JSDG 🦏 | Objects | Объекты | chapter 6

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

Объект — это составное значение: он агрегирует множество значений (элементарные значения или другие объекты) и позволяет хранить и извлекать внутренние значения по имени. Объект представляет собой неупорядоченную коллекцию *свойству* каждое из которых имеет имя и значение.

Такое сопоставление строк со значениями называется по-разному: возможно, вы уже знакомы с фундаментальной структурой данных по имени “хеш”, “хеш-таблица”, “словарь” или “ассоциативный массив”.

Помимо поддержки собственного набора свойств объект JavaScript также наследует свойства еще одного объекта, известного как его “прототип”. Методы объекта обычно представляют собой наследуемые свойства, и такое наследование прототипов” считается ключевой возможностью JavaScript.

Объекты JavaScript являются динамическими, т.к. свойства обычно можно Добавлять и удалять, но они могут использоваться для эмуляции статических объектов и “структур” из статически типизированных языков. Их также можно применять (за счет игнорирования части, касающейся значений, в отображении строк на значения) для представления наборов строк.

В JavaScript любое значение, отличающееся от строки, числа, значения Symbol, true, false, null или undefined, является объектом. И хотя строки, числа и булевские значения не относятся к объектам, они могут вести себя как неизменяемые объекты.

Временами важно иметь возможность проводить различие между свойства ми, определяемыми напрямую объектом, и

свойствами, которые унаследованы от объекта прототипа. Свойства, которые не были унаследованы, в JavaScript называются ***собственными свойствами*.**

В дополнение к имени и значению каждое свойство имеет три атрибута свойства:
• атрибут writable (допускает запись) указывает, можно ли устанавливать значение свойства;
• атрибут enumerable (допускает перечисление) указывает, возвращается ли имя свойства в цикле for/in;
• атрибут configurable (допускает конфигурирование) указывает, можно ли удалять свойство и изменять его атрибуты.

Многие встроенные объекты JavaScript имеют свойства, которые допускают только чтение, не разрешают перечисление или не поддерживают конфигурирование.

Объекты можно создавать с помощью объектных литералов, ключевого слова new и функции Object.create().
——
Объектные литералы

Самый легкий способ создания объекта предусматривает включение в код JavaScript объектного литерала. В своей простейшей форме *объектный литерал* представляет собой разделенный запятыми список пар имя: значение, заключенный в фигурные скобки.

```
let empty = {}; // Объект без свойств.
let point = { x: 0, y: 0 }; / Два числовых свойства.
let p2 = { x: point.x , y: point.y+1 }; / Более сложные значения.
let book = {
      "main title": "JavaScript", // Имена этих свойств включают пробелы и дефисы,
      "sub-title": "The Definitive Guide", // а потому для них используются строковые литералы.
      for: "all audiences", // for - зарезервированное слово,но без кавычек.
      author: {  // Значением свойства this является
            firstname: "David" // сам объект.
            surname: "Flanagan"
      )
```

}

new

let o = new Object () ; // Создает пустой объект: то же, что и {}
let a = new Array; // Создает пустой массив: то же, что и []
let d = new Date () ; // Создает объект Date, представляющий текущее время
let r = new Map () : // Создает объект Map для отображения ключ/значение

___

6.2.3. Прототипы

Чтобы можно было приступить к исследованию третьей методики создания
объектов, необходимо уделить внимание прототипам.

Почти с каждым объектом JavaScript ассоциирован второй объект JavaScript, который называется прототипом, и первый объект наследует свойства от прототипа.

Все объекты, создаваемые объектными литералами, имеют тот же самый объект - прототип, на который можно ссылаться в коде JavaScript как на Object.prototype.

Объекты, создаваемые с использованием ключевого слова new и вызова конструктора, применяют в качестве своих прототипов значение
свойства prototype функции конструктора.

Таким образом, объект, созданный посредством new Object () , наследует Object.prototype, как и объект, созданный с помощью (). Аналогично объект, созданный с использованием
new Array (), применяет в качестве прототипа Array. prototype, а объект, созданный с использованием new Date (), получает прототип Date. prototype.

