> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Работа с объектами

JavaScript спроектирован на основе простой парадигмы. В основе концепции лежат простые объекты. Объект — это набор свойств, и каждое свойство состоит из имени и значения, ассоциированного с этим именем. Значением свойства может быть функция, которую можно назвать *методом* объекта. В дополнение к встроенным в браузер объектам, вы можете определить свои собственные объекты. Эта глава описывает как пользоваться объектами, свойствами, функциями и методами, а также как создавать свои собственные объекты.

Обзор объектов

Объекты в JavaScript, как и во многих других языках программирования, похожи на объекты реальной жизни. Концепцию объектов JavaScript легче понять, проводя параллели с реально существующими в жизни объектами.

В JavaScript объект — это самостоятельная единица, имеющая свойства и определённый тип. Сравним, например, с чашкой. У чашки есть цвет, форма, вес, материал, из которого она сделана, и т.д. Точно так же, объекты JavaScript имеют свойства, которые определяют их характеристики.

Объекты и свойства

В JavaScript объект имеет свойства, ассоциированные с ним. Свойство объекта можно понимать как переменную, закреплённую за объектом. Свойства объекта в сущности являются теми же самыми переменными JavaScript, за тем исключением,

что они закреплены за объектом. Свойства объекта определяют его характеристики. Получить доступ к свойству объекта можно с помощью точечной записи:

```
JS
objectName.propertyName;
```

Имя свойства

переменной) и имя свойства являются чувствительными к регистру. Вы можете определить свойство указав его значение. Например, давайте создадим объект `myCar` и определим его свойства `make`, `model`, и `year` следующим образом:

```
JS
var myCar = new Object();
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
```

Неопределённые свойства объекта являются `undefined` (а не `null`).

```
JS
myCar.color; // undefined
```

Свойства объектов JavaScript также могут быть доступны или заданы с использованием скобочной записи (более подробно см. property accessors). Объекты иногда называются *ассоциативными массивами*, поскольку каждое свойство связано со строковым значением, которое можно использовать для доступа к нему. Так, например, вы можете получить доступ к свойствам объекта `myCar` следующим образом:

```
JS
myCar["make"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

Имена свойств объекта могут быть строками JavaScript, или тем, что может быть сконвертировано в строку, включая пустую строку. Как бы то ни было, доступ к

любому имени свойства, которое содержит невалидный JavaScript идентификатор (например, имя свойства содержит в себе пробел и тире или начинается с цифры), может быть получен с использованием квадратных скобок. Этот способ записи также полезен, когда имена свойств должны быть динамически определены (когда имя свойства не определено до момента исполнения). Примеры далее:

JS

```
var myObj = new Object(),
  str = "myString",
  rand = Math.random(),
  obj = new Object();

myObj.type = "Dot syntax";
myObj["date created"] = "String with space";
myObj[str] = "String value";
myObj[rand] = "Random Number";
myObj[obj] = "Object";
myObj[""] = "Even an empty string";

console.log(myObj);
```

Обратите внимание, что все ключи с квадратными скобками преобразуются в тип String, поскольку объекты в JavaScript могут иметь в качестве ключа только тип String. Например, в приведённом выше коде, когда ключ `obj` добавляется в `myObj`, JavaScript вызывает метод `obj.toString()` и использует эту результирующую строку в качестве нового ключа.

Вы также можете получить доступ к свойствам, используя значение строки, которое хранится в переменной:

JS

```
var propertyName = "make";
myCar[propertyName] = "Ford";

propertyName = "model";
myCar[propertyName] = "Mustang";
```

Вы можете пользоваться квадратными скобками в конструкции `for...in` чтобы выполнить итерацию всех свойств объекта, для которых она разрешена. Чтобы

показать как это работает, следующая функция показывает все свойства объекта, когда вы передаёте в неё сам объект и его имя как аргументы функции:

```
JS
function showProps(obj, objName) {
  var result = "";
  for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
      result += objName + "." + i + " = " + obj[i] + "\n";
    }
  }
  return result;
}
```

Так что если вызвать эту функцию вот так `showProps(myCar, "myCar")`, то получим результат:

