**Лабораторная работа №3. Структуры данных**

*Методы у примитивов. Числа. Строки. Массивы. Методы массивов. Объекты.*

**Задание:** изучите теорию ирешите задачи.

**Задачи:**

1. Пользователь вводит данные. Если он ввел число, то преобразуйте его в 16-ричную систему счисления (вывод в верхнем регистре). Если число дробное – округлите его до целого. Если пользователь ввел текст, то преобразуйте его к верхнему и нижнему регистру.
2. На сайте есть галерея. Ширина блока галереи 940px. При добавлении фотографии пользователь указывает ее ширину в пикселях. Блок галереи не имеет внутренних отступов, каждая фотография имеет внешние отступы по 5px. Исключены левый отступ у левого крайнего изображения и правый – у правого крайнего. Сколько фотографий поместить в ряд и какой размер отступов должен быть, если отступы должны быть одинаковыми?
3. Пользователь ввел число. Округлите его до наибольшего, наименьшего и ближайшего целого.
4. Пользователь хочет изменить цвет фона, для этого он вводит цвет в формате RGB. Преобразуйте его в 16-ричную систему и предложите пользователю контрастный цвет текста (черный или белый).
5. Выпускник сдает ЦТ по русскому языку. Ему дано словарное слово, необходимо ввести в текстовое поле правильный вариант ответа. Проверьте его ответ и сообщите в каком символе он допустил ошибку, если она есть.
6. Разработайте геометрический калькулятор для расчета параметров треугольника: площадь, периметр, высота, медиана, биссектриса, радиусы вписанной и описанной окружностей, cos, sin, tg, ctg. Пользователь указывает длину катетов.

**Теория**

1. **Математические методы**

JavaScript предоставляет базовые тригонометрические и некоторые другие функции для работы с числами.

*[Тригонометрия](https://learn.javascript.ru/number" \l "trigonometriya):*

Math.acos(x) – Возвращает арккосинус x (в радианах)

Math.asin(x) – Возвращает арксинус x (в радианах)

Math.atan(x)–Возвращает арктангенс x (в радианах)

Math.atan2(y, x)–Возвращает угол до точки (y, x). Описание функции: [Atan2](http://en.wikipedia.org/wiki/Atan2).

Math.sin(x)–Вычисляет синус x

Math.cos(x)–Вычисляет косинус x

Math.tan(x)–Возвращает тангенс x

[*Функции общего назначения*](https://learn.javascript.ru/number#funktsii-obschego-naznacheniya)*:*

Math.sqrt(x)–Возвращает квадратный корень из x.

Math.log(x)–Возвращает натуральный (по основанию e) логарифм x.

Math.pow(x, exp)–Возводит число в степень, возвращает xexp, например Math.pow(2,3) = 8. Работает в том числе с дробными и отрицательными степенями, например: Math.pow(4, -1/2) = 0.5.

Math.abs(x)–Возвращает абсолютное значение числа

Math.exp(x)–Возвращает ex, где e – основание натуральных логарифмов.

Math.max(a, b, c...)–Возвращает наибольший из списка аргументов

Math.min(a, b, c...)–Возвращает наименьший из списка аргументов

Math.random()–Возвращает псевдослучайное число в интервале [0,1) – то есть между 0 (включительно) и 1 (не включая). Генератор случайных чисел инициализируется текущим временем.

1. **Формат JSON**

Допустим, у нас есть сложный объект, и мы хотели бы преобразовать его в строку, чтобы отправить по сети. Естественно, такая строка должна включать в себя все важные свойства. Мы могли бы реализовать преобразование следующим образом:

let user = {

name: "John",

age: 30,

toString() {

return `{name: "${this.name}", age: ${this.age}}`;

}

};

alert(user); // {name: "John", age: 30}

Но в процессе разработки добавляются новые свойства, старые свойства переименовываются и удаляются. Обновление такого toString каждый раз может стать проблемой. Мы могли бы попытаться перебрать свойства в нем, но что, если объект является сложным и имеет вложенные объекты в свойствах? Мы должны были бы также осуществить их преобразование.

К счастью, нет необходимости писать код для обработки всего этого. У задачи есть простое решение.

**[JSON.stringify](https://learn.javascript.ru/json" \l "json-stringify)**

[JSON](https://ru.wikipedia.org/wiki/JSON) (JavaScript Object Notation) – это общий формат для представления значений и объектов. Его описание задокументировано в стандарте [RFC 4627](http://tools.ietf.org/html/rfc4627). Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java/любой другой язык.

