Лабораторная работа № 3

Тема: *Операторы управления, функции. Объекты ядра JavaScript.*

Цель: *ознакомиться с функциями и операторами управления JavaScript, научиться применять их на практике.*

1 Краткая теория

1.1 ОПЕРАТОРЫ JavaScript

Операторы служат для управления потоком команд в JavaScript. Один объект может быть разбит на несколько строк, или, наоборот в одной строке может быть несколько операторов.

Необходимо знать следующее, во-первых, блоки операторов, такие как определения функций, должны быть заключены в фигурные скобки. Во-вторых, точка с запятой служит разделителем отдельных операторов. Если пропустить точку с запятой, поведение программы станет непредсказуемым.

Так как JavaScript не имеет жестких требований к форматированию текста программы, можно вставлять символы перевода строки и отступа для лучшей читабельности текста.

Ниже описаны операторы, которые используются в JavaScript:

- break – специальная директива, которая помогает выходить из в любой момент. сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.
- comment – оставляет комментарии.
- continue – позволяет избегать вложения. ! ВАЖНО ! *continue/break* не являются выражениями и не могут быть использованы с тернарным оператором ?. В частности, использование таких директив, как break/continue, вызовет ошибку.

for – самый распространенный цикл. Алгоритм работы цикла выглядит следующим образом:

```
Выполнить *начало*
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ ...
```

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.

- for...in
- function – объявление функции.
- if...else
- return
- var – ключевое слово *почти* то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.
- while – имеет следующий синтаксис:

```
while (condition) {
  // код
```

```
  // также называемый "телом цикла"
}
```

Код из тела цикла выполняется, пока условие condition истинно.

- with

1.2 ФУНКЦИИ JavaScript

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 alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

Объявление функции

Для создания функций мы можем использовать *объявление функции*.

Пример объявления функции:

```
function showMessage() {
  alert( '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 её имени: showMessage().

Например:

```
function showMessage() {
  alert( '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
  alert( message );
}
showMessage(); // Привет, я JavaScript!
alert( 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;
  alert(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;
  alert(message);
}
alert( userName ); // Вася перед вызовом функции
showMessage();
alert( 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. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована:

```
let userName = 'Вася';
function showMessage() {
  let userName = "Петя"; // объявляем локальную переменную
  let message = 'Привет, ' + userName; // Петя
  alert(message);
}
// функция создаст и будет использовать свою собственную локальную переменную userName
```

```
showMessage();
alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную
```

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 (также называемые *аргументами функции*).

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

```
function showMessage(from, text) { // аргументы: from, text
  alert(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.

Вот ещё один пример: у нас есть переменная from, и мы передаём её функции. Обратите внимание: функция изменяет значение from, но это изменение не видно снаружи. Функция всегда получает только копию значения:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // немного украсим "from"
  alert( from + ': ' + text );
}
let from = "Аня";
showMessage(from, "Привет"); // *Аня*: Привет
// значение "from" осталось прежним, функция изменила значение локальной переменной
alert( from ); // Аня
```

Параметры по умолчанию

Если параметр не 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") {
  alert( from + ": " + text );
}
showMessage("Аня"); // Аня: текст не добавлен
```

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

В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра.

В примере выше anotherFunction() будет вызываться каждый раз, когда showMessage() вызывается без параметра text.

Использование параметров по умолчанию в ранних версиях JavaScript

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах.

Например, явная проверка на undefined:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'текст не добавлен';
  }
  alert( from + ": " + text );
}
```

…Или с помощью оператора ||:

```
function showMessage(from, text) {
  // Если значение text ложно, тогда присвоить параметру text значение по умолчанию
  text = text || '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);
alert( 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) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции.

Например:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }
  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

В коде выше, если checkAge(age) вернёт false, showMovie не выполнит alert.

Результат функции с пустым return или без него – undefined

Если функция не возвращает значения, это всё равно, как если бы она возвращала undefined:

```
function doNothing() { /* пусто */ }
alert( doNothing() === undefined ); // true
```

Пустой return аналогичен return undefined:

```
function doNothing() {
  return;
```

```
}
alert( doNothing() === undefined ); // true
```

! *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 простым, 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 (в каком либо значении)
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 `alert` 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.

! *Сверхкороткие имена функций*

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке jQuery есть функция с именем `$`. В библиотеке Lodash основная функция представлена именем `_`.

Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

Например, сравним ниже две функции `showPrimes(n)`. Каждая из них выводит простое число до `n`.

Первый вариант использует метку `nextPrime`:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert( i ); // простое
  }
}
```

Второй вариант использует дополнительную функцию isPrime(n) для проверки на простое:

```
function showPrimes(n) {
  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
    alert(i);  // простое
  }
}
function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Второй вариант легче для понимания, не правда ли? Вместо куска кода мы видим название действия (isPrime). Иногда разработчики называют такой код *самодокументируемым*.

Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

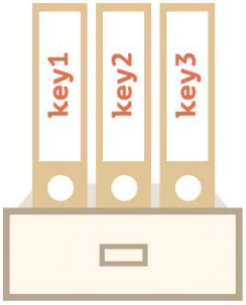
1.12 ОБЪЕКТЫ ЯДРА JavaScript

В JavaScript существует 8 типов данных. Семь из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое).

Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

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, используя один из двух вариантов синтаксиса:

```
let user = new Object(); // синтаксис "конструктор объекта"
let user = {};  // синтаксис "литерал объекта"
```

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом объекта* или *литеральной нотацией*.

Литералы и свойства

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

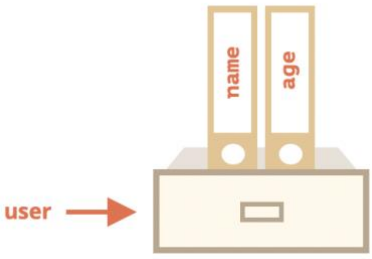
```
let user = {     // объект
  name: "John",  // под ключом "name" хранится значение "John"
  age: 30        // под ключом "age" хранится значение 30
};
```

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».



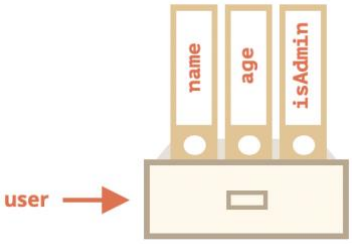
Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

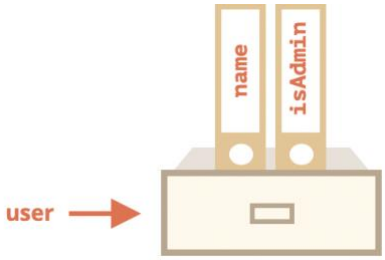
```
// получаем свойства объекта:
alert( user.name ); // John
alert( user.age ); // 30
```

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

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

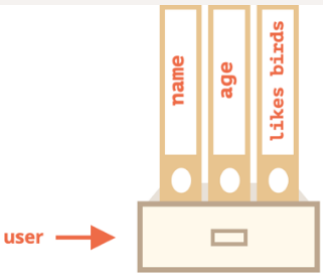


Для удаления свойства мы можем использовать оператор delete:



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
};
```



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"; // (*)
alert(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
alert(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
alert( user[key] ); // John (если ввели "name")
```

Запись «через точку» такого не позволяет:

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

```
alert( user.key ); // undefined
```

Ограничения на имена свойств

Как мы уже знаем, имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д.

Но для свойств объекта такого ограничения нет:

```
// эти имена свойств допустимы
let obj = {
  for: 1,
  let: 2,
  return: 3
};
alert( obj.for + obj.let + obj.return );  // 6
```

Иными словами, нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов (специальный тип для идентификаторов, который будет рассмотрен позже).

Все другие типы данных будут автоматически преобразованы к строке.

Например, если использовать число `0` в качестве ключа, то оно превратится в строку `"0"`:

```
let obj = {
  0: "Тест" // то же самое что и "0": "Тест"
};
// обе функции alert выведут одно и то же свойство (число 0 преобразуется в строку "0")
alert( obj["0"] ); // Тест
alert( obj[0] ); // Тест (то же свойство)
```

Есть небольшой подводный камень, связанный со специальным свойством `__proto__`. Мы не можем установить его в необъектное значение:

```
let obj = {};
obj.__proto__ = 5; // присвоим число
alert(obj.__proto__); // [object Object], значение - это объект, т.е. не то, что мы ожидали
```

Как мы видим, присвоение примитивного значения `5` игнорируется.

Проверка существования свойства, оператор «in»

В отличие от многих других языков, особенность JavaScript-объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет!

При обращении к свойству, которого нет, возвращается `undefined`. Это позволяет просто проверить существование свойства:

```
let user = {};
alert( user.noSuchProperty === undefined ); // true означает "свойства нет"
```

Также существует специальный оператор `"in"` для проверки существования свойства в объекте.

Синтаксис оператора:

```
"key" in object
```

Пример:

```
let user = { name: "John", age: 30 };
alert( "age" in user ); // true, user.age существует
alert( "blabla" in user ); // false, user.blabla не существует
```

Обратите внимание, что слева от оператора in должно быть *имя свойства*. Обычно это строка в кавычках.

Если мы опускаем кавычки, это значит, что мы указываем переменную, в которой находится имя свойства. Например:

```
let user = { age: 30 };
let key = "age";
alert( key in user ); // true, имя свойства было взято из переменной key
```

Для чего вообще нужен оператор in? Разве недостаточно сравнения с undefined?

В большинстве случаев прекрасно сработает сравнение с undefined. Но есть особый случай, когда оно не подходит, и нужно использовать "in".

Это когда свойство существует, но содержит значение undefined:

```
let obj = {
  test: undefined
};
alert( obj.test ); //  выведет undefined, значит свойство не существует?
alert( "test" in obj ); // true, свойство существует!
```

В примере выше свойство obj.test технически существует в объекте. Оператор inсработал правильно.

Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств мы используем значение null. Таким образом, оператор in является экзотическим гостем в коде.

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
  alert( key );  // name, age, isAdmin
  // значения ключей
```

```
  alert( user[key] ); // John, 30, true
}
```

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

Кроме того, мы могли бы использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

Упорядочение свойств объекта

Упорядочены ли свойства объекта? Другими словами, если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? Можем ли мы на это рассчитывать?

