, когда нужно узнать, равны ли они оба true.

Например, перед тем как пойти на занятия, вы хотите убедиться, что приняли душ, а также взяли рюкзак. Если оба эти условия истинны (true), можно идти на учёбу, но если хоть одно ложно (false), вы еще не готовы.

```
var hadShower = true;
var hasBackpack = false;
hadShower && hasBackpack;
false
```

Здесь мы устанавливаем переменную hadShower («вы приняли душ?») в true, а переменную hasBackpack («вы взяли рюкзак?») в false. Далее, вводя hadShower && hasBackpack, мы спрашиваем JavaScript: «равны ли оба этих значения true»? Поскольку это не так (рюкзак не в руках), JavaScript возвращает false (то есть вы не готовы идти на занятия).

Давайте повторим попытку, установив на этот раз обе переменные в true:

```
var hadShower = true;
var hasBackpack = true;
hadShower && hasBackpack;
true
```

Теперь JavaScript сообщает нам, что hadShower && hasBackpack равняется true. Ура! Можно идти на занятия!

II (ИЛИ)

Булев оператор || означает «или». Так его и следует называть — «или», или даже «или-или», хотя некоторые называют его «пайпс», поскольку среди англоязычных программистов символ | зовется «пайп» («труба»). Используйте оператор || с двумя булевыми значениями для проверки, что как минимум одно из них равняется true.

Предположим, вы снова готовитесь идти на занятия и хотите взять с собой к обеду фрукты, причем вам неважно, будет это яблоко, или апельсин, или и то и другое. С помощью JavaScript можно проверить, есть ли у вас хотя бы один из этих плодов:

```
var hasApple = true;
var hasOrange = false;
hasApple || hasOrange;
true
```

Выражение hasApple || hasOrange даст true, если либо hasApple («взяли яблоко?»), либо hasOrange («взяли апельсин?»), либо обе эти переменные имеют значение true. Однако если обе они равны false, выражение даст false (то есть у вас с собой нет ни одного фрукта).

!(НЕ)

Оператор ! означает «не» — так его и называйте. Используйте этот оператор, чтобы превратить false в true или, наоборот, true в false. Это полезно для работы со значениями-противоположностями. Например:

```
var isWeekend = true;
var needToShowerToday = !isWeekend;
needToShowerToday;
false
```

В этом примере мы установили переменную isWeekend («сейчас выходной?») в true. Затем мы дали переменной needToShowerToday («сегодня нужно принять душ?») значение !isWeekend. Оператор ! преобразует значение в противоположное — то есть, если isWeekend равно true, !isWeekend даст нам не true (то есть false). Соответственно, запрашивая значение needToShowerToday, мы получаем false (сегодня выходной, так что мыться совсем не обязательно).

Поскольку needToShowerToday равно false, !needToShowerToday даст true:

```
needToShowerToday;
false
!needToShowerToday;
true
```

Совмещение логических операторов

Операторы дают больше возможностей, если использовать их совместно. Допустим, вам нужно идти на занятия, если сегодня не выходной, и вы приняли душ, и у вас с собой есть яблоко или апельсин. Вот как с помощью JavaScript проверить, выполняются ли все эти условия:

```
var isWeekend = false;
var hadShower = true;
var hasApple = false;
var hasOrange = true;
var shouldGoToUniversity = !isWeekend && hadShower && (hasApple || hasOrange);
shouldGoToUniversity;
true
```

В данном случае сегодня не выходной, вы приняли душ, у вас нет с собой яблока, зато есть апельсин — значит, нужно идти на занятия.

Выражение hasApple || hasOrange записано в скобках, поскольку нам важно убедиться, что эта проверка выполнена в первую очередь. Точно так же как JavaScript выполняет умножение прежде сложения, в логических выражениях он выполняет && прежде ||.