```
JS
myCar.make = Ford;
myCar.model = Mustang;
myCar.year = 1969;
```

Перечисление всех свойств объекта

Начиная с ECMAScript 5, есть три способа перечислить все свойства объекта (получить их список):

- циклы `for...in` (en-US) Этот метод перебирает все перечисляемые свойства объекта и его цепочку прототипов
- `Object.keys(o)` (en-US) Этот метод возвращает массив со всеми собственными (те, что в цепочке прототипов, не войдут в массив) именами перечисляемых свойств объекта `o`.
- `Object.getOwnPropertyNames(o)` (en-US) Этот метод возвращает массив содержащий все имена своих свойств (перечисляемых и неперечисляемых) объекта `o`.

До ECMAScript 5 не было встроенного способа перечислить все свойства объекта. Однако это можно сделать с помощью следующей функции:

```
JS
function listAllProperties(o) {
  var objectToInspect;
  var result = [];

  for (
    objectToInspect = o;
    objectToInspect !== null;
    objectToInspect = Object.getPrototypeOf(objectToInspect)
  ) {
    result = result.concat(Object.getOwnPropertyNames(objectToInspect));
  }

  return result;
}
```

Это может быть полезно для обнаружения скрытых (hidden) свойств (свойства в цепочке прототипа, которые недоступны через объект, в случае, если другое свойство имеет такое же имя в предыдущем звене из цепочки прототипа). Перечислить доступные свойства можно, если удалить дубликаты из массива.

Создание новых объектов

JavaScript содержит набор встроенных объектов. Также вы можете создавать свои объекты. Начиная с JavaScript 1.2, вы можете создавать объект с помощью инициализатора объекта. Другой способ — создать функцию-конструктор и сделать экземпляр объекта с помощью этой функции и оператора `new`.

Использование инициализаторов объекта

Помимо создания объектов с помощью функции-конструктора вы можете создавать объекты и другим, особым способом. Фактически, вы можете записать объект синтаксически, и он будет создан интерпретатором автоматически во время выполнения. Эта синтаксическая схема приведена ниже:

```
JS
var obj = {
  property_1: value_1, // property_# may be an identifier...
  2: value_2, // or a number...
  // ...,
```

```
  "property n": value_n,
}; // or a string
```

здесь `obj` — это имя нового объекта, каждое `property_i` — это идентификатор (имя, число или строковый литерал), и каждый `value_i` — это значения, назначенные *`property_i`*. Имя `obj` и ссылка объекта на него необязательна; если далее вам не надо будет ссылаться на данный объект, то вам не обязательно назначать объект переменной. (Обратите внимание, что вам потребуется обернуть литерал объекта в скобки, если объект находится в месте, где ожидается инструкция, чтобы интерпретатор не перепутал его с блоком.)

Если объект создан при помощи инициализатора объектов на высшем уровне скрипта, то JavaScript интерпретирует объект каждый раз, когда анализирует выражение, содержащее объект, записанный как литерал. Плюс, если пользоваться функцией инициализатором, то он будет создаваться каждый раз, когда функция вызывается.

Следующая инструкция создаёт объект и назначает его переменной `x`, когда выражение `cond` истинно.

JS

```
if (cond) var x = { hi: "there" };
```

Следующий пример создаёт объект `myHonda` с тремя свойствами. Заметьте, что свойство `engine` — это также объект со своими собственными свойствами.

JS

```
var myHonda = {
  color: "red",
  wheels: 4,
  engine: {
    cylinders: 4,
    size: 2.2,
  },
};
```

Вы также можете использовать инициализатор объекта для создания массивов. Смотрите array literals.

До JavaScript 1.1 не было возможности пользоваться инициализаторами объекта. Единственный способ создавать объекты — это пользоваться функциями-конструкторами или функциями других объектов, предназначенных для этой цели. Смотрите Using a constructor function.

Использование функции конструктора

Другой способ создать объект в два шага описан ниже:

1. Определите тип объекта, написав функцию-конструктор. Название такой функции, как правило, начинается с заглавной буквы.
2. Создайте экземпляр объекта с помощью ключевого слова `new`.

Чтобы определить тип объекта создайте функцию, которая определяет тип объекта, его имя, свойства и методы. Например предположим, что вы хотите создать тип объекта для описания машин. Вы хотите, чтобы объект этого типа назывался `car`, и вы хотите, чтобы у него были свойства make, model, и year. Чтобы сделать это, напишите следующую функцию:

JS

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Заметьте, что используется `this` чтобы присвоить значения (переданные как аргументы функции) свойствам объекта.

Теперь вы можете создать объект, называемый `mycar`, следующим образом:

JS

