Урок 9

Объекты

Объекты используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

Объект может быть создан с помощью фигурных скобок `{…}` с необязательным списком *свойств*. Свойство – это пара «ключ: значение», где **ключ** – это строка (также называемая «именем свойства»), а **значение** может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо.

Пустой объект («пустой ящик») можно создать с помощью такого синтаксиса:

```
let user = {};  // синтаксис "литерал объекта"
```

Свойства

Мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

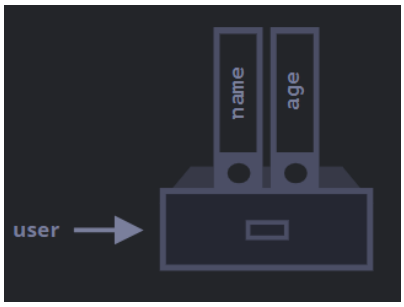
```
let user = {     // объект
    name: "John",  // под ключом "name" хранится значение "John"
    age: 30        // под ключом "age" хранится значение 30
  };
```

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие `":"`, и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте `user` сейчас находятся два свойства:

1. Первое свойство с именем `"name"` и значением `"John"`.
2. Второе свойство с именем `"age"` и значением `30`.

Можно сказать, что наш объект `user` – это ящик с двумя папками, подписанными «name» и «age».



Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

```
// получаем свойства объекта:
console.log( user.name ); // John
console.log( user.age ); // 30
```

Значение может быть любого типа. Давайте **добавим** свойство с логическим значением:

```
user.isAdmin = true;
```

Для **удаления** свойства мы можем использовать оператор `delete`:

```
delete user.age;
```

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

```
let user = {
    name: "John",
    age: 30,
    "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
  };
```

Последнее свойство объекта может заканчиваться запятой:

```
let user = {
    name: "John",
    age: 30,
  }
```

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

Объект, объявленный как константа, может быть изменён
Объект, объявленный через `const`, *может* быть изменён.

Например:

```
const user = {
    name: "John"
  };

  user.name = "Pete"; // (*)

  console.log(user.name); // Pete
```

Может показаться, что строка `(*)` должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление `const` защищает от изменений только саму переменную `user`, а не её содержимое.

Определение `const` выдаст ошибку только если мы присвоим переменной другое значение: `user=...`.

Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

```
// это вызовет синтаксическую ошибку
user.likes birds = true
```

JavaScript видит, что мы обращаемся к свойству `user.likes`, а затем идёт непонятное слово `birds`. В итоге синтаксическая ошибка.

Точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме `$` и `_`.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

```
let user = {};

// присваивание значения свойству
user["likes birds"] = true;

// получение значения свойства
console.log(user["likes birds"]); // true

// удаление свойства
delete user["likes birds"];
```

Сейчас всё в порядке. Обратите внимание, что строка в квадратных скобках заключена в кавычки (подойдёт любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

```
let key = "likes birds";

// то же самое, что и user["likes birds"] = true;
user[key] = true;
```

Здесь переменная `key` может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого мы используем её для доступа к свойству. Это даёт нам большую гибкость.

Пример:

```
let user = {
    name: "John",
    age: 30
  };

  let key = prompt("Что вы хотите узнать о пользователе?", "name");

  // доступ к свойству через переменную
  console.log( user[key] ); // John (если ввели "name")
```

Запись «через точку» такого не позволяет:

```
let user = {
    name: "John",
    age: 30
  };

  let key = "name";
  console.log( user.key ); // undefined
```

Цикл "for..in"

Для перебора всех свойств объекта используется цикл `for..in`. Этот цикл отличается от изученного ранее цикла `for(;;)`.

Синтаксис:

```
for (key in object) {
    // тело цикла выполняется для каждого свойства объекта
  }
```

К примеру, давайте выведем все свойства объекта `user`:

```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
  };

  for (let key in user) {
    // ключи
    console.log( key );  // name, age, isAdmin
    // значения ключей
    console.log( user[key] ); // John, 30, true
  }
```

Обратите внимание, что все конструкции «for» позволяют нам объявлять переменную внутри цикла, как, например, `let key` здесь.

Упорядочение свойств объекта

Свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. Разберёмся подробнее.