Сравнение чисел с помощью булевых значений

Булевы значения можно использовать для проверки чисел, если эта проверка подразумевает простой ответ: да или нет. Например, представьте, что вы работаете в парке развлечений, где один из аттракционов имеет ограничение: туда допускаются посетители ростом не менее 150 см (иначе они могут вывалиться из кабинки!). Когда кто-нибудь хочет прокатиться, он сообщает свой рост, и вам нужно понять, больше названное число или меньше.

Больше

Чтобы узнать, больше ли одно число, чем другое, нужно использовать оператор «больше» (>). Например, для проверки, что рост посетителя (155 см) больше, чем ограничение по росту (150 см), мы можем задать переменной height (рост посетителя) значение 155, а переменной heightRestriction (ограничение по росту) значение 150, а затем использовать оператор > для сравнения двух переменных:

```
var height = 155;
var heightRestriction = 150;
height > heightRestriction;
true
```

Введя height > heightRestriction, мы просим JavaScript показать нам, больше ли первое значение, чем второе, или нет. В данном случае посетитель достаточно высок!

Но что если рост посетителя в точности равен 150 см?

```
var height = 150;
var heightRestriction = 150;
height > heightRestriction;
false
```

Нет, посетитель недостаточно высок! Хотя если ограничение по росту — 150 см, наверное, стоит пускать и тех, чей рост в точности равен 150 см? Это нужно исправить. К счастью, в JavaScript есть еще один оператор, >=, что означает «больше или равно».

```
var height = 150;
var heightRestriction = 150;
height >= heightRestriction;
true
```

Ну вот, теперь лучше — 150 удовлетворяет условию «больше или равно 150».

Меньше

Оператор, противоположный «больше» (>), зовется оператором «меньше» (<). Он пригодится, если аттракцион предназначен только для маленьких детей. Например, пусть рост посетителя равен 150 см, но по правилам аттракциона на него допускаются посетители ростом не более 120 см:

```
var height = 150;
var heightRestriction = 120;
height < heightRestriction;
false
```

Мы хотим убедиться, что рост посетителя меньше ограничения, и поэтому используем <. Поскольку 150 не меньше 120, ответом будет false (человек ростом 150 см слишком высок для этого аттракциона).

И, как вы, наверное, уже догадались, есть оператор <=, что означает «меньше или равно».

```
var height = 120;
var heightRestriction = 120;
height <= heightRestriction;
true
```

Посетителю, рост которого равен 120 см, вход все еще разрешен.

Равно

Чтобы проверить два числа на точное равенство, используйте тройной знак равенства (===) — это оператор «равно». Будьте осторожны, не путайте === с одиночным знаком равенства (=), поскольку === означает «равны ли эти два числа?», а = означает «положить значение справа в переменную слева».

Иначе говоря, === задает вопрос, а = присваивает переменной значение.

При использовании = имя переменной должно стоять слева, а значение, которое вы хотите в эту переменную положить, справа. Однако === служит лишь для проверки двух значений на равенство, поэтому неважно, какое значение с какой стороны стоит.

Представьте, что вы загадали своим друзьям Диме, Стасу, и Азату число, а именно число 5. Вы облегчили им задачу, сообщив, что это число от 1 до 9, и ваши друзья начали угадывать. Сначала присвоим переменной mySecretNumber значение 5. Первый из играющих, Дима, загадывает ответ 3, который мы кладем в переменную dimaGuess. Поглядим, что будет дальше:

```
var mySecretNumber = 5;
var dimaGuess = 3;
mySecretNumber === dimaGuess;
false
var stasGuess = 7;
mySecretNumber === stasGuess;
false
var azatGuess = 5;
mySecretNumber === azatGuess;
true
```

Число, которое вы загадали, находится в переменной mySecretNumber. Переменные dimaGuess, stasGuess и azatGuess соответствуют предположениям ваших друзей. Далее с помощью оператора === можно проверить, равен ли какой-нибудь ответ вашему числу. Третий друг, Азат, назвал 5 и победил.