```
var mycar = new Car("Eagle", "Talon TSi", 1993);
```

Эта инструкция создаёт объект типа Car со ссылкой `mycar` и присваивает определённые значения его свойствам. Значением `mycar.make` станет строка "Eagle", `mycar.year` — это целое число 1993, и так далее.

Вы можете создать столько объектов `car`, сколько нужно, просто вызывая `new`. Например:

JS

```
var kenscar = new Car("Nissan", "300ZX", 1992);
var vpgscar = new Car("Mazda", "Miata", 1990);
```

Объект может иметь свойство, которое будет другим объектом. Например, далее определяется объект типа `Person` следующим образом:

JS

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

и затем создать два новых экземпляра объектов `Person` как показано далее:

JS

```
var rand = new Person("Rand McKinnon", 33, "M");
var ken = new Person("Ken Jones", 39, "M");
```

Затем, вы можете переписать определение `car` и включить в него свойство `owner`, которому назначить объект `person` следующим образом:

JS

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

Затем, чтобы создать экземпляры новых объектов, выполните следующие инструкции:

JS

```
var car1 = new Car("Eagle", "Talon TSi", 1993, rand);
var car2 = new Car("Nissan", "300ZX", 1992, ken);
```

Заметьте, что вместо того, чтобы передавать строку, литерал или целое число при создании новых объектов, в выражениях выше передаются объекты `rand` и `ken` как аргумент функции. Теперь, если вам нужно узнать имя владельца car2, это можно сделать следующим образом:

JS

```
car2.owner.name;
```

Заметьте, что в любое время вы можете добавить новое свойство ранее созданному объекту. Например, выражение

JS

```
car1.color = "black";
```

добавляет свойство `color` к car1, и устанавливает его значение равным "black." Как бы там ни было, это не влияет на любые другие объекты. Чтобы добавить новое свойство всем объектам одного типа, вы должны добавить свойство в определение типа объекта `car`.

Использование метода Object.create

Объекты также можно создавать с помощью метода `Object.create`. Этот метод очень удобен, так как позволяет вам указывать объект прототип для нового вашего объекта без определения функции конструктора.

JS

```
// список свойств и методов для Animal
var Animal = {
  type: "Invertebrates", // Значение type по умолчанию
  displayType: function () {
```

```
    // Метод отображающий тип объекта Animal
    console.log(this.type);
  },
};

// Создаём объект Animal
var animal1 = Object.create(Animal);
animal1.displayType(); // Выведет:Invertebrates

// Создаём объект Animal и присваиваем ему type = Fishes
var fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Выведет:Fishes
```

Наследование

Все объекты в JavaScript наследуются как минимум от другого объекта. Объект, от которого произошло наследование называется прототипом, и унаследованные свойства могут быть найдены в объекте `prototype` конструктора.

Индексы свойств объекта

В JavaScript 1.0 вы можете сослаться на свойства объекта либо по его имени, либо по его порядковому индексу. В JavaScript 1.1 и позже, если вы изначально определили свойство по имени, вы всегда должны ссылаться на него по его имени, и если вы изначально определили свойство по индексу, то должны ссылаться на него по его индексу.

Это ограничение налагается когда вы создаёте объект и его свойства с помощью функции конструктора (как мы это делали ранее с типом *Car*) и когда вы определяете индивидуальные свойства явно (например, `myCar.color = "red"`). Если вы изначально определили свойство объекта через индекс, например `myCar[5] = "25 mpg"` , то впоследствии сослаться на это свойство можно только так `myCar[5]` .

Исключение из правил — объекты, отображаемые из HTML, например массив `forms` . Вы всегда можете сослаться на объекты в этих массивах или используя их индекс (который основывается на порядке появления в HTML документе), или по их именам (если таковые были определены). Например, если второй html-тег `<FORM>` в документе имеет значение атрибута `NAME` равное "myForm", вы можете сослаться на

эту форму вот так: `document.forms[1]` или `document.forms["myForm"]` или `document.myForm`.

Определение свойств для типа объекта

Вы можете добавить свойство к ранее определённому типу объекта воспользовавшись специальным свойством `prototype`. Через `prototype` создаётся свойство, единое для всех объектов данного типа, а не одного экземпляра этого типа объекта. Следующий код демонстрирует это, добавляя свойство `color` ко всем объектам типа `car`, а затем присваивая значение свойству `color` объекта `car1`.