Короткий ответ: свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания.

В качестве примера рассмотрим объект с телефонными кодами:

```
let codes = {
  "49": "Германия",
  "41": "Швейцария",
  "44": "Великобритания",
  // ..,
  "1": "США"
};
for (let code in codes) {
  alert(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:

```
// Math.trunc - встроенная функция, которая удаляет десятичную часть
alert( String(Math.trunc(Number("49"))) ); // "49", то же самое ⇒ свойство целочисленное
alert( String(Math.trunc(Number("+49"))) ); // "49", не то же самое, что "+49" ⇒ свойство не целочисленное
```

```
alert( String(Math.trunc(Number("1.2"))) ); // "1", не то же самое, что "1.2" ⇒ свойство не целочисленное
```

…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) {
  alert( 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) {
  alert( +code ); // 49, 41, 44, 1
}
```

Теперь код работает так, как мы задумывали.

Копирование объектов и ссылки

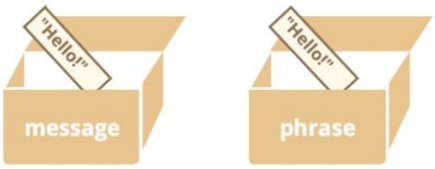
Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке».

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

Например:

```
let message = "Привет!";
let phrase = message;
```

В результате мы имеем две независимые переменные, каждая из которых хранит строку `"Привет!"`.

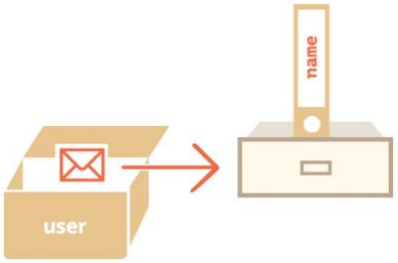


Объекты ведут себя иначе.

! Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. !

Проиллюстрируем это:

```
let user = {
  name: "Иван"
};
```



Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти.

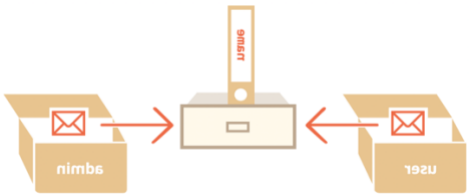
! Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется. !

Если мы представляем объект как ящик, то переменная – это ключ к нему. Копирование переменной дублирует ключ, но не сам ящик.

Например:

```
let user = { name: "Иван" };
let admin = user; // копируется ссылка
```

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект:



Мы можем использовать любую из переменных для доступа к ящику и изменения его содержимого:

```
let user = { name: 'Иван' };
let admin = user;
admin.name = 'Петя'; // изменено по ссылке из переменной "admin"
alert(user.name); // 'Петя', изменения видны по ссылке из переменной "user"
```

Приведённый выше пример демонстрирует, что объект только один. Как если бы у нас был один ящик с двумя ключами и мы использовали один из них (admin), чтобы войти в него и что-то изменить, а затем, открыв ящик другим ключом (user), мы бы увидели эти изменения.

Сравнение по ссылке

Операторы равенства == и строгого равенства === для объектов работают одинаково.

! Два объекта равны только в том случае, если это один и тот же объект. !

В примере ниже две переменные ссылаются на один и тот же объект, поэтому они равны друг другу:

```
let a = {};
let b = a; // копирование по ссылке
alert( a == b ); // true, т.к. обе переменные ссылаются на один и тот же объект
alert( a === b ); // true
```

В другом примере два разных объекта не равны, хотя оба пусты:

```
let a = {};
let b = {}; // два независимых объекта
alert( a == b ); // false
```

Для сравнений типа `obj1 > obj2` или для сравнения с примитивом `obj == 5` объекты преобразуются в примитивы.

Клонирование и объединение объектов, Object.assign

При копировании переменной с объектом создаётся ещё одна ссылка на тот же самый объект.

Но что, если нам всё же нужно дублировать объект? Создать независимую копию, клон?

Это выполнимо, но немного сложно, так как в JavaScript нет встроенного метода для этого. Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их.

Например так:

```
let user = {
  name: "Иван",
  age: 30
};
let clone = {}; // новый пустой объект
// скопируем все свойства user в него
for (let key in user) {
  clone[key] = user[key];
}
// теперь в переменной clone находится абсолютно независимый клон объекта
clone.name = "Пётр"; // изменим в нём данные
alert( user.name ); // в оригинальном объекте значение свойства `name` осталось прежним – Иван.
```

Кроме того, для этих целей мы можем использовать метод Object.assign.