Сравнивая два числа с помощью ===, вы получаете true, только когда оба числа совпадают. Поскольку в azatGuess находится значение 5, а mySecretNumber также равно 5, выражение mySecretNumber === azatGuess вернет true. Другие варианты ответа не совпадают с mySecretNumber, поэтому сравнение с ними даст false.

Также с помощью === можно сравнить две строки или два булевых значения. Если же сравнивать так значения **разных** типов, ответом всегда будет **false**.

Двойной знак равенства

Еще немного запутаю вас: в JavaScript есть еще один оператор сравнения (двойное равно, ==), который означает «практически равно». Используйте его для проверки двух значений на соответствие друг другу, даже если одно из них строка, а другое — число. Все значения принадлежат к тому или иному типу, так что число 5 отличается от строки «5», хоть они и выглядят похоже. Если сравнить их с помощью ===, JavaScript ответит, что значения не равны. Однако при сравнении через == они окажутся равными:

```
var stringNumber = "5";
var actualNumber = 5;
stringNumber === actualNumber;
false
stringNumber == actualNumber;
true
```

Возможно, тут вы подумаете: «Похоже, двойное равно удобнее, чем тройное!» Однако будьте очень осторожны: двойное равно может ввести вас в заблуждение. Например, как считаете, 0 равен false? А строка "false" значению false? При сравнении через двойное равно 0 оказывается равным false, а строка "false" не равна false:

```
0 == false;
true
"false" == false;
false
```

Дело в том, что, сравнивая значения через двойное равно, JavaScript первым делом пытается преобразовать их к одному типу. В данном случае булево значение он преобразует в числовое — при этом false становится нулем, а true — единицей. Поэтому, сравнивая 0 == false, вы получите true!

Из-за всех этих особенностей преобразования типов двойного равно лучше пока пользуйтесь только оператором ===.

undefined и null

И наконец, в JavaScript есть два особых значения, они называются undefined и null. Оба они означают «пусто», но смысл этого в обоих случаях немного различается.

JavaScript использует значение undefined, когда не может найти иного значения. Например, если, создав новую переменную, вы не присвоите ей значение с помощью оператора =, ее значением будет undefined:

```
var myVariable;
myVariable;
undefined
```

А значение null обычно используется, чтобы явно обозначить — «тут пусто».

```
var myNullVariable = null;
myNullVariable;
null
```

Пока вы будете нечасто использовать undefined и null. Вы получите undefined, если создадите переменную и не присвоите ей значения, — JavaScript всегда возвращает undefined, когда

значение не определено. Однако специально undefined обычно ничему не присваивают; если вам захочется обозначить, что в переменной «пусто», используйте для этого null.

Иначе говоря, null нужен, чтобы явно показать отсутствие значения, и порой это бывает полезно. Например, есть переменная, обозначающая ваш любимый овощ. Если вы терпеть не можете все без исключения овощи, имеет смысл дать переменной «любимый овощ» значение null.

Этим вы явно покажете любому, кто увидит ваш код, что у вас нет любимого овоща. Однако если в переменной будет undefined, кто-нибудь может подумать, что вы просто еще не приписали ей значения.

МАССИВЫ

Мы уже изучили числа и строки — типы данных, которые можно хранить и использовать в своих программах. Но одни лишь числа и строки — это как-то скучновато; не столь уж многое можно сделать со строкой как таковой. С помощью массивов JavaScript позволяет создавать и группировать данные более любопытными способами. А по сути своей массив — всего лишь список, где хранятся другие значения.

Например, если вашему другу интересно, какие три вида динозавров вам нравятся больше всего, вы можете создать массив и расположить там по порядку названия этих динозавров:

```
var myTopThreeDinosaurs = ["Тираннозавр", "Велоцираптор", "Стегозавр"];
```

Зачем нужны массивы?

