ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

УТВЕРЖДАЮ

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
лабораторно-практической работы №2

учебной дисциплины   
*МДК. 09.01 Проектирование и разработка веб-приложений*

*Тема: «Использование встроенных структур данных»*

г. Томск – 2021 г

РАССМОТРЕНО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБПОУ «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Грушевский Ю.В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика лабораторно-практической работы №2 4](#_heading=h.gjdgxs)

[2. Краткие теоретические сведения 5](#_heading=h.tyjcwt)

[3. Задания для выполнения 38](#_heading=h.111kx3o)

[4. Индивидуальные задания 42](#_heading=h.3l18frh)

[5. Контрольные вопросы 44](#_heading=h.2zbgiuw)

[6. Рекомендованная литература 45](#_heading=h.1egqt2p)

[ПРИЛОЖЕНИЕ А 46](#_heading=h.3ygebqi)

[ПРИЛОЖЕНИЕ Б 47](#_heading=h.2dlolyb)

# ОБЩАЯ ХАРАКТЕРИСТИКА ЛАБОРАТОРНО-ПРАКТИЧЕСКОЙ РАБОТЫ №2 Тема: «Использование встроенных структур данных»

**Цель работы:** формирование умений по использованию встроенных структур данных в языке JavaScript.

**Проверяемые компетенции:**

ПК 9.1, ПК 9.2, ПК 9.3, ПК 9.4, ПК 9.5, ОК 1, ОК 2, ОК 4, ОК 5, ОК 9, ОК 10.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы;
2. Выполните предложенные задания для выполнения;
3. Выполните предложенные индивидуальные задания в соответствии с вариантом;
4. Ответе на контрольные вопросы;
5. Оформите отчет по практической (шаблон представлен в приложении);
6. Отправьте отчет на проверку.

**Время выполнения заданий:** 10 часов.

**Критерии оценки:**

| **№** | **Критерий** | **Баллы** |
| --- | --- | --- |
|  | Отчет соответствует предложенному шаблону | 2 |
|  | Задания для выполнения решены верно (штраф 1 балл за каждое неверное решение) | 10 |
|  | Функция №1 из индивидуального задание реализована верно | 2 |
|  | Функция №2 из индивидуального задание реализована верно | 2 |
|  | Функция №3 из индивидуального задание реализована верно | 2 |
|  | Функция №4 из индивидуального задание реализована верно | 2 |
|  | Функция №5 из индивидуального задание реализована верно | 2 |
|  | Диаграммы для функций из индивидуального задания составлены верно (штраф 1 балл за каждое неверное решение) | 5 |
|  | Представлены ответы на контрольные вопросы (штраф 1 балл за каждый неправильный ответ) | 4 |
|  | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 2 |
|  | Своевременность выполнения задания | 2 |
| Итого | | 35 |

**Перевод в пятибалльную систему оценивания:**

«отлично»: > 25 баллов

«хорошо»: 21-25 баллов

«удовлетворительно»: 16-20 баллов

«неудовлетворительно»: < 15 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

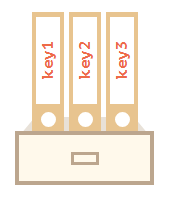
## Объекты

Как мы знаем, в JavaScript существует семь типов данных. Шесть 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. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

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

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо (Рисунок 1).



**Рисунок 1.** Визуализация объекта

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса (Листинг 1).

**Листинг 1** Создание объекта

**let *user*** = **new *Object***(); *// синтаксис "конструктор объекта"*

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



**Рисунок 2.** Создание 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*

};

Свойства объекта также иногда называют полями объекта.

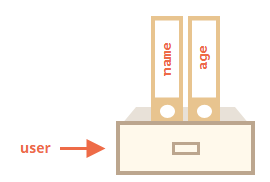
У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

Первое свойство с именем "name" и значением "John".

Второе свойство с именем "age" и значением 30.

Можно сказать, что наш объект user – это ящик с двумя папками, подписанными «name» и «age» (Рисунок 3)



**Рисунок 3.** Объект user

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»:

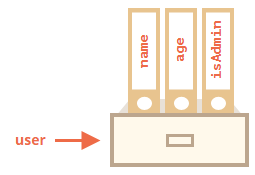
*// получаем свойства объекта:*

*alert*( user.**name** ); *// John*

*alert*( user.age ); *// 30*

Значение может быть любого типа. Давайте добавим свойство с логическим значением (Рисунок 4):

user.**isAdmin** = **true**;



**Рисунок 4.** Объект user. Задание нового свойства

Для удаления свойства мы можем использовать оператор delete:

**delete** user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки (Рисунок 5):

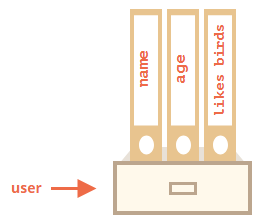
**let *user*** = {

**name**: **"John"**,

**age**: 30,

**"likes birds"**: **true** *// имя свойства из нескольких слов должно быть в кавычках*

};



**Рисунок 5.** Объект user. Свойство состоящее из нескольких слов

### 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**

Так происходит, потому что 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 (Листинг 2).

**Листинг 2** Обращение к свойствам с названиями состоящим из нескольких слов

**let *user*** = {};

*// присвоение 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 закавычена (подойдёт любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной (Листинг 3)

**Листинг 3** Обращение к свойствам через переменную

**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 дает нам большую гибкость (Листинг 4).

**Листинг 4** Пример использования переменных для обращения к свойствам объекта

**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")*

### Вычисляемые свойства

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства (Листинг 5).