В качестве примера рассмотрим объект с телефонными кодами:

```
let codes = {
    "49": "Германия",
    "41": "Швейцария",
    "44": "Великобритания",
    // ..,
    "1": "США"
  };
```

```
for (let code in codes) {
    console.log(code); // 1, 41, 44, 49
  }
```

Если мы делаем сайт для немецкой аудитории, то, вероятно, мы хотим, чтобы код `49` был первым.

Но если мы запустим код, мы увидим совершенно другую картину:

- США (1) идёт первым
- затем Швейцария (41) и так далее.

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: `1, 41, 44, 49`.

Целочисленные свойства? Это что?
Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений.

То есть, `"49"` – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства `"+49"` или `"1.2"` таковыми не являются:

...С другой стороны, если ключи не целочисленные, то они перебираются в порядке создания, например:

```
let user = {
    name: "John",
    surname: "Smith"
  };
  user.age = 25; // добавим ещё одно свойство

// не целочисленные свойства перечислены в порядке создания
for (let prop in user) {
    console.log( prop ); // name, surname, age
  }
```

Таким образом, чтобы решить нашу проблему с телефонными кодами, мы можем схитрить, сделав коды не целочисленными свойствами. Добавления знака `"+"` перед каждым кодом будет достаточно.

Пример:

```
let codes = {
    "+49": "Германия",
    "+41": "Швейцария",
    "+44": "Великобритания",
    // ..,
```

```
  "+1": "США"
};

for (let code in codes) {
  console.log( +code ); // 49, 41, 44, 1
}
```

Теперь код работает так, как мы задумывали.

Итого

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

- Ключи свойств должны быть строками или символами (обычно строками).
- Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

- Запись через точку: `obj.property`.
- Квадратные скобки `obj["property"]`. Квадратные скобки позволяют взять ключ из переменной, например, `obj[varWithKey]`.

Дополнительные операторы:

- Удаление свойства: `delete obj.prop`.
- Проверка существования свойства: `"key" in obj`.
- Перебор свойств объекта: цикл for `for (let key in obj)`.

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это `console.log(message)`, `prompt(message, default)` и `alert(message)`. Но можно создавать и свои.

Объявление функции

Для создания функций мы можем использовать *объявление функции*.

Пример объявления функции:

```
function showMessage() {
    console.log( 'Всем привет!' );
  }
```

Вначале идёт ключевое слово `function`, после него *имя функции*, затем список *параметров* в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

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

Наша новая функция может быть вызвана по своему имени: `showMessage()`.

Например:

```
function showMessage() {
    console.log( 'Всем привет!' );
  }

  showMessage();
  showMessage();
```

Вызов `showMessage()` выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции.

Например:

```
function showMessage() {
    let message = "Привет, я JavaScript!"; // локальная переменная

    console.log( message );
  }

  showMessage(); // Привет, я JavaScript!

  console.log( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции
```

Внешние переменные

У функции есть доступ к внешним переменным, например:

```
let userName = 'Вася';

function showMessage() {
  let message = 'Привет, ' + userName;
  console.log(message);
}

showMessage(); // Привет, Вася
```

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Например:

```
let userName = 'Вася';

function showMessage() {
  userName = "Петя"; // (1) изменяем значение внешней переменной

  let message = 'Привет, ' + userName;
  console.log(message);
}
```

```
console.log( userName ); // Вася перед вызовом функции

showMessage();

console.log( userName ); // Петя, значение внешней переменной было изменено функцией
```

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Глобальные переменные
Переменные, объявленные снаружи всех функций, такие как внешняя переменная `userName` в вышеприведённом коде – называются *глобальными*.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры.