Вернемся к нашим динозаврам. Положим, вы решили написать программу для учета всех видов динозавров, которые вам известны. Вы можете создать для каждого вида отдельную переменную:

```
var dinosaur1 = "Тираннозавр";
var dinosaur2 = "Велоцираптор";
var dinosaur3 = "Стегозавр";
var dinosaur4 = "Трицератопс";
var dinosaur5 = "Брахиозавр";
var dinosaur6 = "Птеранодон";
var dinosaur7 = "Апатозавр";
var dinosaur8 = "Диплодок";
var dinosaur9 = "Компсогнат";
```

Однако пользоваться этим списком не слишком удобно — у вас есть девять переменных там, где можно обойтись лишь одной. А теперь представьте, что динозавров в программе не девять, а 1000! Пришлось бы создать 1000 отдельных переменных, работать с которыми было бы решительно невозможно.

Это похоже на список покупок, составленный так, что каждая покупка указана на отдельном листе бумаги. На одном листке написано «яйца», на другом — «хлеб», на следующем — «апельсины». Большинство людей предпочли бы видеть весь список на одном листе бумаги. Так не проще ли сгруппировать всех динозавров в один список? Вот для этого и нужны массивы.

Создание массива

Чтобы создать массив, используйте квадратные скобки []. Фактически для задания пустого массива достаточно лишь пары квадратных скобок:

```
[];
[]
```

Но кому нужен пустой массив? Давайте-ка заполним его динозаврами!

Чтобы создать массив со значениями, нужно перечислить эти значения внутри квадратных скобок, разделяя их запятыми. Отдельные значения, хранящиеся в массиве, называют элементами. В данном примере все элементы будут строковыми (это названия любимых динозавров), поэтому запишем их в кавычках. Сохраним наш массив в переменной с именем dinosaurs:

```
var dinosaurs = ["Тираннозавр", "Велоцираптор", "Стегозавр",
                 "Трицератопс", "Брахиозавр", "Птеранодон",
                 "Апатозавр", "Диплодок", "Компсогнат"];
```

Длинный список сложно читать, когда он записан одной строкой, но, к счастью, это не единственный способ форматирования кода при создании массива. Вы можете поставить открывающую квадратную скобку на одной строке, каждый элемент писать с новой строки и последней строкой поставить закрывающую квадратную скобку:

```
var dinosaurs = [
"Тираннозавр",
"Велоцираптор",
"Стегозавр",
"Трицератопс",
"Брахиозавр",
"Птеранодон",
"Апатозавр",
"Диплодок",
"Компсогнат"
];
```

Чтобы ввести такой код в консоли, вам придется нажимать одновременно с ENTER клавишу SHIFT каждый раз, когда нужно перейти к новой строке. Иначе JavaScript попытается выполнить то, что вы уже ввели, даже если команда еще не завершена. Поэтому при работе в консоли проще записывать массивы одной строкой.

Для JavaScript неважно, отформатируете вы код, расположив весь массив на одной строке или на нескольких строках по частям. Сколько бы ни стояло переносов, JavaScript увидит один и тот же массив — в нашем случае состоящий из девяти строк.

Доступ к элементам массива

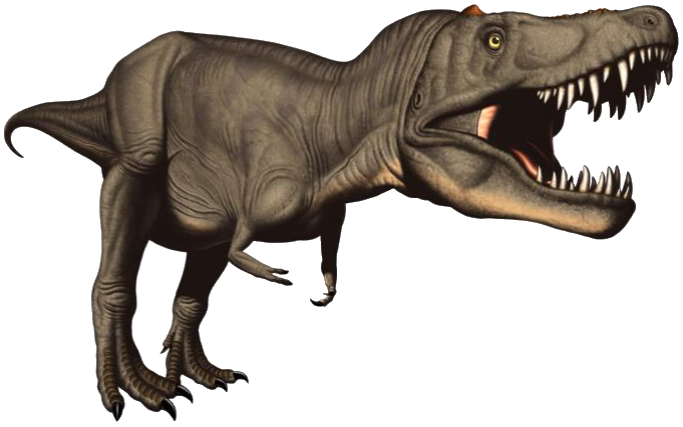
Чтобы получить доступ к элементам массива, используйте квадратные скобки с индексом нужного вам элемента, как в этом примере:

```
dinosaurs[0];
"Тираннозавр"
dinosaurs[8];
"Компсогнат"
```