**Листинг 5** Вычисляемое свойство

**let *fruit*** = *prompt*(**"Какой фрукт купить?"**, **"apple"**);

**let *bag*** = {

[***fruit***]: 5, *// имя свойства будет взято из переменной fruit*

};

*alert*( ***bag***.**apple** ); *// 5, если fruit="apple"*

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

И если посетитель введет слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

### Свойство из переменной

В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем (Листинг 6).

**Листинг 6** Свойство из переменной с тем же именем

**function** *makeUser*(name, age) {

**return** {

**name**: name,

**age**: age

*// ...другие свойства*

};

}

**let *user*** = *makeUser*(**"John"**, 30);

*alert*(***user***.**name**); *// John*

В примере выше название свойств name и age совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств. Такой подход настолько распространён, что существуют специальные *короткие свойства* для упрощения этой записи (Листинг 7).

**Листинг 7** Свойство из переменной с тем же именем. Сокращенная запись

**function** *makeUser*(name, age) {

**return** {

name, *// то же самое, что и name: name*

age *// то же самое, что и age: age*

*// ...*

};

}

### Проверка существования свойства

Особенность 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 (Листинг 8).

**Листинг 8** Обращение к несуществующему свойству

**let *user*** = {};

*alert*( ***user***.noSuchProperty === **undefined** ); *// true означает "свойства нет"*

Также существует специальный оператор "in" для проверки существования свойства в объекте (Листинг 9).

**Листинг 9** Синтаксис оператора in

**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.

### 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 (Листинг 10).

**Листинг 10** Перебор свойств объекта

**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.

### Копирование по ссылке

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

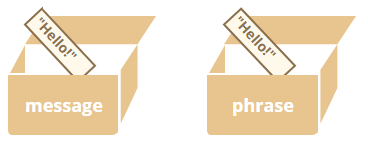
Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

Например:

**let *message*** = **"Hello!"**;

**let *phrase*** = ***message***;

В результате мы имеем две независимые переменные, каждая из которых хранит строку "Hello!" (Рисунок 6).



**Рисунок 6.** Объект user. Свойство состоящее из нескольких слов

Объекты ведут себя иначе.

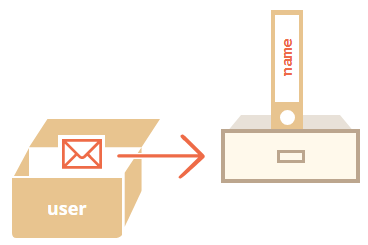
**Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него.**

Проиллюстрируем это (Рисунок 7):

**let *user*** = {

**name**: **"John"**

};



**Рисунок 7.** Организация данных в объекте

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

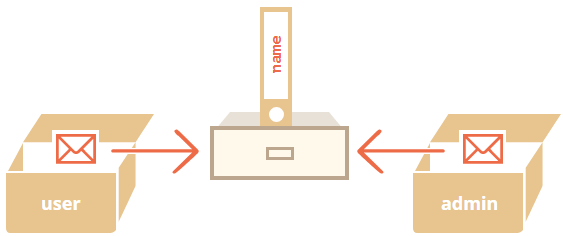
**Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется.**

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

**let *user*** = { **name**: **"John"** };

**let *admin*** = ***user***; *// копируется ссылка*

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект (Рисунок 8).



**Рисунок 8.** Копирование объекта

Мы можем использовать любую из переменных для доступа к ящику и изменения его содержимого (Листинг 11).

**Листинг 11** Доступ к объекту

**let *user*** = { **name**: **'John'** };

**let *admin*** = ***user***;

***admin***.**name** = **'Pete'**; *// изменено по ссылке из переменной "admin"*

*alert*(***user***.**name**); *// 'Pete', 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 объектов

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

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

### Клонирование и объединение объектов, Object.assign

Как мы узнали ранее, при копировании переменной объекта 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. На самом деле, такая нужда возникает редко. В большинстве случаев нам достаточно копирования по ссылке.

Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их (Листинг 3.12).

**Листинг 12** Копирование объекта

**let *user*** = {

**name**: **"John"**,

**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** = **"Pete"**; *// изменим в нём данные*

*alert*( ***user***.**name** ); *// в оригинальном объекте значение свойства `name` осталось прежним – John.*

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

Синтаксис:

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

* Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
* Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один (Листинг 13).

**Листинг 13** Использование assign

**let *user*** = { **name**: **"John"** };

**let *permissions1*** = { **canView**: **true** };

**let *permissions2*** = { **canEdit**: **true** };

*// копируем все свойства из permissions1 и permissions2 в user*

***Object***.assign(***user***, ***permissions1***, ***permissions2***);

*// now user = { name: "John", canView: true, canEdit: true }*

Мы также можем использовать Object.assign для простого клонирования (Листинг 14).

**Листинг 14** Использование assign для простого клонирования

**let *user*** = {

**name**: **"John"**,

**age**: 30

};

**let *clone*** = ***Object***.assign({}, ***user***);

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. На самом деле, такое клонирование работает так же, как и через цикл, но короче.

## Методы объекта, "this"

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

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

Такие действия в JavaScript представлены свойствами-функциями объекта.

### Примеры методов

Для начала давайте научим пользователя user здороваться (Листинг 15).

**Листинг 15** Пример метода

**let *user*** = {

**name**: **"Джон"**,

**age**: 30

};

***user***.sayHi = **function**() {

*alert*(**"Привет!"**);

};

***user***.sayHi(); *// Привет!*

Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили её свойству user.sayHi нашего объекта.