Синтаксис:

```
Object.assign(dest, [src1, src2, src3...])
```

- Первый аргумент `dest` — целевой объект.
- Остальные аргументы `src1, ..., srcN` (может быть столько, сколько нужно)) являются исходными объектами

- Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект.
- Возвращает объект dest.

Например, объединим несколько объектов в один:

```
let user = { name: "Иван" };
let permissions1 = { canView: true };
let permissions2 = { canEdit: true };
// копируем все свойства из permissions1 и permissions2 в user
Object.assign(user, permissions1, permissions2);
// теперь user = { name: "Иван", canView: true, canEdit: true }
```

Если принимающий объект (user) уже имеет свойство с таким именем, оно будет перезаписано:

```
let user = { name: "Иван" };
Object.assign(user, { name: "Пётр" });
alert(user.name); // теперь user = { name: "Пётр" }
```

Мы также можем использовать Object.assign для замены for..in на простое клонирование:

```
let user = {
  name: "Иван",
  age: 30
};
let clone = Object.assign({}, user);
```

Этот метод скопирует все свойства объекта user в пустой объект и возвратит его.

Вложенное клонирование

До сих пор мы предполагали, что все свойства объекта user хранят примитивные значения. Но свойства могут быть ссылками на другие объекты. Что с ними делать?

Например, есть объект:

```
let user = {
  name: "Иван",
  sizes: {
    height: 182,
    width: 50
  }
};
alert( user.sizes.height ); // 182
```

Теперь при клонировании недостаточно просто скопировать clone.sizes = user.sizes, поскольку user.sizes – это объект, он будет скопирован по ссылке. А значит объекты clone и user в своих свойствах sizes будут ссылаться на один и тот же объект:

```
let user = {
  name: "Иван",
  sizes: {
```

```
    height: 182,
    width: 50
  }
};
let clone = Object.assign({}, user);
alert( user.sizes === clone.sizes ); // true, один и тот же объект
// user и clone обращаются к одному sizes
user.sizes.width++;       // меняем свойство в одном объекте
alert(clone.sizes.width); // 51, видим результат в другом объекте
```

Чтобы исправить это, мы должны в цикле клонирования делать проверку, не является ли значение user[key] объектом, и если это так – скопировать и его структуру тоже. Это называется «глубокое клонирование».

Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед, использовать готовую реализацию — метод _.cloneDeep(obj) из JavaScript-библиотеки lodash.

Сборка мусора

- Сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её.
- Объекты сохраняются в памяти, пока они достижимы.
- Наличие ссылки не гарантирует, что объект достижим (от корня): несколько взаимосвязанных объектов могут стать недостижимыми как единое целое.

Современные интерпретаторы реализуют передовые алгоритмы сборки мусора.

Методы объекта, "this"

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

```
// Объект пользователя
let user = {
  name: "Джон",
  age: 30
};
```

И так же, как и в реальном мире, пользователь может *совершать действия*: выбирать что-то из корзины покупок, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

- Функции, которые находятся в объекте в качестве его свойств, называются «методами».
- Методы позволяют объектам «действовать»: object.doSomething().
- Методы могут ссылаться на объект через this.

Значение this определяется во время исполнения кода.

- При объявлении любой функции в ней можно использовать this, но этот this не имеет значения до тех пор, пока функция не будет вызвана.

- Эта функция может быть скопирована между объектами (из одного объекта в другой).
- Когда функция вызывается синтаксисом «метода» – object.method(), значением thisво время вызова является объект перед точкой.

Также ещё раз заметим, что стрелочные функции являются особенными – у них нет this. Когда внутри стрелочной функции обращаются к this, то его значение берётся снаружи.

Конструкторы, создание объектов через "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора "new".