В начале изучения JavaScript такая особенность может сбивать с толку. Запомните:

почти все объекты имеют прототип, но только относительно небольшое количество объектов располагают свойством prototype.

Именно эти объекты со свойствами prototype определяют прототипы для всех остальных объектов. Object.prototype - один из редких объектов, не имеющих прототипов:
он не наследует никаких свойств.

Другие объекты-прототипы являются нормальными объектами, которые имеют прототип.

Большинство встроенных конструкторов (и большинство определяемых пользователем конструкторов) имеют прототип, унаследованный от Object.prototype. Скажем, Date.prototype наследует свойства от Object.prototype, поэтому объект Date, созданный посредством new Date (), наследует свойства от Date.prototype и Object.prototype. Такая связанная последовательность объектов-прототипов называется цепочкой прототипов.

Работа наследования свойств объясняется в подразделе 6.3.2. В главе 9 более подробно обсуждается связь между прототипами и конструкторами: в ней показано, как определять новые "классы" объектов за счет написания функции конструктора и установки ее свойства prototype в объект-прототип, который должен применяться "экземплярами", созданными с помощью этого конструктора. В разделе 14.3 мы выясним, каким образом запрашивать (и даже изменять) прототип объекта.

___

Object.create()

Object. create () создает новый объект, используя в качестве его прототипа первый аргумент:

let o1 = Object.create ((x: 1, y: 2)): // o1 наследует свойства x и y.
o1.x + 01.y // => 3

Вы можете передать null, чтобы создать новый объект, не

имеющий прототипа, но в таком случае вновь созданный объект ничего не унаследует, даже базовые методы вроде toString() (т.е. он не будет работать с операцией +):

let o2 = Object.create (null); // 02 не наследует ни свойства ни методы

Если вы хотите создать обыкновенный пустой объект (подобный объекту, возвращаемому () или new Object()), тогда передайте Object .prototype:

let 03 = Object.create (Object prototype) ; // o3 подобен {} или new Object ()

Объекты как ассоциативные массивы

object.property
object ["property"]

___

Наследование

Объекты JavaScript имеют набор “собственных свойств” и вдобавок наследуют набор свойств от своих объектов-прототипов. Чтобы разобраться в наследовании, мы должны обсудить доступ к свойствам более подробно.

Если в о отсутствует собственное свойство с таким именем, тогда свойство x запрашивается в объекте-прототипе о. Если объект-прототип не имеет собственного свойства по имени x, но сам располагает прототипом, то запрос выполняется для прототипа объекта-прототипа. Процесс продолжается до тех пор, пока свойство x не будет найдено или не обнаружится объект с прототипом null.

let o = {}; // o наследует методы объекта от Object.prototype
o.x = 1; // и теперь имеет собственное свойство x.
let p = Object.create (o) ; // p наследует свойства от o и Object .prototype
p.y = 2; / и имеет собственное свойство y.
let g = Object. create (p) : // q наследует свойства от p, o и...

9. z = 3; // ...Object.prototype и имеет собственное свойство z.
let f = q.toString () ; // toString наследуется от Object.prototype
q.x + q.y // => 3; x и y наследуются от o и p

Ошибки доступа к свойствам

Если свойство x не найдено как собственное или унаследованное свойство объекта o, то вычисление выражения доступа к свойству o.x дает значение undefined.

Тем не менее, попытка запрашивания свойства объекта, который не существу ет, является ошибкой. Значения n u ll и undefined не имеют свойств, и запраши вание свойств таких значений будет ошибкой.

Как было описано в подразделе 4.4.1, в ES2020 поддерживается условный доступ к свойствам с помощью ?., что позволяет переписать предыдущее выражение присваивания следующим образом:

let surname = book?.author?. surname;

Попытка установки свойства для null или undefined также генерирует TypeError.