В нижеприведённом примере функции передаются два параметра: `from` и `text`.

```
function showMessage(from, text) { // параметры: from, text
    console.log(from + ': ' + text);
  }

  showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
  showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

Когда функция вызывается в строках `(*)` и `(**)`, переданные значения копируются в локальные переменные `from` и `text`. Затем они используются в теле функции.

Значение, передаваемое в качестве параметра функции, также называется *аргументом*.

Другими словами:

- Параметр – это переменная, указанная в круглых скобках в объявлении функции.
- Аргумент – это значение, которое передаётся функции при её вызове.

Мы объявляем функции со списком параметров, затем вызываем их, передавая аргументы.

Рассматривая приведённый выше пример, мы могли бы сказать:
"функция `showMessage` объявляется с двумя параметрами, затем вызывается с двумя аргументами.

Значения по умолчанию

Если при вызове функции аргумент не был указан, то его значением становится `undefined`.

Например, вышеупомянутая функция `showMessage(from, text)` может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведёт к ошибке. Такой вызов выведет `"*Аня*: undefined"`. В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`.

Если мы хотим задать параметру `text` значение по умолчанию, мы должны указать его после `=`:

```
function showMessage(from, text = "текст не добавлен") {
    console.log( from + ": " + text );
  }

  showMessage("Аня"); // Аня: текст не добавлен
```

Теперь, если параметр `text` не указан, его значением будет `"текст не добавлен"`

Возврат значения

Функция может вернуть результат, который будет передан в вызвавший её код.

Простейшим примером может служить функция сложения двух чисел:

```
function sum(a, b) {
    return a + b;
  }

  let result = sum(1, 2);
  console.log( result ); // 3
```

Директива `return` может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной `result` выше).

Вызовов `return` может быть несколько, например:

```
function checkAge(age) {
    if (age > 18) {
      return true;
    } else {
      return confirm('А родители разрешили?');
    }
  }

  let age = prompt('Сколько вам лет?', 18);

  if ( checkAge(age) ) {
    console.log( 'Доступ получен' );
  } else {
    console.log( 'Доступ закрыт' );
  }
```

Никогда не добавляйте перевод строки между `return` и его значением
Для длинного выражения в `return` может быть заманчиво разместить его на нескольких отдельных строках, например так:

```
return
(some + long + expression + or + whatever * f(a) + f(b))
```

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после `return`. Для него это будет выглядеть так:

```
return;
 (some + long + expression + or + whatever * f(a) + f(b))
```

Таким образом, это фактически стало пустым `return`.

Если мы хотим, чтобы возвращаемое выражение занимало несколько строк, нужно начать его на той же строке, что и `return`. Или, хотя бы, поставить там открывающую скобку, вот так:

```
return (
    some + long + expression
    + or +
    whatever * f(a) + f(b)
    )
```

И тогда всё сработает, как задумано.

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть кратким, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с `"show"` обычно что-то показывают.

Функции, начинающиеся с...

- `"get…"` – возвращают значение,
- `"calc…"` – что-то вычисляют,
- `"create…"` – что-то создают,
- `"check…"` – что-то проверяют и возвращают логическое значение, и т.д.

Примеры таких имён:

```
showMessage(..)     // показывает сообщение
getAge(..)          // возвращает возраст (получая его каким-то образом)
calcSum(..)         // вычисляет сумму и возвращает результат
createForm(..)      // создаёт форму (и обычно возвращает её)
checkPermission(..) // проверяет доступ, возвращая true/false
```

Благодаря префиксам, при первом взгляде на имя функции становится понятным, что делает её код, и какое значение она может возвращать.

Одна функция – одно действие
Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием.

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Несколько примеров, которые нарушают это правило:

- `getAge` – будет плохим выбором, если функция будет выводить `console.log` с возрастом (должна только возвращать его).
- `createForm` – будет плохим выбором, если функция будет изменять документ, добавляя форму в него (должна только создавать форму и возвращать её).

- `checkPermission` – будет плохим выбором, если функция будет отображать сообщение с текстом `доступ разрешён/запрещён` (должна только выполнять проверку и возвращать её результат).

В этих примерах использовались общепринятые смыслы префиксов. Конечно, вы в команде можете договориться о других значениях, но обычно они мало отличаются от общепринятых. В любом случае вы и ваша команда должны чётко понимать, что значит префикс, что функция с ним может делать, а чего не может.

Итого

Объявление функции имеет вид:

```
function имя(параметры, через, запятую) {
    /* тело, код функции */
  }
```

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
- Функция может возвращать значение. Если этого не происходит, тогда результат равен `undefined`.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
- Функция – это действие, поэтому её имя обычно является глаголом.
- Есть много общепринятых префиксов, таких как: `create…`, `show…`, `get…`, `check…` и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.