Функция-конструктор

Функции-конструкторы являются обычными функциями. Но есть *два соглашения*:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Вася");
alert(user.name); // Вася
alert(user.isAdmin); // false
```

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Опциональная цепочка '?.'

Синтаксис опциональной цепочки ?. имеет три формы:

1. obj?.prop – возвращает obj.prop, если существует obj, и undefined в противном случае.
2. obj?.[prop] – возвращает obj[prop], если существует obj, и undefined в противном случае.
3. obj.method?.() – вызывает obj.method(), если существует obj.method, в противном случае возвращает undefined.

Как мы видим, все они просты и понятны в использовании. ?. проверяет левую часть выражения на равенство null/undefined, и продолжает дальнейшее вычисление, только если это не так.

Цепочка ?. позволяет без возникновения ошибок обратиться к вложенным свойствам.

Тем не менее, нужно разумно использовать ?. — только там, где это уместно, если допустимо, что левая часть не существует. Чтобы таким образом не скрывать возможные ошибки программирования.

Преобразование объектов в примитивы

Преобразование объектов в примитивы вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве аргумента.

Существует всего 3 типа преобразований (хинтов):

- "string" (для alert и других операций, которым нужна строка)
- "number" (для математических операций)
- "default" (для некоторых операций)

В спецификации явно указано, какой хинт должен использовать каждый оператор. И существует совсем немного операторов, которые не знают, что ожидать, и используют хинт со значением "default". Обычно для встроенных объектов хинт "default" обрабатывается так же, как "number". Таким образом, последние два очень часто объединяют вместе.

Алгоритм преобразований к примитивам следующий:

1. Сначала вызывается метод obj[Symbol.toPrimitive](hint), если он существует.
2. Иначе, если хинт равен "string"
   - происходит попытка вызвать obj.toString(), затем obj.valueOf(), смотря что есть.
3. Иначе, если хинт равен "number" или "default"
   - происходит попытка вызвать obj.valueOf(), затем obj.toString(), смотря что есть.

На практике довольно часто достаточно реализовать только obj.toString() как «универсальный» метод для всех типов преобразований, возвращающий «читаемое» представление объекта, достаточное для логирования или отладки.

Методы у примитивов

- Все примитивы, кроме null и undefined, предоставляют множество полезных методов. Мы познакомимся с ними поближе в следующих главах.
- Формально эти методы работают с помощью временных объектов, но движки JavaScript внутренне очень хорошо оптимизируют этот процесс, так что их вызов не требует много ресурсов.

Числа

Чтобы писать числа с большим количеством нулей:

- Используйте краткую форму записи чисел – "e", с указанным количеством нулей. Например: 123e6 это 123 с 6-ю нулями 123000000.
- Отрицательное число после "e" приводит к делению числа на 1 с указанным количеством нулей. Например: 123e-6 это 0.000123 (123 миллионных).

Для других систем счисления:

- Можно записывать числа сразу в шестнадцатеричной (0x), восьмеричной (0o) и бинарной (0b) системах счисления
- parseInt(str, base) преобразует строку в целое число в соответствии с указанной системой счисления: 2 ≤ base ≤ 36.
- num.toString(base) представляет число в строковом виде в указанной системе счисления base.

Для преобразования значений типа 12pt и 100px в число:

- Используйте parseInt/parseFloat для «мягкого» преобразования строки в число, данные функции по порядку считывают число из строки до тех пор пока не возникнет ошибка.

Для дробей:

- Используйте округления Math.floor, Math.ceil, Math.trunc, Math.round или num.toFixed(precision)
- Помните, что при работе с дробями происходит потеря точности.

Ещё больше математических функций:

- Документация по объекту Math. Библиотека маленькая, но содержит всё самое важное.

Строки

- Есть три типа кавычек. Строки, использующие обратные кавычки, могут занимать более одной строки в коде и включать выражения ${…}.
- Строки в JavaScript кодируются в UTF-16.
- Есть специальные символы, такие как \n, и можно добавить символ по его юникодному коду, используя \u….
- Для получения символа используйте [].
- Для получения подстроки используйте slice или substring.
- Для того, чтобы перевести строку в нижний или верхний регистр, используйте toLowerCase/toUpperCase.
- Для поиска подстроки используйте indexOf или includes/startsWith/endsWith, когда надо только проверить, есть ли вхождение.
- Чтобы сравнить строки с учётом правил языка, используйте localeCompare.

Строки также имеют ещё кое-какие полезные методы:

- str.trim() — убирает пробелы в начале и конце строки.
- str.repeat(n) — повторяет строку n раз.
- …и другие, которые вы можете найти в справочнике.

Массивы

Массив – это особый тип объекта, предназначенный для работы с упорядоченным набором элементов.

- Объявление:

```
// квадратные скобки (обычно)
let arr = [item1, item2...];
```

```
// new Array (очень редко)
let arr = new Array(item1, item2...);
```

Вызов new Array(number) создаёт массив с заданной длиной, но без элементов.

- Свойство length отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.
- Если мы уменьшаем length вручную, массив укорачивается.

Мы можем использовать массив как двустороннюю очередь, используя следующие операции:

- push(...items)добавляет items в конец массива.
- pop() удаляет элемент в конце массива и возвращает его.
- shift() удаляет элемент в начале массива и возвращает его.
- unshift(...items) добавляет items в начало массива.

Чтобы пройтись по элементам массива:

- for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.
- for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).
- for (let i in arr) – никогда не используйте для массивов!

Методы массивов

Массивы предоставляют множество методов. Чтобы было проще, в этой главе они разбиты на группы.

- Для добавления/удаления элементов:
  - push (...items) – добавляет элементы в конец,
  - pop() – извлекает элемент с конца,
  - shift() – извлекает элемент с начала,
  - unshift(...items) – добавляет элементы в начало.
  - splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items.
  - slice(start, end) – создаёт новый массив, копируя в него элементы с позиции start до end (не включая end).
  - concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
- Для поиска среди элементов:
  - indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
  - includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
  - find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
  - findIndex похож на find, но возвращает индекс вместо значения.

- Для перебора элементов:
  - forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
- Для преобразования массива:
  - map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
  - sort(func) – сортирует массив «на месте», а потом возвращает его.
  - reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
  - split/join – преобразует строку в массив и обратно.
  - reduce(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
- Дополнительно:
  - Array.isArray(arr) проверяет, является ли arr массивом.

Обратите внимание, что методы sort, reverse и splice изменяют исходный массив.

Изученных нами методов достаточно в 99% случаев, но существуют и другие.

- arr.some(fn)/arr.every(fn) проверяет массив.

Функция fn вызывается для каждого элемента массива аналогично map. Если какие-либо/все результаты вызовов являются true, то метод возвращает true, иначе false.

- arr.fill(value, start, end) – заполняет массив повторяющимися value, начиная с индекса start до end.
- arr.copyWithin(target, start, end) – копирует свои элементы, начиная со start и заканчивая end, в *собственную* позицию target (перезаписывает существующие).

Полный список есть в справочнике MDN.

На первый взгляд может показаться, что существует очень много разных методов, которые довольно сложно запомнить. Но это гораздо проще, чем кажется.

Object.keys, values, entries

Для простых объектов доступны следующие методы:

- Object.keys(obj) – возвращает массив ключей.
- Object.values(obj) – возвращает массив значений.
- Object.entries(obj) – возвращает массив пар [ключ, значение].

Обратите внимание на различия (по сравнению с map, например):

	Map	**Object**
Синтаксис вызова	map.keys()	Object.keys(obj), не obj.keys()
Возвращает	перебираемый объект	«реальный» массив

Первое отличие в том, что мы должны вызвать Object.keys(obj), а не obj.keys().

Почему так? Основная причина – гибкость. Помните, что объекты являются основой всех сложных структур в JavaScript. У нас может быть объект data, который реализует свой собственный метод data.values(). И мы всё ещё можем применять к нему стандартный метод Object.values(data).

Второе отличие в том, что методы вида Object.* возвращают «реальные» массивы, а не просто итерируемые объекты. Это в основном по историческим причинам.

Например:

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

- Object.keys(user) = ["name", "age"]
- Object.values(user) = ["John", 30]
- Object.entries(user) = [["name","John"], ["age",30]]

Вот пример использования Object.values для перебора значений свойств в цикле:

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

// перебор значений
for (let value of Object.values(user)) {
  alert(value); // John, затем 30
}
```

Object.keys/values/entries игнорируют символьные свойства

Так же, как и цикл for..in, эти методы игнорируют свойства, использующие Symbol(...) в качестве ключей.

Обычно это удобно. Но если требуется учитывать и символьные ключи, то для этого существует отдельный метод Object.getOwnPropertySymbols, возвращающий массив только символьных ключей. Также, существует метод Reflect.ownKeys(obj), который возвращает *все* ключи.

Трансформации объекта

У объектов нет множества методов, которые есть в массивах, например map, filter и других.

Если мы хотели бы их применить, то можно использовать Object.entries с последующим вызовом Object.fromEntries:

1. Вызов Object.entries(obj) возвращает массив пар ключ/значение для obj.
2. На нём вызываем методы массива, например, map.
3. Используем Object.fromEntries(array) на результате, чтобы преобразовать его обратно в объект.

Например, у нас есть объект с ценами, и мы хотели бы их удвоить:

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};
let doublePrices = Object.fromEntries(
  // преобразовать в массив, затем map, затем fromEntries обратно объект
  Object.entries(prices).map(([key, value]) => [key, value * 2])
);
alert(doublePrices.meat); // 8
```

Деструктурирующее присваивание

- Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
- Полный синтаксис для объекта:

```
let {prop : varName = default, ...rest} = object
```

Свойства, которые не были упомянуты, копируются в объект rest.

- Полный синтаксис для массива:

```
let [item1 = default, item2, ...rest] = array
```

Первый элемент отправляется в item1; второй отправляется в item2, все остальные элементы попадают в массив rest.

- Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

Дата и время

- Дата и время в JavaScript представлены объектом Date. Нельзя создать «только дату» или «только время»: объекты Date всегда содержат и то, и другое.
- Счёт месяцев начинается с нуля (да, январь – это нулевой месяц).
- Дни недели в getDay() также отсчитываются с нуля, что соответствует воскресенью.
- Объект Date самостоятельно корректируется при введении значений, выходящих за рамки допустимых. Это полезно для сложения/вычитания дней/месяцев/недель.
- Даты можно вычитать, и разность возвращается в миллисекундах. Так происходит, потому что при преобразовании в число объект Date становится таймстампом.
- Используйте Date.now() для быстрого получения текущего времени в формате таймстампа.

Учтите, что, в отличие от некоторых других систем, в JavaScript таймстамп в миллисекундах, а не в секундах.

Порой нам нужно измерить время с большей точностью. Собственными средствами JavaScript измерять время в микросекундах (одна миллионная секунды) нельзя, но в большинстве сред такая возможность есть. К примеру, в

браузерах есть метод performance.now(), возвращающий количество миллисекунд с начала загрузки страницы с точностью до микросекунд (3 цифры после точки):