Удаление свойств

delete book.author; // Объект book теперь не имеет свойства author
delete book ["main title"]; // А теперь он не имеет и свойства "main title"

Кроме того, delete вычисляется как true, когда применяется (бессмысленно) с
выражением, не являющимся выражением доступа к свойству:
let o = {x: 1}; / o имеет собственное свойство x и наследует свойство toString

delete o.x // => true: свойство x удаляется
delete o.x => true: ничего не делает (x не существует), но в любом случае true
delete o.toString / => true: ничего не делает (toString - не

собственное свойство)
delete 1 // => true: бессмысленно, но в любом случае true

___

Проверка свойств

Делается с помощью операции
- **in**

Операция in ожидает с левой стороны имя свойства и с правой стороны объект. Она возвращает true, если объект имеет собственное или унаследованное свойство с указанным именем:

let o = { X: 1 };
"y" in o // => true: o имеет собственное свойство "y"
"y" in o // => false: o не имеет свойства "y"
"toString" in o // => true: o наследует свойство toString

Вместо использования операции **in часто достаточно просто** запросить свойство и применить !== для его проверки на предмет undefined:

Операция **in** может делать то, на что не способна показанная здесь простая методика доступа к свойствам.
Операция in проводит различие между свойствами, которые не существуют, и свойствами, которые существуют, но были установлены в undefined.

let o = { x: undefined }: // Свойство явно устанавливается в undefined
o.x !== undefined / => false: свойство существует, но равно undefined
o.y !== undefined " => false: свойство не существует
"x" in o / => true: свойство существует
"y" in o // => false: свойство не существует
delete o.x; // Удаление свойства x
"x" in o // => false: свойство больше не существует

- посредством методов **hasOwnProperty** ()

Метод The hasOwnProperty () объекта проверяет, имеет ли данный объект
собственное свойство с заданным именем. Для унаследованных свойств он воз-
вращает false:

let o = { x: 1 };
o.hasOwnProperty ("x") // => true: o имеет собственное свойство x
o.hasOwnProperty ("y") / => false: o не имеет свойства y
o.hasOwnProperty("toString") // => false: toString - унаследованноесвойство

- **propertyIsEnumerable** ()

Метод propertyIsEnumerable () улучшает проверку hasOwnProperty ( )

Он возвращает true, только если именованное свойство является собственным и атрибут enumerable имеет значение true.

Определенные встроенные свойства не поддерживают перечисление. Свойства, созданные нормальным кодом JavaScript, перечислимы при условии, что вы не применяли одну из методик,
показанных в разделе 14.1, чтобы сделать их не перечислимыми.

let o = { x: 1 }:
o.propertyIsEnumerable ("x") // => true: o имеет собственное перечислимое свойство x
o.propertyIsEnumerable ("toString") // =>false: не собственное свойство
Object.prototype.propertyIsEnumerable ("toString") // => false: не перечислимое свойство

- или просто путем запрашивания свойства.

Перечисление свойств

Цикл for / in был раскрыт в подразделе 5.4.5.

Он выполняет тело цикла по одному разу для каждого перечислимого свойства (собственного или унаследованного) указанного объекта, присваивая имя свойства переменной цикла.

Встроенные методы, наследуемые объектами, не являются перечислимыми, но свойства, которые ваш код добавляет к объектам, по умолчанию будут перечислимыми.

```
let o = (x: 1, y: 2, z: 3): // Три перечислимых собственных свойства
o.propertyIsEnumerable ("toString") // => false: не перечислимое
for (let p in o) { // Проход в цикле по свойствам
console.log (p); // Выводится x, y и z, но не toString
```

Чтобы избежать перечисления унаследованных свойств посредством f o r / i n , вы можете поместить внутрь тела цикла явную проверку:

```
for (let p in o) {
      if (!o.hasOwnProperty (p)) continue; // Пропускать
унаследованные свойства
      for (let p in o) if (typeof o[p] === "function") continue; / /
Пропускать все методы
}
```

====

В качестве альтернативы использованию цикла for/ in часто легче получить массив имен свойств для объекта и затем проходить по этому массиву в цикле for/of.

Есть четыре функции, которые можно применять для получения массива имен свойств:

• Функция Object. keys () возвращает массив имен перечислимых собственных свойств объекта. Она не включает не перечислимые свойства, унаследованные свойства или свойства с именами, представленными посредством значений Symbol (см. подраздел 6.10.3).