Затем мы вызвали её. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют *методом* этого объекта.

Итак, мы получили метод sayHi объекта user.

### Сокращённая запись метода

Существует более короткий синтаксис для методов в литерале объекта (Листинг 16).

**Листинг 16** Сокращенная запись метода

*// эти объекты делают одно и то же (одинаковые методы)*

**user** = {

sayHi: **function**() {

*alert*(**"Привет"**);

}

};

*// сокращённая запись выглядит лучше, не так ли?*

**user** = {

sayHi() { *// то же самое, что и "sayHi: function()"*

*alert*(**"Привет"**);

}

};

### Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user.

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.**

Значение this – это объект «перед точкой», который использовался для вызова метода (Листинг 17).

**Листинг 17** Использование this

**let *user*** = {

**name**: **"Джон"**,

**age**: 30,

sayHi() {

*// this - это "текущий объект"*

*alert*(**this**.**name**);

}

};

***user***.sayHi(); *// Джон*

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект) (Листинг 18)

**Листинг 18** Код без использование this

**let *user*** = {

**name**: **"Джон"**,

**age**: 30,

sayHi() {

*alert*(***user***.**name**); *// используем переменную "user" вместо ключевого слова "this"*

}

};

…Но такой код будет ненадежным. Если мы решим скопировать ссылку на объект user в другую переменную, например, admin = user, и перепишем переменную user чем-то другим, тогда будет осуществлен доступ к неправильному объекту при вызове метода из admin.

### «this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции (Листинг 19).

**Листинг 19** this в функции

**function** *sayHi*() {

*alert*( **this**.**name** );

}

Значение this вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах (Листинг 20).

**Листинг 20** this в зависимости от контекста

**let *user*** = { **name**: **"Джон"** };

**let *admin*** = { **name**: **"Админ"** };

**function** *sayHi*() {

*alert*( **this**.**name** );

}

*// используем одну и ту же функцию в двух объектах*

user.f = sayHi;

admin.f = sayHi;

*// вызовы функции, приведённые ниже, имеют разное значение this*

*// "this" внутри функции является ссылкой на объект, который указан "перед точкой"*

user.f(); *// Джон (this == user)*

admin.f(); *// Админ (this == admin)*

admin[**'f'**](); *// Админ (не важен способ доступа к методу - через точку или квадратные скобки)*

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

### У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы используем this внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Например, здесь arrow() использует значение this из внешнего метода user.sayHi() (Листинг 21).

**Листинг 21** this в стрелочной функции

**let *user*** = {

**firstName**: **"Илья"**,

sayHi() {

**let** *arrow* = () => *alert*(**this**.**firstName**);

*arrow*();

}

};

***user***.sayHi(); *// Илья*

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение this, а хотим брать его из внешнего контекста.

## Преобразование объектов в примитивы

Что произойдёт, если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj)?

В этом случае объекты сначала автоматически преобразуются в примитивы, а затем выполняется операция.

В главе Преобразование типов мы видели правила для числовых, строковых и логических преобразований. Но обделили вниманием объекты. Теперь, поскольку мы уже знаем о методах объектов и символах, можно исправить это упущение.

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда мы вычитаем объекты или выполняем математические операции. Например, объекты Date (мы рассмотрим их в статье Дата и время) могут вычитаться, и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, когда мы выводим объект alert(obj), а также в других случаях, когда объект используется как строка.

### Преобразование к примитивам

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в спецификации:

**"string"**

Для преобразования объекта к строке, когда операция ожидает получить строку, например alert (Листинг 21)

**Листинг 21** Преобразование объекта к строке

*// вывод*

*alert*(obj);

*// используем объект в качестве имени свойства*

anotherObj[obj] = 123;

**"number"**

Для преобразования объекта к числу, в случае математических операций (Листинг 22)

**Листинг 22** Преобразование объекта к числу

*// явное преобразование*

**let *num*** = ***Number***(obj);

*// математическое (исключая бинарный оператор "+")*

**let *n*** = +obj; *// унарный плюс*

**let *delta*** = date1 - date2;

*// сравнения больше/меньше*

**let *greater*** = user1 > user2;

**"default"**

Происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать с обоими типами: строками (объединять их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или когда происходит сравнение объектов с помощью нестрогого равенства == со строкой, числом или символом, и неясно, какое преобразование должно быть выполнено (Листинг 23).

**Листинг 23** Преобразование “default”

*// бинарный плюс*

**let *total*** = car1 + car2;

*// obj == string/number/symbol*

**if** (user == 1) { ... };

Оператор больше/меньше <> также может работать как со строками, так и с числами. Однако, по историческим причинам он использует хинт «number», а не «default».

На практике все встроенные объекты, исключая Date (мы познакомимся с ним чуть позже), реализуют "default" преобразования тем же способом, что и "number". И нам следует поступать также.

Обратите внимание, что существуют лишь три варианта хинтов. Всё настолько просто. Не существует хинта со значением «boolean» (все объекты являются true в логическом контексте) или каких-либо ещё. И если мы считаем "default" и "number" одинаковыми, как большинство встроенных объектов, то остаются всего два варианта преобразований.

**В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:**

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен "string"
   1. пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он существует.
3. В случае, если хинт равен "number" или "default"
   1. пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он существует.