```
JS
```

```
Car.prototype.color = null;
car1.color = "black";
```

Смотрите свойство `prototype` (en-US) объекта `Function` в Справочнике JavaScript для получения деталей.

Определение методов

Метод — это функция, ассоциированная с объектом или, проще говоря, метод — это свойство объекта, являющееся функцией. Методы определяются так же, как и обычные функции, за тем исключением, что они присваиваются свойству объекта. Например вот так:

```
JS
```

```
objectName.methodname = function_name;

var myObj = {
  myMethod: function (params) {
    // ...do something
  },
};
```

где `objectName` — это существующий объект, `methodname` — это имя, которое вы присваиваете методу, и `function_name` — это имя самой функции.

Затем вы можете вызвать метод в контексте объекта следующим образом:

```
JS
object.methodname(params);
```

Вы можете определять методы для типа объекта, включая определение метода в функцию конструктора объекта. Например, вы можете определить функцию, которая форматирует и отображает свойства до этого определённых объектов `car`. Например,

```
JS
function displayCar() {
  var result = "A Beautiful " + this.year + " " + this.make + " " + this.model;
  pretty_print(result);
}
```

где `pretty_print` — это функция отображения горизонтальной линии и строки. Заметьте, что использование `this` позволяет ссылаться на объект, которому принадлежит метод.

Вы можете сделать эту функцию методом `car`, добавив инструкцию

```
JS
this.displayCar = displayCar;
```

к определению объекта. Таким образом, полное определение `car` примет следующий вид:

```
JS
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
```

Теперь вы можете вызвать метод `displayCar` для каждого из объектов как показано ниже:

JS

```
car1.displayCar();
car2.displayCar();
```

Использование `this` для ссылки на объект

В JavaScript есть специальное ключевое слово this, которое вы можете использовать внутри метода, чтобы ссылаться на текущий объект. Предположим, у вас есть функция validate, которая сверяет свойство value, переданного ей объекта с некоторыми верхним и нижним значениями:

JS

```
function validate(obj, lowval, hival) {
  if (obj.value < lowval || obj.value > hival) alert("Invalid Value!");
}
```

Вы можете вызвать эту функцию `validate` в каждом элементе формы, в обработчике события `onchange`. Используйте `this` для доступа к этому элементу, как это сделано ниже:

HTML

```
<input type="text" name="age" size="3" onChange="validate(this, 18, 99)" />
```

В общем случае, `this` ссылается на объект, вызвавший метод.

Через `this` можно обратиться и к родительской форме элемента, воспользовавшись свойством `form`. В следующем примере форма `myForm` содержит элемент ввода `Text` и кнопку `button1`. Когда пользователь нажимает кнопку, значению объекта `Text` назначается имя формы. Обработчик событий кнопки `onclick` пользуется `this.form` чтобы сослаться на текущую форму, `myForm`.

HTML

```
<form name="myForm">
  <p>
    <label>Form name:<input type="text" name="text1" value="Beluga" /></label>
  </p>
  <p>
    <input
      name="button1"
      type="button"
      value="Show Form Name"
      onclick="this.form.text1.value = this.form.name" />
  </p>
</form>
```

Определение геттеров и сеттеров

Геттер(от англ. get - получить) — это метод, который получает значение определённого свойства. *Сеттер* (от англ. set — присвоить) — это метод, который присваивает значение определённому свойству объекта. Вы можете определить геттеры и сеттеры для любых из встроенных или определённых вами объектов, которые поддерживают добавление новых свойств. Синтаксис определения геттеров и сеттеров использует литеральный синтаксис объектов.

Ниже проиллюстрировано, как могут работать геттеры и сеттеры в объекте определённом пользователем:

JS

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};

console.log(o.a); // 7
console.log(o.b); // 8
o.c = 50;
console.log(o.a); // 25
```

Объект `o` получит следующие свойства:

- `o.a` — число
- `o.b` — геттер, который возвращает `o.a` плюс 1
- `o.c` — сеттер, который присваивает значение `o.a` половине значения которое передано в `o.c`

Следует особо отметить, что имена функций, указанные в литеральной форме "[gs]et *propertyName*() { }" не будут в действительности являться именами геттера и сеттера. Чтобы задать в качестве геттера и сеттера функции с явно определёнными именами, используйте метод `Object.defineProperty` (или его устаревший аналог `Object.prototype.__defineGetter__ (en-US)`).