• Функция Object.getOwnPropertyNames ( ) работает подобно

Object. keys(), но возвращает массив также имен не перечислимых собственных свойств при условии, что их имена представлены строками.

• Функция Object. getOwnPropertySymbols () возвращает собственные свойства, имена которых являются значениями Symbol, перечислимые они или нет.

• Функция Reflect. ownKeys ( ) возвращает имена всех собственных свойств, перечислимых и не перечислимых, представленных как строками, так и значениями Symbol. (См. раздел 14.6.)

====

В ES6 формально определен порядок, в котором перечисляются собственные свойства объекта.

Object.keys (),
Object.getOwnPropertyNames (),
Object.getOwnPropertySymbols (),
Reflect. ownKeys ()

и связанные методы, такие как JSON.stringify (),
перечисляют свойства в следующем порядке с учетом своих дополнительных ограничений на то, перечисляют ли они неперечислимые свойства или свойства, чьи имена представлены строками или значениями Symbol.

___

Расширение объектов

в версии ES6 эта возможность стала частью базового языка JavaScript в форме Object.assign().

Функция Object.assign() ожидает получения в своих аргументах двух и более объектов. Она модифицирует и возвращает первый аргумент, в котором указан целевой объект, но не изменяет второй и любой последующий аргумент, где указаны исходные объекты.

Для каждого исходного объекта функция Object.assign() копирует его перечислимые собственные свойства (включая те, имена которых являются значениями Symbol) в целевой объект.

Функция Object.assign () копирует свойства с помощью обычных операций получения и установки свойств, поэтому если исходный объект располагает методом получения или целевой объект имеет метод установки, то они будут вызваны во время копирования, но сами не скопируются.

Одна из причин передачи свойств из одного объекта другому - когда имеется объект, который определяет стандартные значения для множества свойств, и необходимо скопировать такие стандартные свойства в другой объект, если свойства с теми же самыми именами в нем пока не существуют.

Наивное применение Object.assign () не обеспечит то, что нужно:

Object.assign(o, defaults); // Переопределяет все в o стандартными свойствами

Взамен вы можете создать новый объект, скопировать в него стандартные свойства и затем переопределить их посредством свойств в o:

o = Object.assign ({}, defaults, o);

что выразить такое действие копирования и переопределения можно также с использованием операции распространения

o = {...defaults, ...o};

Мы могли бы избежать накладных расходов, связанных с созданием и копированием дополнительного объекта, написав версию функции Object.assign (), которая копирует свойства, только если они отсутствуют:

// Похожа на Object.assign (), но не переопределяет существующие
// свойства (и также не обрабатывает свойства Symbol).

```
function merge (target, ..sources)
      for (let source of sources) {
            for (let key of Object.keys (source)) {
                  if (! (key in target) ) ( // Это отличается от Object.assign
()
                        target [key] = source [key];
                  }
            }
      }

      return target;
}

Object. assign(lx: 1), (x: 2, y: 21, (y: 3, z: 41) // 1 => (x: 2, y: 3, z: 4)
merge(x: 11, {x: 2, y: 2}, (y: 3, 2: 4}) // => { x: 1, y: 2, z: 4 }
```

Сериализация объектов

Сериализация объектов представляет собой процесс преобразования состояния объекта в строку, из которой позже он может быть восстановлен.

Функции **JSON.stringify() и JSON.parse()** сериализируют и восстанавливают объекты JavaScript.

```
let o = (x: 1, y: (z: [false, null, ""])); // Определение тестового
объекта
let s = JSON.stringify(o); // s == '("y":1, "y": ("z": [false, null, ""J))*
let p = JSON.parse(s) ; // p == (x: 1, y: (z: [false, null, ""]))
```

6.9. Методы Object

Как обсуждалось ранее, все объекты JavaScript (кроме явно созданных без прототипа) наследуют свойства от Object .prototype.

К таким свойствам в основном относятся методы, которые из-за

своей повсеместной доступности представляют особый интерес для программистов на JavaScript.