JavaScript предоставляет методы:

* JSON.stringify для преобразования объектов в JSON.
* JSON.parse для преобразования JSON обратно в объект.

Например, здесь мы преобразуем через JSON.stringify данные студента:

let student = {

name: 'John',

age: 30,

isAdmin: false,

courses: ['html', 'css', 'js'],

wife: null

};

let json = JSON.stringify(student);

alert(typeof json); // мы получили строку!

alert(json);

/\* выведет объект в формате JSON:

{

"name": "John",

"age": 30,

"isAdmin": false,

"courses": ["html", "css", "js"],

"wife": null

}

\*/

Метод JSON.stringify(student) берет объект и преобразует его в строку. Полученная строка json называется JSON-форматированным или сериализованным объектом. Мы можем отправить его по сети или поместить в обычное хранилище данных.Обратите внимание, что объект в формате JSON имеет несколько важных отличий от объектного литерала:

* Строки используют двойные кавычки. Никаких одинарных кавычек или обратных кавычек в JSON. Так 'John'становится "John".
* Имена свойств объекта также заключаются в двойные кавычки. Это обязательно. Так age:30 становится "age":30.

JSON.stringify может быть применён и к примитивам.

JSON поддерживает следующие типы данных:

* Объекты { ... }
* Массивы [ ... ]
* Примитивы:
* строки,
* числа,
* логические значения true/false,
* null.

Например:

// число в JSON остаётся числом

alert( JSON.stringify(1) ) // 1

// строка в JSON по-прежнему остаётся строкой, но в двойных кавычках

alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]

JSON является независимой от языка спецификацией для данных, поэтому JSON.stringify пропускает некоторые специфические свойства объектов JavaScript.

А именно:

* Свойства-функции (методы).
* Символьные свойства.
* Свойства, содержащие undefined.

let user = {

sayHi() { // будет пропущено

alert("Hello");

},

[Symbol("id")]: 123, // также будет пропущено

something: undefined // как и это - пропущено

};

alert( JSON.stringify(user) ); // {} (пустой объект)

Обычно это нормально, но можно улучшить. Вложенные объекты поддерживаются и конвертируются автоматически.

Например:

let meetup = {

title: "Conference",

room: {

number: 23,

participants: ["john", "ann"]

}

};

alert( JSON.stringify(meetup) );

/\* вся структура преобразована в строку:

{

"title":"Conference",

"room":{"number":23,"participants":["john","ann"]},

}

\*/

Важное ограничение: не должно быть циклических ссылок. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: ["john", "ann"]

};

meetup.place = room; // meetup ссылается на room

room.occupiedBy = meetup; // room ссылается на meetup

JSON.stringify(meetup); // Ошибка: Преобразование цикличной структуры в JSON

Здесь преобразование завершается неудачно из-за циклической ссылки: room.occupiedBy ссылается на meetup, и meetup.place ссылается на room:



**[Исключаем и преобразуем: replacer](https://learn.javascript.ru/json" \l "isklyuchaem-i-preobrazuem-replacer)**

Полный синтаксис JSON.stringify:

let json = JSON.stringify(value[, replacer, space])

**value**

Значение для кодирования.

**replacer**

Массив свойств для кодирования или функция соответствия function(key, value).

**space**

Дополнительное пространство (отступы), используемое для форматирования.

В большинстве случаев JSON.stringify используется только с первым аргументом. Но если нам нужно настроить процесс замены, например, отфильтровать циклические ссылки, то можно использовать второй аргумент JSON.stringify.

Если мы передадим ему массив свойств, будут закодированы только эти свойства.

Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants']) );

// {"title":"Conference","participants":[{},{}]}

Здесь мы, наверно, слишком строги. Список свойств применяется ко всей структуре объекта. Так что свойство внутри participants объекты пустые, потому что name нет в списке. Включим в список все свойства, кроме room.occupiedBy, из-за которого появляется цикличная ссылка:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );

/\*

{

"title":"Conference",

"participants":[{"name":"John"},{"name":"Alice"}],

"place":{"number":23}

}

\*/

Теперь все, кроме occupiedBy, сериализовано. Но список свойств довольно длинный. Но в качестве replacer мы можем использовать функцию, а не массив. Функция будет вызываться для каждой пары (key, value) и должна возвращать «заменённое» значение, которое будет использоваться вместо исходного. Или undefined, чтобы пропустить значение. В нашем случае мы можем вернуть value «как есть» для всего, кроме occupiedBy. Чтобы игнорировать occupiedBy, код ниже возвращает undefined:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, function replacer(key, value) {

alert(`${key}: ${value}`);

return (key == 'occupiedBy') ? undefined : value;

}));

/\* пары ключ:значение, которые приходят в replacer:

: [object Object]

title: Conference

participants: [object Object],[object Object]

0: [object Object]

name: John

1: [object Object]

name: Alice

place: [object Object]

number: 23

\*/

Обратите внимание, что функция replacer получает каждую пару ключ/значение, включая вложенные объекты и элементы массива. И она применяется рекурсивно. Значение this внутри replacer – это объект, который содержит текущее свойство.

Первый вызов – особенный. Ему передаётся специальный «объект-обёртка»: {"": meetup}. Т.е. первая (key, value) пара имеет пустой ключ, а значением является целевой объект в общем. Вот почему первая строка из примера выше будет ":[object Object]". Идея состоит в том, чтобы дать как можно больше возможностей replacer – у него есть возможность проанализировать и заменить/пропустить даже весь объект целиком, если это необходимо.

**[Форматирование: spacer](https://learn.javascript.ru/json" \l "formatirovanie-spacer)**

Третий аргумент в JSON.stringify(value, replacer, spaces) – это количество пробелов, используемых для удобного форматирования. Ранее все JSON-форматированные объекты не имели отступов и лишних пробелов. Это нормально, если мы хотим отправить объект по сети. Аргумент spacer используется исключительно для вывода в удобочитаемом виде.

Ниже spacer = 2 указывает JavaScript отображать вложенные объекты в несколько строк с отступом в 2 пробела внутри объекта:

let user = {

name: "John",

age: 25,

roles: {

isAdmin: false,

isEditor: true

}

};

alert(JSON.stringify(user, null, 2));

/\* отступ в 2 пробела:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

/\* для JSON.stringify(user, null, 4) результат содержал больше отступов:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

Параметр spaces применяется для логирования и красивого вывода.

**[Пользовательский «toJSON»](https://learn.javascript.ru/json" \l "polzovatelskiy-tojson)**

Как и toString для преобразования строк, объект может предоставлять метод toJSON для преобразования в JSON. JSON.stringify автоматически вызывает его, если он есть. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

date: new Date(Date.UTC(2017, 0, 1)),

room

};

alert( JSON.stringify(meetup) );

/\*

{

"title":"Conference",

"date":"2017-01-01T00:00:00.000Z", // (1)

"room": {"number":23} // (2)

}

\*/

Как видим, date (1) стал строкой. Это потому, что все объекты имеют встроенный метод toJSON, который возвращает такую строку. Теперь давайте добавим собственную реализацию метода toJSON в наш объект room (2):

let room = {

number: 23,

toJSON() {

return this.number;

}

};

let meetup = {

title: "Conference",

room

};

alert( JSON.stringify(room) ); // 23

alert( JSON.stringify(meetup) );

/\*

{

"title":"Conference",

"room": 23

}

\*/

Как видите, toJSON используется как при прямом вызове JSON.stringify(room), так когда room вложен в другой сериализуемый объект.

**[JSON.parse](https://learn.javascript.ru/json" \l "json-parse)**

Чтобы декодировать JSON-строку, нам нужен другой метод с именем [JSON.parse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse).

Синтаксис:

let value = JSON.parse(str, [reviver]);

**str**

JSON для преобразования в объект.

**reviver**

Необязательная функция, которая будет вызываться для каждой пары (ключ, значение) и может преобразовывать значение.

Например:

// строковый массив

let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

Или для вложенных объектов:

let user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1

JSON может быть настолько сложным, насколько это необходимо, объекты и массивы могут включать другие объекты и массивы. Но они должны быть в том же JSON-формате.

Вот типичные ошибки в написанном от руки JSON (иногда приходится писать его для отладки):

let json = `{

name: "John", // Ошибка: имя свойства без кавычек

"surname": 'Smith', // Ошибка: одинарные кавычки в значении (должны быть двойными)

'isAdmin': false // Ошибка: одинарные кавычки в ключе (должны быть двойными)

"birthday": new Date(2000, 2, 3), // Ошибка: не допускается конструктор "new", только значения.