В коде ниже показано, как с помощью геттера и сеттера можно расширить прототип объекта `Date` и добавить ему свойство `year`, которое будет работать у всех экземпляров класса `Date`. Этот код использует существующие методы класса `Date` - `getFullYear` и `setFullYear` для работы геттера и сеттера.

Определение геттера и сеттера для свойства `year`:

JS

```
var d = Date.prototype;
Object.defineProperty(d, "year", {
  get: function () {
    return this.getFullYear();
  },
  set: function (y) {
    this.setFullYear(y);
  },
});
```

Использование свойства `year` заданного геттером и сеттером:

JS

```
var now = new Date();
console.log(now.year); // 2000
now.year = 2001; // 987617605170
```

```
console.log(now);
// Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

В принципе, геттеры и сеттеры могут быть либо:

- определены при использовании Инициализаторов объекта, или
- добавлены существующему объекту в любой момент, при использовании методов добавления геттеров и сеттеров.

Когда определение геттера и сеттера использует инициализаторы объекта, всё что вам нужно, это дополнить геттер префиксом `get` а сеттер префиксом `set`. При этом, метод геттера не должен ожидать каких либо параметров, в то время как метод сеттера принимает один единственный параметр (новое значение для присвоения свойству). Например:

JS

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};
```

Геттеры и сеттеры, могут быть добавлены существующему объекту в любой момент, при помощи метода `Object.defineProperties`. Первый параметр этого метода - объект, которому вы хотите присвоить геттер и сеттер. Второй параметр - это объект, имена свойств которого будут соответствовать именам создаваемых свойств, а значения - объекты определяющие геттер и сеттер создаваемых свойств. В следующем примере создаются в точности такие же геттер и сеттер, как и в примере выше:

JS

```
var o = { a: 0 };

Object.defineProperties(o, {
```

```
  b: {
    get: function () {
      return this.a + 1;
    },
  },
  c: {
    set: function (x) {
      this.a = x / 2;
    },
  },
});

o.c = 10; // Запускает сеттер, который присваивает 10 / 2 (5) свойству 'a'
console.log(o.b); // Запускает геттер, который возвращает a + 1 (тоесть 6)
```

То, какую из двух форм использовать для определения свойств, зависит от вашего стиля программирования и стоящей перед вами задачи. Если вы уже используете инициализатор объекта для определения прототипа, то, скорее всего, в большинстве случаев, вы воспользуетесь первой формой. Она более компактна и естественна. Однако, не редко, вторая форма является единственно возможной, в случаях, когда вы работаете с существующим объектом без доступа к его определению. Вторая форма наилучшим образом отражает динамическую природу JavaScript — но может сделать код сложным для чтения и понимания.

Удаление свойств

Вы можете удалить свойство используя оператор `delete`. Следующий код показывает как удалить свойство.

JS

```
//Creates a new object, myobj, with two properties, a and b.
var myobj = new Object();
myobj.a = 5;
myobj.b = 12;

//Removes the a property, leaving myobj with only the b property.
delete myobj.a;
```

Вы также можете воспользоваться `delete` чтобы удалить глобальную переменную, если ключевое слово `var` не было использовано при её объявлении:

JS

```
g = 17;
delete g;
```

Смотри [delete](Expressions_and_operators#delete) чтобы получить дополнительную информацию.

Сравнение объектов

В JavaScript объекты имеют ссылочный тип. Два отдельных объекта никогда не будут равными, даже если они имеют равный набор свойств. Только сравнение двух ссылок на один и тот же объект вернёт true.

JS

```
// Две переменных ссылаются на два объекта с одинаковыми свойствами
var fruit = { name: "apple" };
var fruitbear = { name: "apple" };

fruit == fruitbear; // вернёт false
fruit === fruitbear; // вернёт false
```

JS

```
// Две переменные ссылаются на один общий объект
var fruit = { name: "apple" };
var fruitbear = fruit; // присвоим переменной fruitbear ссылку на объект fruit

// теперь fruitbear и fruit ссылаются на один и тот же объект
fruit == fruitbear; // вернёт true
fruit === fruitbear; // вернёт true
```

JS

```
fruit.name = "grape";
console.log(fruitbear); // вернёт { name: "grape" } вместо { name: "apple" }
```

Подробнее смотрите Операторы сравнения (en-US).

Смотрите также

- Для детального изучения читайте подробнее об объектной модели JavaScript.
- Для изучения классов ECMAScript 2015 (новый способ определения объектов), читайте главу классы JavaScript.