Скажем, мы уже встречались с методами hasOwnProperty() и propertylsEnumerable().

toString()

Метод toString() не имеет аргументов; он возвращает строку, каким-то образом представляющую значение объекта, на котором он вызван. Интерпретатор TavaScript вызывает данный метод всякий раз, когда объект необходимо преобразовать в строку. Подобное происходит, например, при использовании операции + для конкатенации строки с объектом или при передаче объекта методу, который ожидает строку.

```
let s = I x: 1, y: 1 ).toString (); // s == "[object Object]"

let point = {
      X: 1,
      y: 2,
      toString: function () I return ' ($(this.x), $(this.y})';
}
String (point) / => " (1, 2)": toString () применяется для
преобразований в строки
```

Метод toLocaleString()

В дополнение к базовому методу t o S t r i n g () все объекты имеют метод toLocaleString (), который предназначен для возвращения локализованного строкового представления объекта.

```
let point = (
      x: 1000,
      y: 2000,
      toString: function () I return ' ($(this.x), $(this.y)'; 1,
      toLocaleString: function () {
            return ' ($(this.x.tolocaleString()), $(this.y.tolocaleString (l})';
      };

      point.toString () 11 => " (1000, 2000)"
```

point.toLocaleString () / => " (1,000, 2,000)": обратите внимание на наличие разделителей тысяч

При реализации метода toLocaleString ( ) могут оказаться полезными
классы интернационализации, описанные в разделе 11.7.

valueOf()

Метод valueOf () во многом похож на метод toString (), но вызывается, когда интерпретатору JavaScript необходимо преобразовать объект в какой-то элементарный тип, отличающийся от строки — обычно в число.

Интерпретатор JavaScript вызывает valueOf () автоматически, если объект используется в контексте, где требуется элементарное значение. Стандартный метод valueOf () не делает ничего интересного, но некоторые встроенные классы определяют собственные методы valueOf ().

.toJSON()

В Object.prototpe на самом деле не определен метод toJSON(), но метод JSON.stringify () (см. раздел 6.8) ищет метод toJSON() в каждом объекте, который нужно сериализировать.

Если этот метод существует в обьекте, подлекащем сериализации, тогда он вызывается и сериализируется его возвращаемое значение, а не первоначальный обьект.

Класс Date (см. раздел 11.4) определяет метод toJSON(), который возвращает допускающее сериализацию строковое представление даты. Вы можете делать то же самое с объектом Point:

```
let point = {
    X: 1,
    y: 2,
    toString: function () I return ' (${this.×), $(this.y)'; 1.
    toJSON: function () { return this.toString(); }
};
JSON.stringify([point]) // => '[" (1, 2)"]'
```

___

Расширенный синтаксис объектных литералов

```
const PROPERTY NAME = "pI";
function computePropertyName () { return "p" + 2; }

let p = {
      [PROPERTY NAME] : 1,
      [computePropertyName()] : 2 // Вычисляемые имена свойств
};
p.p1 + p.p2 [/ => 3


const extension = Symbol ("my extension symbol");
let o = {
      [extension]: { /* в этом объекте хранятся данные расширения
* / )
};
o[extension].x = 0; // Это не будет конфликтовать с остальными
свойствами o
```

Однако каждое значение Symbol отличается от любого другого значения Symbol, т.е. символы хороши для создания уникальных имен свойств.
Новое значение Symbol создается вызовом фабричной функции Symbol (). (Символы представляют собой элементарные значения, не объекты, а потому Symbol () — не функция конструктора, которая вызывается с new.) Значение, возвращаемое Symbol (), не равно никакому другому значению Symbol или значению другого типа.

=====

Смысл значений Symbol связан не с защитой, а с определением безопасного механизма расширения для объектов JavaScript.

Если вы получаете объект из стороннего кода, находящегося вне вашего контроля, и хотите добавить в объект собственные свойства, но гарантировать, что они не будут конфликтовать с любыми существующими свойствами, тогда можете безопасно

использовать для имен своих свойств значения Symbol.

Поступая подобным образом, вы так же можете быть уверены в том, что сторонний код не изменит случайно ваши свойства с символьными именами.
(Разумеется, сторонний код мог бы применить Object.getOwnPropertySymbols() для выяснения используемых вами значений Symbol и затем изменить или удалить ваши свойства. Вот почему значения Symbol не являются механизмом защиты.)