### Symbol.toPrimitive

Начнём с универсального подхода – символа Symbol.toPrimitive: метод с таким названием (если есть) используется для всех преобразований:

obj[***Symbol***.**toPrimitive**] = **function**(hint) {

*// должен вернуть примитивное значение*

*// hint равно чему-то одному из: "string", "number" или "default"*

};

Для примера используем его в реализации объекта user (Листинг 24):

**Листинг 24** Использование Symbol.toPrimitive

**let *user*** = {

**name**: **"John"**,

**money**: 1000,

[***Symbol***.**toPrimitive**](hint) {

*alert*(**`hint:** ${hint}**`**);

**return** hint == **"string"** ? **`{name: "**${**this**.**name**}**"}`** : **this**.**money**;

}

};

*// демонстрация результатов преобразований:*

*alert*(***user***); *// hint: string -> {name: "John"}*

*alert*(+***user***); *// hint: number -> 1000*

*alert*(***user*** + 500); *// hint: default -> 1500*

Как мы видим из кода, user преобразовывается либо в информативную читаемую строку, либо в денежный счёт в зависимости от значения хинта. Единственный метод user[Symbol.toPrimitive] смог обработать все случаи преобразований.

### Методы toString/valueOf

Методы toString и valueOf берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода Symbol.toPrimitive, движок JavaScript пытается найти эти методы и вызвать их следующим образом:

* toString -> valueOf для хинта со значением «string».
* valueOf -> toString – в ином случае.

Для примера, используем их в реализации всё того же объекта user. Воспроизведём его поведение комбинацией методов toString и valueOf (Листинг 25)

**Листинг 25** Использование toString и valueOf

**let *user*** = {

**name**: **"John"**,

**money**: 1000,

*// для хинта равного "string"*

toString() {

**return `{name: "**${**this**.**name**}**"}`**;

},

*// для хинта равного "number" или "default"*

valueOf() {

**return this**.**money**;

}

};

*alert*(***user***); *// toString -> {name: "John"}*

*alert*(+***user***); *// valueOf -> 1000*

*alert*(***user*** + 500); *// valueOf -> 1500*

Как видим, получилось то же поведение, что и в предыдущем примере с Symbol.toPrimitive.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один toString (Листинг 26)

**Листинг 26** Использование одного toString

**let *user*** = {

**name**: **"John"**,

toString() {

**return this**.**name**;

}

};

*alert*(***user***); *// toString -> John*

*alert*(***user*** + 500); *// toString -> John500*

В отсутствие Symbol.toPrimitive и valueOf, toString обработает все случаи преобразований к примитивам.

### Возвращаемые типы

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно требуемый «хинтом» тип примитива.

Нет обязательного требования, чтобы toString() возвращал именно строку, или чтобы метод Symbol.toPrimitive возвращал именно число для хинта «number».

**Единственное обязательное требование: методы должны возвращать примитив, а не объект.**

## 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".

Например (Листинг 27):

**Листинг 27** Использование одного toString

**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.

То есть, результат вызова new User("Вася") – это тот же объект, что и:

**let *user*** = {

**name**: **"Вася"**,

**isAdmin**: **false**

};

Теперь, когда нам необходимо будет создать других пользователей, мы можем использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new, и выполнится алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором, и её нужно вызывать через new.

### Создание методов в конструкторе

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В this мы можем добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi (Листинг 28):

**Листинг 28** Создание методов в конструкторе

**function** *User*(name) {

**this**.**name** = name;

**this**.sayHi = **function**() {

*alert*( **"Меня зовут: "** + **this**.**name** );

};

}

**let *vasya*** = **new** *User*(**"Вася"**);

***vasya***.sayHi(); *// Меня зовут: Вася*

*/\**

*vasya = {*

*name: "Вася",*

*sayHi: function() { ... }*

*}*

*\*/*

## Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто мы понимаем, что нам необходима упорядоченная коллекция данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

### Объявление

Существует два варианта синтаксиса для создания пустого массива:

**let *arr*** = **new *Array***();

**let *arr*** = [];

Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках (Листинг 29):

**Листинг 29** Обращение к элементам в массиве

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits***[0] ); *// Яблоко*

*alert*( ***fruits***[1] ); *// Апельсин*

*alert*( ***fruits***[2] ); *// Слива*

Мы можем заменить элемент:

fruits[2] = **'Груша'**; *// теперь ["Яблоко", "Апельсин", "Груша"]*

…Или добавить новый к существующему массиву:

fruits[3] = **'Лимон'**; *// теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]*

Общее число элементов массива содержится в его свойстве length (Листинг 30):

**Листинг 30** Получение количества элементов в массиве

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits***.**length** ); *// 3*

Вывести массив целиком можно при помощи alert (Листинг 31).

**Листинг 31** Вывод элементов массива

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits*** ); *// Яблоко, Апельсин, Слива*

В массиве могут храниться элементы любого типа (Листинг 32).

**Листинг 32** Хранение в массиве элементов разного типа

*// разные типы значений*

**let *arr*** = [ **'Яблоко'**, { **name**: **'Джон'** }, **true**, **function**() { *alert*(**'привет'**); } ];

*// получить элемент с индексом 1 (объект) и затем показать его свойство*

*alert*( ***arr***[1].**name** ); *// Джон*

*// получить элемент с индексом 3 (функция) и выполнить её*

***arr***[3](); *// привет*

### Методы pop/push, shift/unshift

*Очередь* – один из самых распространенных вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

* push добавляет элемент в конец.
* shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.



**Рисунок 9.** Визуализация очереди

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

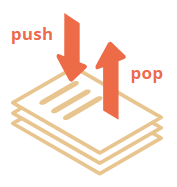
Существует и другой вариант применения для массивов – структура данных, называемая *стек*.