Индекс — это номер элемента, в котором хранится значение. Аналогично символам в строке, первому элементу массива соответствует индекс 0, второму — 1, третьему — 2 и т. д. Поэтому, запросив индекс 0 в массиве dinosaurs, мы получили "Тираннозавр" (это первый элемент), а запросив индекс 8 — "Компсогнат" (девятый элемент).

Возможность доступа к отдельным элементам массива очень полезна. Например, если вы хотите показать кому-то самого-самого любимого своего динозавра, ни к чему показывать весь массив. Вместо этого просто возьмите первый элемент:

```
dinosaurs[0];
"Тираннозавр"
```



Создание и изменение элементов

Используя индекс в квадратных скобках, можно задавать или изменять значения элементов и даже добавлять новые элементы. Например, чтобы заменить содержимое первого элемента массива dinosaurs ("Тираннозавр") на "Тираннозавр рекс", можно написать:

```
dinosaurs[0] = "Тираннозавр рекс";
```

После этого массив dinosaurs станет таким:

```
["Тираннозавр рекс", "Велоцираптор", "Стегозавр", "Трицератопс", "Брахиозавр",
"Птеранодон", "Апатозавр", "Диплодок", "Компсогнат"]
```

С помощью индексов также можно добавлять в массив элементы. Например, вот как создать массив dinosaurs, задавая каждый элемент через квадратные скобки:

```
var dinosaurs = [];
dinosaurs[0] = "Тираннозавр";
dinosaurs[1] = "Велоцираптор";
dinosaurs[2] = "Стегозавр";
dinosaurs[3] = "Трицератопс";
dinosaurs[4] = "Брахиозавр";
dinosaurs[5] = "Птеранодон";
dinosaurs[6] = "Апатозавр";
dinosaurs[7] = "Диплодок";
dinosaurs[8] = "Компсогнат";
dinosaurs;
["Тираннозавр", "Велоцираптор", "Стегозавр", "Трицератопс", "Брахиозавр", "Птеранодон",
"Апатозавр", "Диплодок", "Компсогнат"]
```

Сначала создаем пустой массив: var dinosaurs = []. Затем в каждой из следующих строк добавляем по одному элементу командами dinosaurs[] с индексом от 0 до 8. Закончив наполнение массива, можно посмотреть его содержимое (набрав dinosaurs;) и убедиться, что JavaScript расположил значения по порядку, в соответствии с индексами.

На самом деле в массив можно добавить элемент с любым индексом. Например, чтобы добавить нового (выдуманного) динозавра с индексом 33, введем:

```
dinosaurs[34] = "Уфазавр";
dinosaurs;
["Тираннозавр", "Велоцираптор", "Стегозавр", "Трицератопс", "Брахиозавр", "Птеранодон",
"Апатозавр", "Диплодок", "Компсогнат", undefined × 25, "Уфазавр"]
```

Элементы между индексами 8 и 34 получат значение undefined. При печати массива Chrome сообщает количество этих undefined-элементов, а не выводит каждый из них по отдельности.

Разные типы данных в одном массиве

Не обязательно, чтобы все элементы массива были одного типа. Например, вот массив, в котором хранится число (3), строка ("динозавры"), массив (["трицератопс", "стегозавр", 3627.5]) и еще одно число (10):

```
var dinosaursAndNumbers = [3, "динозавры", ["трицератопс", "стегозавр", 3627.5], 10];
```

Чтобы обратиться к элементам массива, вложенного в другой массив, нужно использовать вторую пару квадратных скобок. Например, если команда dinosaursAndNumbers[2]; вернет весь вложенный массив, то dinosaursAndNumbers[2][0]; — лишь первый элемент этого вложенного массива ("трицератопс").

```
dinosaursAndNumbers[2];
["трицератопс", "стегозавр", 3627.5]
dinosaursAndNumbers[2][0];
"трицератопс"
```

Вводя dinosaursAndNumbers[2][0] мы просим JavaScript обратиться к индексу 2 массива dinosaursAndNumbers, где находится массив ["трицератопс", "стегозавр", 3627.5], и вернуть значение с индексом 0 из этого вложенного массива — это первый элемент, "трицератопс".