___

Операция распространения

В ES2018 и последующих версиях можно копировать свойства существующего объекта в новый объект, используя внутри объектного литерала "операцию распространения" ....

```
let position = { x: 0, y: 0 ):
let dimensions = I width: 100, height: 75 ):
let rect = { ...position, ...dimensions };
rect.x + rect.y + rect.width + rect.height // => 175
```

Обратите внимание, что такой синтаксис . . . часто называется операцией распространения,

но он не является подлинной операцией JavaScript в каком-либо смысле.

Это синтаксис особого случая, доступный только внутри объектных литералов. (В других контекстах JavaScript многоточие применяется для других целей, но объектные литералы — единственный контекст, где многоточие вызывает такую вставку одного объекта в другой объект.)

Также обратите внимание, что операция распространения распространяет только собственные свойства объекта, но не унаследованные:

```
let o = Object.create ( {x: 1}); // o наследует свойство x
let p = { ...o };
p.x  //  => undefined
```

___

Сокращенная запись методов

Тем не менее, в ES6 синтаксис объектных литералов (и также синтаксис определения классов, рассматриваемый в главе 9) был расширен, чтобы сделать возможным сокращение, где ключевое слово function и двоеточие опускаются, давая в результате такой код:

```
let square = |
      area() { return this.side * this.side; }, // area: function() { return this.side * this.side; },
      side: 10
};
square.area () // => 100


const METHOD NAME = "m";
const symbol = Symbol () ;
let weirdMethods = {
      "method with Spaces" (x) { return × + 1; } ,
      [METHOD NAME] (x) { return x + 2; },
      [symbol] (x) { return x + 3 };
} ;
weirdMethods ["method With Spaces"] (1) // 2
weirdMethods [METHOD NAME] (1) // 3
weirdMethods [symbol] (1) // 4
```

Использование значения Symbol в качестве имени метода не настолько странно, как кажется.

Чтобы сделать объект итерируемым (что позволит его применять с циклом for/of), вы обязаны определить метод с символьным именем Symbol.iterator; примеры будут показаны в главе 12.

___

Методы получения и установки свойств

Все свойства объектов, которые обсуждались до сих пор, были *свойствами с данными* имеющими имена и обыкновенные значения.

В JavaScript также поддерживаются *свойства с методами доступа*, которые не имеют одиночного значения, но взамен располагают одним или двумя методами доступа: методом *получения* и/или методом *установки*.

```
let o = {
	dataProp: value, // Обыкновенное свойство с данными

	get accessorProp () I return this. dataProp; ), // Свойство с методами доступа определяется как пара функций
	set accessorProp (value) { this.dataProp = value; }
};


// Этот объект генерирует сгрого увеличивающиеся порядковые номера
const serialnum = {
	// Свойство с данными, которое хранит следующий порядковый номер. Символ в имени свойства подсказывает, что оно предназначено только для внутреннего использования.
	n: 0,

	// Возвратить текущее значение и инкрементировать его
	get next () { return this. n++; },

	// Установить новое значение п, но только если оно больше текущего
	set next (n) {
		if (n > this. n) this. n = n;
		else throw new Error ("порядковый номер можно устанавливать только в большее значение") ;
	}
}
```

serialnum.next = 10; // Установить начальный порядковый номер
serialnum.next // => 10
serialnum.next // => 11: при каждом обращении к next мы получаем отличающееся значение

Наконец, ниже приведен еще один пример использования метода получения для реализации свойства с "магическим" поведением!

// Этот объект имеет свойства с методами доступа,
// которые возвращают случайные числа.
// Скажем, каждое вычисление выражения random.octet в результате
// дает случайное число между 0 и 255.

const random = {
    get octet () (return Math. floor (Math. random () *256); J,
    get uint16 () { return Math. floor (Math. random () *65536);),
    get int16 () (return Math. floor (Math. random () *65536)
-32768; 3
}

___

Все значения JavaScript, не относящиеся к элементарным значениям, являются объектами.
Это касается массивов и функций, которые будут рассматривать ся в следующих двух главах.

___