"friends": [0,1,2,3] // Здесь все в порядке

}`;

Кроме того, JSON не поддерживает комментарии. Добавление комментария в JSON делает его недействительным. Существует ещё один формат [JSON5](http://json5.org/), который поддерживает ключи без кавычек, комментарии и т.д. Но это самостоятельная библиотека, а не спецификация языка.

Обычный JSON настолько строг не потому, что его разработчики ленивы, а потому, что позволяет легко, надёжно и очень быстро реализовывать алгоритм кодирования и чтения.

**[Использование reviver](https://learn.javascript.ru/json" \l "ispolzovanie-reviver)**

Представьте себе, мы получили объект meetup с сервера в виде строчки данных.

Вот такой:

// title: (meetup title), date: (meetup date)

let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

А теперь нам нужно десериализовать его, т.е. снова превратить в объект JavaScript.

Давайте сделаем это, вызвав JSON.parse:

let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!

Ошибка: значением meetup.date является строка, а не Date объект. JSON.parse не знает, что он должен преобразовать эту строку в Date. Давайте передадим JSON.parse функцию восстановления вторым аргументом, которая возвращает все значения «как есть», но date станет Date:

let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( meetup.date.getDate() ); // 30 - теперь работает!

Это работает и для вложенных объектов:

let schedule = `{

"meetups": [

{"title":"Conference","date":"2017-11-30T12:00:00.000Z"},

{"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}

]

}`;

schedule = JSON.parse(schedule, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( schedule.meetups[1].date.getDate() ); // 18 - отлично!

1. **Intl: интернационализация в JavaScript**

Из-за многообразия языков, стран и их особенностей возникают некоторые проблемы работы со строками, датами, числами в JavaScript. В частности:

**Строки**

При сравнении сравниваются коды символов, а это неправильно, к примеру, в русском языке оказывается, что "ё" > "я" и "а" > "Я", хотя всем известно, что я – последняя буква алфавита и это она должна быть больше любой другой.

**Даты**

В разных странах принята разная запись дат. Где-то пишут 31.12.2014 (Россия), а где-то 12/31/2014 (США), где-то иначе.

**Числа**

В одних странах выводятся цифрами, в других – иероглифами, длинные числа разделяются где-то пробелом, где-то запятой.

Все современные браузеры, кроме IE10- (но есть библиотеки и для него) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), предназначенный решить эти проблемы навсегда.

**[Основные объекты](https://learn.javascript.ru/intl" \l "osnovnye-obekty)**

**Intl.Collator**

Умеет правильно сравнивать и сортировать строки.

**Intl.DateTimeFormat**

Умеет форматировать дату и время в соответствии с нужным языком.

**Intl.NumberFormat**

Умеет форматировать числа в соответствии с нужным языком.

**[Локаль](https://learn.javascript.ru/intl" \l "lokal)**

Локаль – первый и самый важный аргумент всех методов, связанных с интернационализацией. Локаль описывается строкой из трёх компонентов, которые разделяются дефисом:

1. Код языка.
2. Код способа записи.
3. Код страны.

На практике не всегда указаны три, обычно меньше:

1. ru – русский язык, без уточнений.
2. en-GB – английский язык, используемый в Англии (GB).
3. en-US – английский язык, используемый в США (US).
4. zh-Hans-CN – китайский язык (zh), записываемый упрощённой иероглифической письменностью (Hans), используемый в Китае.

Также через суффикс -u-\* можно указать расширения локалей, например "th-TH-u-nu-thai" – тайский язык (th), используемый в Тайланде (TH), с записью чисел тайскими буквами (๐, ๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙) .

Стандарт, который описывает локали – [RFC 5464](http://tools.ietf.org/html/rfc5646), языки описаны в [IANA language registry](http://www.iana.org/assignments/language-subtag-registry/language-subtag-registry). Все методы принимают локаль в виде строки или массива, содержащего несколько локалей в порядке предпочтения. Если локаль не указана или undefined – берётся локаль по умолчанию, установленная в окружении (браузере).

**[Подбор локали localeMatcher](https://learn.javascript.ru/intl" \l "podbor-lokali-localematcher)**

localeMatcher – вспомогательная настройка, которую тоже можно везде указать, она определяет способ подбора локали, если желаемая недоступна.