Она поддерживает два вида операций:

* push добавляет элемент в конец.
* pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху (Рисунок 10)



**Рисунок 10.** Визуализация стека

Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется *двусторонняя очередь*.

**Методы, работающие с концом массива:**

**pop**

Удаляет последний элемент из массива и возвращает его (Листинг 33).

**Листинг 33** Метод pop

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

*alert*( ***fruits***.pop() ); *// удаляем "Груша" и выводим его*

*alert*( ***fruits*** ); *// Яблоко, Апельсин*

**push**

Добавляет элемент в конец массива (Листинг 34).

**Листинг 34** Метод push

***let fruits*** *= [****"Яблоко"****,* ***"Апельсин"****];*

***fruits****.push(****"Груша"****);*

*alert(* ***fruits*** *); // Яблоко, Апельсин, Груша*

**Методы, работающие с началом массива:**

**shift**

Удаляет из массива первый элемент и возвращает его (Листинг 35).

**Листинг 35** Метод shift

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

*alert*( ***fruits***.shift() ); *// удаляем Яблоко и выводим его*

*alert*( ***fruits*** ); *// Апельсин, Груша*

**unshift**

Добавляет элемент в начало массива (Листинг 36).

**Листинг 36** Метод unshift

**let *fruits*** = [**"Апельсин"**, **"Груша"**];

***fruits***.unshift(**'Яблоко'**);

*alert*( ***fruits*** ); *// Яблоко, Апельсин, Груша*

### Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам (Листинг 37).

**Листинг 37** Перебор массива с помощью цикла for

**let *arr*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

**for** (**let *i*** = 0; ***i*** < ***arr***.**length**; ***i***++) {

*alert*( ***arr***[***i***] );

}

Но для массивов возможен и другой вариант цикла, for..of (Листинг 38).

**Листинг 4.38** Перебор массива с помощью цикла for … of

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*// проходит по значениям*

**for** (**let *fruit* of *fruits***) {

*alert*( ***fruit*** );

}

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

### Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц (Листинг 39).

**Листинг 39** Многомерные массивы

**let *matrix*** = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

*alert*( ***matrix***[1][1] ); *// 5, центральный элемент*

### toString

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделенных запятыми (Листинг 40).

**Листинг 40** Реализация метода toString в массивах

**let *arr*** = [1, 2, 3];

*alert*( ***arr*** ); *// 1,2,3*

*alert*( ***String***(***arr***) === **'1,2,3'** ); *// true*

## Методы массивов

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

### Добавление/удаление элементов

#### splice

Метод arr.splice(str) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

*arr.splice(index[, deleteCount, elem1, ..., elemN])*

Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удаленных элементов.

Этот метод проще всего понять, рассмотрев примеры.

Начнем с удаления (Листинг 41).

**Листинг 41** Удаление элементов с помощью метода splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**];

***arr***.splice(1, 1); *// начиная с позиции 1, удалить 1 элемент*

*alert*( ***arr*** ); *// осталось ["Я", "JavaScript"]*

Легко, правда? Начиная с позиции 1, он убрал 1 элемент.

В следующем примере мы удалим 3 элемента и заменим их двумя другими (Листинг 42).

**Листинг 42** Замена элементов с помощью метода splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**, **"прямо"**, **"сейчас"**];

*// удалить 3 первых элемента и заменить их другими*

***arr***.splice(0, 3, **"Давай"**, **"танцевать"**);

*alert*( ***arr*** ) *// теперь ["Давай", "танцевать", "прямо", "сейчас"]*

Здесь видно, что splice возвращает массив из удалённых элементов (Листинг 43).

**Листинг 43** Возвращение методом splice удаленных элементов

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**, **"прямо"**, **"сейчас"**];

*// удалить 2 первых элемента*

**let *removed*** = ***arr***.splice(0, 2);

*alert*( ***removed*** ); *// "Я", "изучаю" <-- массив из удалённых элементов*

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0 (Листинг 44).

**Листинг 44** Вставка элементов методом splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**];

*// с позиции 2*

*// удалить 0 элементов*

*// вставить "сложный", "язык"*

***arr***.splice(2, 0, **"сложный"**, **"язык"**);

*alert*( ***arr*** ); *// "Я", "изучаю", "сложный", "язык", "JavaScript"*

#### slice

Метод arr.slice намного проще, чем похожий на него arr.splice.

Его синтаксис:

*arr.slice([start], [end])*

Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчет будет осуществляться с конца массива.

Можно вызвать slice и вообще без аргументов: arr.slice() создает копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

#### concat

Метод arr.concat создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

*arr.concat(arg1, arg2...)*

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.

В результате мы получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и так далее…

Если аргумент argN – массив, то все его элементы копируются. Иначе скопируется сам аргумент (Листинг 45).

**Листинг 45** Метод concat

**let *arr*** = [1, 2];

*// создать массив из: arr и [3,4]*

*alert*( ***arr***.concat([3, 4]) ); *// 1,2,3,4*

*// создать массив из: arr и [3,4] и [5,6]*

*alert*( ***arr***.concat([3, 4], [5, 6]) ); *// 1,2,3,4,5,6*

*// создать массив из: arr и [3,4], потом добавить значения 5 и 6*

*alert*( ***arr***.concat([3, 4], 5, 6) ); *// 1,2,3,4,5,6*

### Перебор: forEach

Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

*arr.forEach(function(item, index, array) {*

*// ... делать что-то с item*

*});*

Например, этот код выведет на экран каждый элемент массива (Листинг 46).