```
alert(`Загрузка началась ${performance.now()}мс назад`);
// Получаем что-то вроде: "Загрузка началась 34731.26000000001мс назад"
// .26 –- это микросекунды (260 микросекунд)
// корректными являются только первые три цифры после точки, а остальные -- это ошибка точности
```

В Node.js для этого предусмотрен модуль microtime и ряд других способов. Технически почти любое устройство или среда позволяет добиться большей точности, просто её нет в объекте Date.

Остаточные параметры и оператор расширения

Остаточные параметры (...)

Вызывать функцию можно с любым количеством аргументов независимо от того, как она была определена.

Например:

```
function sum(a, b) {
  return a + b;
}
alert( sum(1, 2, 3, 4, 5) );
```

Лишние аргументы не вызовут ошибку. Но, конечно, посчитаются только первые два.

Остаточные параметры могут быть обозначены через три точки .... Буквально это значит: «собери оставшиеся параметры и положи их в массив».

Например, соберём все аргументы в массив args:

```
function sumAll(...args) { // args — имя массива
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Мы можем положить первые несколько параметров в переменные, а остальные – собрать в массив.

В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Юлий Цезарь
  // Оставшиеся параметры пойдут в массив
  // titles = ["Консул", "Император"]
  alert( titles[0] ); // Консул
  alert( titles[1] ); // Император
  alert( titles.length ); // 2
}
```

```
showName("Юлий", "Цезарь", "Консул", "Император");
```

Остаточные параметры должны располагаться в конце

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку:

```
function f(arg1, ...rest, arg2) { // arg2 после ...rest ?!
  // Ошибка
}
```

...rest должен всегда быть последним.

Переменная "arguments"

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами.

Например:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );
  // Объект arguments можно перебирать
  // for (let arg of arguments) alert(arg);
}
// Вывод: 2, Юлий, Цезарь
showName("Юлий", "Цезарь");
// Вывод: 1, Илья, undefined (второго аргумента нет)
showName("Илья");
```

Замыкание

JavaScript – язык с сильным функционально-ориентированным уклоном. Он даёт нам много свободы. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места.

Мы знаем, что функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Но что произойдёт, когда внешние переменные изменятся? Функция получит последнее значение или то, которое существовало на момент создания функции?

И что произойдёт, когда функция переместится в другое место в коде и будет вызвана оттуда – получит ли она доступ к внешним переменным своего нового местоположения?

Разные языки ведут себя по-разному в таких случаях, и в этой главе мы рассмотрим поведение JavaScript.

Объект лексического окружения состоит из двух частей:

1. *Environment Record* – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).

2. Ссылка на *внешнее лексическое окружение* – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

! "Переменная" – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта». !

Function Declaration

Рассмотрим Function Declaration.

! В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение. !

Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение.

Вот почему мы можем вызвать функцию, объявленную через Function Declaration, до того, как она определена.

! Один вызов – одно лексическое окружение!

Обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется. И, если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

! Лексическое окружение – это специальный внутренний объект!

«Лексическое окружение» – это специальный внутренний объект. Мы не можем получить его в нашем коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки, но видимое поведение объекта должно оставаться таким, как было описано.

Глобальный объект

- Глобальный объект хранит переменные, которые должны быть доступны в любом месте программы.

Это включает в себя как встроенные объекты, например, Array, так и характерные для окружения свойства, например, window.innerHeight – высота окна браузера.

- Глобальный объект имеет универсальное имя – globalThis.

…Но чаще на него ссылаются по-старому, используя имя, характерное для данного окружения, такое как window (браузер) и global (Node.js). Так как globalThisпоявился недавно, он не поддерживается в IE и Edge (не-Chromium версия), но можно использовать полифил.

- Следует хранить значения в глобальном объекте, только если они действительно глобальны для нашего проекта. И стараться свести их количество к минимуму.
- В браузерах, если только мы не используем модули, глобальные функции и переменные, объявленные с помощью var, становятся свойствами глобального объекта.

- Для того, чтобы код был проще и в будущем его легче было поддерживать, следует обращаться к свойствам глобального объекта напрямую, как window.x.

Синтаксис "new Function"

Синтаксис:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

По историческим причинам аргументы также могут быть объявлены через запятую в одной строке.

Эти 3 объявления ниже эквивалентны:

```
new Function('a', 'b', 'return a + b'); // стандартный синтаксис
new Function('a,b', 'return a + b'); // через запятую в одной строке
new Function('a , b', 'return a + b'); // через запятую с пробелами в одной строке
```

Функции, объявленные через new Function, имеют [[Environment]], ссылающийся на глобальное лексическое окружение, а не на родительское. Поэтому они не могут использовать внешние локальные переменные. Но это очень хорошо, потому что страхует нас от ошибок. Переданные явно параметры – гораздо лучшее архитектурное решение, которое не вызывает проблем у минификаторов.