У него два значения:

* "lookup" – означает простейший порядок поиска путём обрезания суффикса, например zh-Hans-CN → zh-Hans → zh → локаль по умолчанию.
* "best fit" – использует встроенные алгоритмы и предпочтения браузера (или другого окружения) для выбора подходящей локали.

***По умолчанию стоит "best fit".***

Если локалей несколько, например ["zh-Hans-CN", "ru-RU"] то localeMatcher пытается подобрать наиболее подходящую локаль для первой из списка (китайская), если не получается – переходит ко второй (русской) и так далее. Если ни одной не нашёл, например на компьютере не совсем поддерживается ни китайский ни русский, то используется локаль по умолчанию.

Как правило, "best fit" является здесь наилучшим выбором.

**[Строки, Intl.Collator](https://learn.javascript.ru/intl" \l "intl-collator)**

Синтаксис:

// создание

var collator = new Intl.Collator([locales, [options]])

Параметры:

**locales**

Локаль, одна или массив в порядке предпочтения.

**options**

Объект с дополнительными настройками:

* localeMatcher – алгоритм выбора подходящей локали.
* usage – цель сравнения: сортировка "sort" или поиск "search", по умолчанию "sort".
* sensitivity – чувствительность: какие различия в символах учитывать, а какие – нет, варианты:
* base – учитывать только разные символы, без диакритических знаков и регистра, например: а ≠ б, е = ё, а = А.
* accent – учитывать символы и диакритические знаки, например: а ≠ б, е ≠ ё, а = А.
* case – учитывать символы и регистр, например: а ≠ б, е = ё, а ≠ А.
* variant – учитывать всё: символ, диакритические знаки, регистр, например: а ≠ б, е ≠ ё, а ≠ А, используется по умолчанию.
* ignorePunctuation – игнорировать знаки пунктуации: true/false, по умолчанию false.
* numeric – использовать ли численное сравнение: true/false, если true, то будет 12 > 2, иначе 12 < 2.
* caseFirst – в сортировке должны идти первыми прописные или строчные буквы, варианты: "upper" (прописные), lower (строчные) или false (стандартное для локали, также является значением по умолчанию). Не поддерживается IE11-.

В подавляющем большинстве случаев подходят стандартные параметры, то есть options указывать не нужно.

Использование:

var result = collator.compare(str1, str2);

Результат compare имеет значение 1 (больше), 0 (равно) или -1 (меньше). Например:

var collator = new Intl.Collator();

alert( "ёжик" > "яблоко" ); // true (ёжик больше, что неверно)

alert( collator.compare("ёжик", "яблоко") ); // -1 (ёжик меньше, верно)

Выше были использованы полностью стандартные настройки. Они различают регистр символа, но это различие можно убрать, если настроить чувствительность sensitivity:

var collator = new Intl.Collator();

alert( collator.compare("ЁжиК", "ёжик") ); // 1, разные

var collator = new Intl.Collator(undefined, {

sensitivity: "accent"

});

alert( collator.compare("ЁжиК", "ёжик") ); // 0, одинаковые

**[Даты, Intl.DateTimeFormat](https://learn.javascript.ru/intl" \l "intl-datetimeformat)**

Синтаксис:

// создание

var formatter = new Intl.DateTimeFormat([locales, [options]])

Первый аргумент – такой же, как и в Collator, а в объекте options мы можем определить, какие именно части даты показывать (часы, месяц, год…) и в каком формате.

Полный список свойств options:

| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| --- | --- | --- | --- |
| localeMatcher | Алгоритм подбора локали | lookup,best fit | best fit |
| formatMatcher | Алгоритм подбора формата | basic, best fit | best fit |
| hour12 | Включать ли время в 12-часовом формате | true -- 12-часовой формат, false -- 24-часовой |  |
| timeZone | Временная зона | Временная зона, например Europe/Moscow | UTC |
| weekday | День недели | narrow, short, long |  |
| era | Эра | narrow, short, long |  |
| year | Год | 2-digit, numeric | undefined или numeric |
| month | Месяц | 2-digit, numeric, narrow, short, long | undefined или numeric |
| day | День | 2-digit, numeric | undefined или numeric |
| hour | Час | 2-digit, numeric |  |
| minute | Минуты | 2-digit, numeric |  |
| second | Секунды | 2-digit, numeric |  |
| timeZoneName | Название таймзоны (нет в IE11) | short, long |  |

Все локали обязаны поддерживать следующие наборы настроек:

* weekday, year, month, day, hour, minute, second
* weekday, year, month, day
* year, month, day
* year, month
* month, day
* hour, minute, second

Если указанный формат не поддерживается, то настройка formatMatcher задаёт алгоритм подбора наиболее близкого формата: basic – по [стандартным правилам](http://www.ecma-international.org/ecma-402/1.0/#BasicFormatMatcher) и best fit – по умолчанию, на усмотрение окружения (браузера).