**Листинг 46** Метод foreach

*// Вызов alert для каждого элемента*

[**"Bilbo"**, **"Gandalf"**, **"Nazgul"**].forEach(*alert*);

### Поиск в массиве

Далее рассмотрим методы, которые помогут найти что-нибудь в массиве.

#### indexOf/lastIndexOf и includes

Методы arr.indexOf, arr.lastIndexOf и arr.includes имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

* arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
* arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например (Листинг 47):

**Листинг 47** Поиск элементов в массиве

**let *arr*** = [1, 0, **false**];

*alert*( ***arr***.indexOf(0) ); *// 1*

*alert*( ***arr***.indexOf(**false**) ); *// 2*

*alert*( ***arr***.indexOf(**null**) ); *// -1*

*alert*( ***arr***.includes(1) ); *// true*

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если мы ищем false, он находит именно false, а не ноль.

Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf.

#### find и findIndex

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод arr.find.

Его синтаксис таков:

*let result = arr.find(function(item, index, array) {*

*// если true - возвращается текущий элемент и перебор прерывается*

*// если все итерации оказались ложными, возвращается undefined*

*});*

Функция вызывается по очереди для каждого элемента массива:

* item – очередной элемент.
* index – его индекс.
* array – сам массив.

Если функция возвращает true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined.

Например, у нас есть массив пользователей, каждый из которых имеет поля id и name. Попробуем найти того, кто с id == 1 (Листинг 48):

**Листинг 48** Использование метода find

**let *users*** = [

{**id**: 1, **name**: **"Вася"**},

{**id**: 2, **name**: **"Петя"**},

{**id**: 3, **name**: **"Маша"**}

];

**let *user*** = ***users***.find(item => item.**id** == 1);

*alert*(***user***.**name**); *// Вася*

В реальной жизни массивы объектов – обычное дело, поэтому метод find крайне полезен.

Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод arr.findIndex – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

#### filter

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true.

На тот случай, если найденных элементов может быть много, предусмотрен метод arr.filter(fn).

Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:

*let results = arr.filter(function(item, index, array) {*

*// если true - элемент добавляется к результату, и перебор продолжается*

*// возвращается пустой массив в случае, если ничего не найдено*

*});*

Пример (Листинг 49)

**Листинг 49** Использование метода filter

**let *users*** = [

{**id**: 1, **name**: **"Вася"**},

{**id**: 2, **name**: **"Петя"**},

{**id**: 3, **name**: **"Маша"**}

];

*// возвращает массив, состоящий из двух первых пользователей*

**let *someUsers*** = ***users***.filter(item => item.**id** < 3);

*alert*(***someUsers***.**length**); *// 2*

### Преобразование массива

Перейдём к методам преобразования и упорядочения массива.

#### map

Метод arr.map является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

*let result = arr.map(function(item, index, array) {*

*// возвращается новое значение вместо элемента*

*});*

Например, здесь мы преобразуем каждый элемент в его длину (Листинг 50):

**Листинг 50** Использование метода map

**let *lengths*** = [**"Bilbo"**, **"Gandalf"**, **"Nazgul"**].map(item => item.**length**);

*alert*(***lengths***); *// 5,7,6*

#### sort(fn)

Вызов arr.sort() сортирует массив на месте, меняя в нём порядок элементов.

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr (Листинг 51).

**Листинг 51** Использование метода sort

**let *arr*** = [ 1, 2, 15 ];

*// метод сортирует содержимое arr*

***arr***.sort();

*alert*( ***arr*** ); *// 1, 15, 2*

Не заметили ничего странного в этом примере?

Порядок стал 1, 15, 2. Это неправильно! Но почему?

**По умолчанию элементы сортируются как строки.**

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок, и действительно выходит, что "2" > "15".

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort().

Функция должна для пары значений возвращать:

**function** *compare*(a, b) {

**if** (a > b) **return** 1; *// если первое значение больше второго*

**if** (a == b) **return** 0; *// если равны*

**if** (a < b) **return** -1; *// если первое значение меньше второго*

}

Например, для сортировки чисел код будет выглядеть так (Листинг 52).

**Листинг 52** Использование своей функции сортировки

**function** *compareNumeric*(a, b) {

**if** (a > b) **return** 1;

**if** (a == b) **return** 0;

**if** (a < b) **return** -1;

}

**let *arr*** = [ 1, 2, 15 ];

***arr***.sort(*compareNumeric*);

*alert*(***arr***); *// 1, 2, 15*

Теперь всё работает как надо.

Давайте возьмём паузу и подумаем, что же происходит. Упомянутый ранее массив arr может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. У нас есть набор каких-то элементов. Чтобы отсортировать его, нам нужна функция, определяющая порядок, которая знает, как сравнивать его элементы. По умолчанию элементы сортируются как строки.

Метод arr.sort(fn) реализует общий алгоритм сортировки. Нам не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная быстрая сортировка). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся нам, это предоставить fn, которая делает это сравнение.

#### reverse

Метод arr.reverse меняет порядок элементов в arr на обратный.

Например (Листинг 53):

**Листинг 53** Использование функции reverse

**let *arr*** = [1, 2, 3, 4, 5];

***arr***.reverse();

*alert*( ***arr*** ); *// 5,4,3,2,1*

Он также возвращает массив arr с изменённым порядком элементов.

#### split и join

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод str.split(delim) именно это и делает. Он разбивает строку на массив по заданному разделителю delim.

В примере ниже таким разделителем является строка из запятой и пробела (Листинг 54).