Планирование: setTimeout и setInterval

- Методы setInterval(func, delay, ...args) и setTimeout(func, delay, ...args) позволяют выполнять func регулярно или только один раз после задержки delay, заданной в мс.
- Для отмены выполнения необходимо вызвать clearInterval/clearTimeout со значением, которое возвращают методы setInterval/setTimeout.
- Вложенный вызов setTimeout является более гибкой альтернативой setInterval. Также он позволяет более точно задать интервал между выполнениями.
- Планирование с нулевой задержкой setTimeout(func,0) или, что то же самое, setTimeout(func) используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.
- Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами setTimeout, а также для setInterval, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

- Перегружен процессор.
- Вкладка браузера в фоновом режиме.
- Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Декораторы и переадресация вызова, call/apply

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты, и сейчас мы рассмотрим, как *перенаправлять* вызовы между ними и как их декорировать.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства, такие как func.calledCount или типа того, то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

Декораторы можно рассматривать как «дополнительные возможности» или «аспекты», которые можно добавить в функцию. Мы можем добавить один или несколько декораторов. И всё это без изменения кода оригинальной функции!

Для реализации cachingDecorator мы изучили методы:

- func.call(context, arg1, arg2…) – вызывает func с данным контекстом и аргументами.
- func.apply(context, args) – вызывает func, передавая context как this и псевдомассив args как список аргументов.

В основном *переадресация вызова* выполняется с помощью apply:

```
let wrapper = function(original, arguments) {
  return original.apply(this, arguments);
};
```

Мы также рассмотрели пример *заимствования метода*, когда мы вызываем метод у объекта в контексте другого объекта. Весьма распространено заимствовать методы массива и применять их к arguments. В качестве альтернативы можно использовать объект с остаточными параметрами ...args, который является реальным массивом.

Привязка контекста к функции

При передаче методов объекта в качестве колбэков, например для setTimeout, возникает известная проблема – потеря this.

Метод bind возвращает «привязанный вариант» функции func, фиксируя контекст this и первые аргументы arg1, arg2…, если они заданы.

Обычно bind применяется для фиксации this в методе объекта, чтобы передать его в качестве колбэка. Например, для setTimeout.

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной».

Задание 1

Создайте объект calculator (калькулятор) с тремя методами:

- *read() (читать) запрашивает два значения и сохраняет их как свойства объекта.*
- *sum() (суммировать) возвращает сумму сохранённых значений.*
- *mul() (умножить) перемножает сохранённые значения и возвращает результат.*

Задание 2

Напишите функцию-конструктор Accumulator(startingValue).

Объект, который она создаёт, должен уметь следующее:

- *Хранить «текущее значение» в свойстве value. Начальное значение устанавливается в аргументе конструктора startingValue.*
- *Метод read() использует prompt для получения числа и прибавляет его к свойству value.*

Таким образом, свойство value является текущей суммой всего, что ввёл пользователь при вызовах метода read(), с учётом начального значения startingValue.

Задание 3

Напишите функцию random(min, max), которая генерирует случайное число с плавающей точкой от min до max (но не включая max).

Задание 4

Есть стоимость в виде строки "$120". То есть сначала идёт знак валюты, а затем – число.

Создайте функцию extractCurrencyValue(str), которая будет из такой строки выделять числовое значение и возвращать его.

Задание 5

Напишите функцию sumInput(), которая:

- *Просит пользователя ввести значения, используя prompt и сохраняет их в массив.*
- *Заканчивает запрашивать значения, когда пользователь введёт не числовое значение, пустую строку или нажмёт «Отмена».*
- *Подсчитывает и возвращает сумму элементов массива.*

Ноль 0 считается числом, не останавливайте ввод значений при вводе «0».

Задание 6

Есть объект salaries с произвольным количеством свойств, содержащих заработные платы.

Напишите функцию sumSalaries(salaries), которая возвращает сумму всех зарплат с помощью метода Object.values и цикла for..of.

Если объект salaries пуст, то результат должен быть 0.

Задание 7

Создайте функцию `topSalary(salaries)`*, которая возвращает имя самого высокооплачиваемого сотрудника.*

- *Если объект* `salaries` *пустой, то нужно вернуть* `null`*.*
- *Если несколько высокооплачиваемых сотрудников, можно вернуть любого из них.*

P.S. Используйте `Object.entries` *и деструктурирование, чтобы перебрать пары ключ/значение.*

Задание 8

Напишите функцию `sum`*, которая работает таким образом:* `sum(a)(b) = a+b`*.*

Да, именно таким образом, используя двойные круглые скобки (не опечатка).

! Контрольные вопросы !

1. Методы массивов : sort, filter, foreach, map, reduce.
2. Что такое this?
3. Что такое глобальный объект?
4. Что такое замыкание?
5. Привязывание контекста : call, apply, bind. Различия между ними.
6. Что такое Object.asign, values, .keys?
7. Как работает деструктурирующее присваивание?
8. Опишите работу сборщика мусора в JavaScript.

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.