Работаем с массивами

Работать с массивами вам помогут свойства и методы. Свойства хранят различные сведения о массиве, а методы обычно либо изменяют его, либо возвращают новый массив. Давайте разберемся.

Длина массива

Порой нужно знать, сколько в массиве элементов. Например, если снова и снова добавлять динозавров в массив dinosaurs, вы можете забыть, сколько их теперь всего.

Для этого есть свойство length (длина), хранящее количество элементов в массиве. Чтобы узнать длину массива, просто добавьте .length после его имени. Давайте посмотрим, как это работает. Но сначала создадим новый массив с тремя элементами:

```
var recidivists = ["Трус", "Балбес", "Бывалый"];
recidivists[0];
"Трус"
recidivists[1];
"Балбес"
recidivists[2];
"Бывалый"
```

Чтобы узнать длину этого массива, добавим .length к recidivists:

```
recidivists.length;
3
```

JavaScript сообщает, что в массиве 3 элемента, и мы знаем, что их индексы — 0, 1 и 2. Отсюда следует полезное наблюдение: последний индекс массива всегда на единицу меньше длины этого массива. Это значит, что есть простой способ получить последний элемент массива, какой бы ни была его длина:

```
recidivists[recidivists.length - 1];
"Бывалый"
```

Мы попросили JavaScript вернуть элемент из нашего массива, но вместо числового индекса ввели в квадратных скобках выражение: длина массива минус 1. JavaScript нашел свойство recidivists.length со значением 3, вычел 1, получив 2, и наконец вернул элемент с индексом 2 — это и есть последний элемент, "Бывалый".

Добавление элементов в массив

Чтобы добавить элемент к концу массива, можно воспользоваться методом push. Введите .push после имени массива, а после в круглых скобках укажите элемент, который нужно добавить:

```
var animals = [];
animals.push("Кот");
1
animals.push("Пёс");
2
animals.push("Лама");
3
animals;
["Кот", "Пёс", "Лама"]
animals.length;
3
```

Командой var animals = []; мы создали пустой массив animals, а затем методом push добавили туда элемент "Кот". Потом снова использовали push, добавив "Пес", а затем "Лама". Запросив теперь содержимое массива animals, мы видим, что "Кот", "Пес" и "Лама" стоят там в том же порядке, в каком мы их добавляли.

Запуск метода в программировании называется вызовом метода. При вызове метода push происходят две вещи. Во-первых, в массив добавляется элемент, указанный в скобках. Во-вторых, метод задает новую длину массива. Именно эти значения длины появляются в консоли после каждого вызова push.

Чтобы добавить элемент в начало массива, используйте метод .unshift(элемент):

```
animals;
```

```
["Кот", "Пёс", "Лама"]
animals[0];
"Кот"
animals.unshift("Мартышка");
4
animals;
["Мартышка", "Кот", "Пёс", "Лама"]
animals.unshift("Белый медведь");
5
animals;
["Белый медведь", "Мартышка", "Кот", "Пёс", "Лама"]
animals[0];
"Белый медведь"
animals[2];
"Кот"
```

Мы начали с массива, созданного раньше, — ["Кот", "Пес", "Лама"]. Затем добавили в его начало элементы "Мартышка" и "Белый медведь", отчего остальные элементы сдвинулись вперед — при каждом добавлении их индексы увеличивались на 1. В результате элемент "Кот", у которого раньше был индекс 0, оказался под индексом 2.

Как и push, метод unshift при каждом вызове задает новую длину массива.