**Листинг 54** Использование функции split

**let *names*** = **'Вася, Петя, Маша'**;

**let *arr*** = ***names***.split(**', '**);

**for** (**let *name* of *arr***) {

*alert*( **`Сообщение получат:** ${***name***}**.`** ); *// Сообщение получат: Вася (и другие имена)*

}

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен.

Вызов arr.join(glue) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними (Листинг 55).

**Листинг 55** Использование функции join

**let *arr*** = [**'Вася'**, **'Петя'**, **'Маша'**];

**let *str*** = ***arr***.join(**';'**); *// объединить массив в строку через ;*

*alert*( ***str*** ); *// Вася;Петя;Маша*

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Напишите код, выполнив задание из каждого пункта отдельной строкой:
   1. Создайте пустой объект user.
   2. Добавьте свойство name со значением John.
   3. Добавьте свойство surname со значением Smith.
   4. Измените значение свойства name на Pete.
   5. Удалите свойство name из объекта.
2. Напишите функцию isEmpty(obj), которая возвращает true, если у объекта нет свойств, иначе false.

Должно работать так:

**let *schedule*** = {};

*alert*( isEmpty(***schedule***) ); *// true*

***schedule***[**"8:30"**] = **"get up"**;

*alert*( isEmpty(***schedule***) ); *// false*

1. Создайте объект calculator (калькулятор) с тремя методами:
   1. read() (читать) запрашивает два значения и сохраняет их как свойства объекта.
   2. sum() (суммировать) возвращает сумму сохранённых значений.
   3. mul() (умножить) перемножает сохраненные значения и возвращает результат.

**let *calculator*** = {

*// ... ваш код ...*

};

***calculator***.read();

*alert*( ***calculator***.sum() );

*alert*( ***calculator***.mul() );

1. Это ladder (лестница) – объект, который позволяет подниматься вверх и спускаться:

**let *ladder*** = {

**step**: 0,

up() {

**this**.**step**++;

},

down() {

**this**.**step**--;

},

showStep: **function**() { *// показывает текущую ступеньку*

*alert*( **this**.**step** );

}

};

Теперь, если нам нужно сделать несколько последовательных вызовов, мы можем выполнить это так:

ladder.up();

ladder.up();

ladder.down();

ladder.showStep(); *// 1*

Измените код методов up, down и showStep таким образом, чтобы их вызов можно было сделать по цепочке, например так:

ladder.up().up().down().showStep(); *// 1*

Такой подход широко используется в библиотеках JavaScript.

1. Создайте функцию-конструктор Calculator, который создаёт объекты с тремя методами:
   1. read() запрашивает два значения при помощи prompt и сохраняет их значение в свойствах объекта.
   2. sum() возвращает сумму введённых свойств.
   3. mul() возвращает произведение введённых свойств.

Например:

**let *calculator*** = **new** Calculator();

***calculator***.read();

*alert*( **"Sum="** + ***calculator***.sum() );

*alert*( **"Mul="** + ***calculator***.mul() );

1. 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.

Ниже вы можете посмотреть работу кода:

**let *accumulator*** = **new** Accumulator(1); *// начальное значение 1*

***accumulator***.read(); *// прибавит ввод prompt к текущему значению*

***accumulator***.read(); *// прибавит ввод prompt к текущему значению*

*alert*(***accumulator***.**value**); *// выведет сумму этих значений*

1. Напишите функцию sumInput(), которая:
   1. Просит пользователя ввести значения, используя prompt и сохраняет их в массив.
   2. Заканчивает запрашивать значения, когда пользователь введёт не числовое значение, пустую строку или нажмёт «Отмена».
   3. Подсчитывает и возвращает сумму элементов массива.

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

1. На входе массив чисел, например: arr = [1, -2, 3, 4, -9, 6].

Задача: найти непрерывный подмассив в arr, сумма элементов в котором максимальна.

Функция getMaxSubSum(arr) должна возвращать эту сумму.

Например:

getMaxSubSum([-1, 2, 3, -9]) = 5 (сумма выделенных)

getMaxSubSum([2, -1, 2, 3, -9]) = 6

getMaxSubSum([-1, 2, 3, -9, 11]) = 11

getMaxSubSum([-2, -1, 1, 2]) = 3

getMaxSubSum([100, -9, 2, -3, 5]) = 100

getMaxSubSum([1, 2, 3]) = 6 (берём все)

Если все элементы отрицательные – ничего не берём(подмассив пустой) и сумма равна «0».

1. У нас есть массив строк arr. Нужно получить отсортированную копию, но оставить arr неизменённым.

Создайте функцию copySorted(arr), которая будет возвращать такую копию.

**let *arr*** = [**"HTML"**, **"JavaScript"**, **"CSS"**];

**let *sorted*** = copySorted(***arr***);

*alert*( ***sorted*** ); *// CSS, HTML, JavaScript*

*alert*( ***arr*** ); *// HTML, JavaScript, CSS (без изменений)*

1. Создайте функцию конструктор Calculator, которая создаёт «расширяемые» объекты калькулятора.

Задание состоит из двух частей.

1. Во-первых, реализуйте метод calculate(str), который принимает строку типа "1 + 2" в формате «ЧИСЛО оператор ЧИСЛО» (разделено пробелами) и возвращает результат. Метод должен понимать плюс + и минус -.

Пример использования:

**let *calc*** = **new** Calculator;

*alert*( ***calc***.calculate(**"3 + 7"**) ); *// 10*

1. Затем добавьте метод addMethod(name, func), который добавляет в калькулятор новые операции. Он принимает оператор name и функцию с двумя аргументами func(a,b), которая описывает его.

Например, давайте добавим умножение \*, деление / и возведение в степень \*\*:

**let *powerCalc*** = **new** Calculator;

***powerCalc***.addMethod(**"\*"**, (a, b) => a \* b);

***powerCalc***.addMethod(**"/"**, (a, b) => a / b);

***powerCalc***.addMethod(**"\*\*"**, (a, b) => a \*\* b);

**let *result*** = ***powerCalc***.calculate(**"2 \*\* 3"**);

*alert*( ***result*** ); *// 8*

Для этой задачи не нужны скобки или сложные выражения.

Числа и оператор разделены ровно одним пробелом.

Не лишним будет добавить обработку ошибок.

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

Выполните предложенное ниже задание согласно своему варианту, добавьте в **отчет** **отчет диаграммы активностей (Activity diagram)** всех алгоритмов, **код** и **результат работы**.

### Обобщенное задание

Выбрать предметную область для базы данных и предложить структуру объектов для описания отдельных записей базы данных. Выбранная структура должна иметь не менее пяти свойств двух и более типов. Для выбранной базы данных написать следующие функции:

1. Функцию формирования массива объектов, значения свойств которых вводятся с клавиатуры. При вводе объектов можно реализовать один из следующих механизмов:

* ввод заранее заданного количества объектов;
* ввод до появления объекта с заданным признаком;
* диалог с пользователем о необходимости продолжать ввод.

1. Функцию просмотра содержимого массива объектов.
2. Функцию дополнения уже существующего массива новыми объектами.
3. Функцию поиска и вывода на экран объекта (объектов) с заданным значением свойства.
4. Функцию упорядочения массива структур по заданному свойству.

### Варианты

1. «Человек»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира).
2. «Школьник»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); школа; класс.
3. «Студент»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); ВУЗ; курс; группа; средний балл; специальность.
4. «Покупатель»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); номер кредитной карточки; банковского счета.
5. «Пациент»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); номер больницы; отделение; номер медицинской карты; диагноз; группа крови.
6. «Владелец автомобиля»: фамилия; имя; отчество; номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира) марка автомобиля; номер автомобиля; номер техпаспорта.
7. «Военнослужащий»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); национальность; дата рождения (год, месяц число); должность; звание.
8. «Рабочий»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); национальность; дата рождения (год, месяц число); № цеха; табельный номер; образование; год поступления на работу.
9. «Владелец телефона»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); № телефона.
10. «Абитуриент»: фамилия; имя; отчество; пол; национальность; дата рождения (год, месяц число); домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); оценки по экзаменам; проходной балл.
11. «Государство»: название страны; столица; государственный язык; население; площадь территории; денежная единица; государственный строй; глава государства.
12. «Автомобиль»: марка; цвет; серийный номер; регистрационный номер; год выпуска; год техосмотра; цена.
13. «Рейс»: марка автомобиля; номер автомобиля; пункт назначения; грузоподъемность (в тоннах); стоимость единицы груза; общая стоимость груза.
14. «Книга»: название; автор (фамилия; имя); год выхода; издательство; себестоимость; цена; прибыль.
15. «Здание»: адрес; тип здания; количество этажей; количество квартир; срок эксплуатации; срок до капитального ремонта (25 лет - срок эксплуатации).

# КОНТРОЛЬНЫЕ ВОПРОСЫ

* Что такое объект? Приведите примеры.
* Чем метод объекта отличается от свойства?
* Приведите пример использования вычисляемого свойства.
* Что такое this?
* Что делает команда new? Приведите примеры.
* Чем массивы отличаются от объектов?
* Какие методы используются для добавления и удаления элементов в массив?
* Для чего используется метод forEach? Приведите пример.

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Разработка модулей программного обеспечения для компьютерных систем: учебник для студ. учреждений сред. проф. образования/Г.Н.Федорова. – М.: Издательский центр «Академия», 2017 – 384 с. ISBN 978-5-4468-6104-0 http://www.academia-moscow.ru/reader/?id=301421
2. Проектирование и разработка информационных систем: учебник для студ. учреждений сред. проф. образования/О.Н.Перлова, О.П.Ляпина, А.В.Гусева. – М.: Издательский центр «Академия», 2017 – 256 с. ISBN 978-5-4468-6208-5 http://www.academia-moscow.ru/reader/?id=304708
3. Современный учебник JavaScript [Электронный ресурс] – Режим доступа: https://learn.javascript.ru Дата обращения: 10.02.2020
4. Cправочники и учебные курсы по различным веб-технологиям, а также руководства по вёрстке веб-страниц и разработке мобильных приложений [Электронный ресурс] – Режим доступа: https://webref.ru Дата обращения: 10.02.2020
5. The World Wide Web Consortium (W3C) [Электронный ресурс] – Режим доступа: https://www.w3.org Дата обращения: 10.02.2020
6. JavaScript, современный стандарт ES5 (англ). [Электронный ресурс] – режим доступа: http://www.ecma-international.org/publications/standards/Ecma-262.htm Дата обращения: 10.02.2020

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Код решенных заданий для выполнения
   2. Диаграммы всех алгоритмов, код и результат работы скриптов индивидуальных заданий
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по лабораторно-практической работе №2

учебной дисциплины   
*МДК. 09.01 Проектирование и разработка веб-приложений*

*Тема: «Использование встроенных структур данных»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель, высшая к.к.

Грушевский Ю.В.

г. Томск – 2021 г