Использование:

var dateString = formatter.format(date);

Например:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru");

alert( formatter.format(date) ); // 31.12.2014

var formatter = new Intl.DateTimeFormat("en-US");

alert( formatter.format(date) ); // 12/31/2014

Длинная дата, с настройками:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {

weekday: "long",

year: "numeric",

month: "long",

day: "numeric"

});

alert( formatter.format(date) ); // среда, 31 декабря 2014 г.

Только время:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {

hour: "numeric",

minute: "numeric",

second: "numeric"

});

alert( formatter.format(date) ); // 12:30:00

**[Числа: Intl.NumberFormat](https://learn.javascript.ru/intl" \l "chisla-intl-numberformat)**

Форматтер Intl.NumberFormat умеет красиво форматировать не только числа, но и валюту, а также проценты.

Синтаксис:

var formatter = new Intl.NumberFormat([locales[, options]]);

formatter.format(number); // форматирование

Параметры, как и раньше – локаль и опции.

Список опций:

|  |  |  |  |
| --- | --- | --- | --- |
| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| localeMatcher | Алгоритм подбора локали | lookup, best fit | best fit |
| style | Стиль форматирования | decimal, percent, currency | decimal |
| currency | Алфавитный код валюты | См. [Список кодов валюты](http://www.currency-iso.org/en/home/tables/table-a1.html), например USD |  |
| currencyDisplay | Показывать валюту в виде кода, локализованного символа или локализованного названия | code, symbol, name | symbol |
| useGrouping | Разделять ли цифры на группы | true, false | true |
| minimumIntegerDigits | Минимальное количество цифр целой части | от `1` до `21` | 21 |
| minimumFractionDigits | Минимальное количество десятичных цифр | от 0 до 20 | для чисел и процентов 0, для валюты зависит от кода. |
| maximumFractionDigits | Максимальное количество десятичных цифр | от minimumFractionDigits до 20. | для чисел max(minimumFractionDigits, 3), для процентов 0, для валюты зависит от кода. |
| minimumSignificantDigits | Минимальное количество значимых цифр | от 1 до 21 | 1 |
| maximumSignificantDigits | Максимальное количество значимых цифр | от minimumSignificantDigitsдо 21 | minimumSignificantDigits |

Пример без опций:

var formatter = new Intl.NumberFormat("ru");

alert( formatter.format(1234567890.123) ); // 1 234 567 890,123

С ограничением значимых цифр (важны только первые 3):

var formatter = new Intl.NumberFormat("ru", {

maximumSignificantDigits: 3

});

alert( formatter.format(1234567890.123) ); // 1 230 000 000

С опциями для валюты:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP"

});

alert( formatter.format(1234.5) ); // 1 234,5 £

С двумя цифрами после запятой:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP",

minimumFractionDigits: 2

});

alert( formatter.format(1234.5) ); // 1 234,50 £

**[Методы в Date, String, Number](https://learn.javascript.ru/intl" \l "metody-v-date-string-number)**

Методы форматирования также поддерживаются в обычных строках, датах, числах:

**String.prototype.localeCompare(that [, locales [, options]])**

Сравнивает строку с другой, с учётом локали, например:

var str = "ёжик";

alert( str.localeCompare("яблоко", "ru") ); // -1

**Date.prototype.toLocaleString([locales [, options]])**

Форматирует дату в соответствии с локалью, например:

var date = new Date(2014, 11, 31, 12, 00);

alert( date.toLocaleString("ru", { year: 'numeric', month: 'long' }) ); // Декабрь 2014

**Date.prototype.toLocaleDateString([locales [, options]])**

То же, что и выше, но опции по умолчанию включают в себя год, месяц, день

**Date.prototype.toLocaleTimeString([locales [, options]])**

То же, что и выше, но опции по умолчанию включают в себя часы, минуты, секунды

**Number.prototype.toLocaleString([locales [, options]])**

Форматирует число, используя опции Intl.NumberFormat.

Все эти методы при запуске создают соответствующий объект Intl.\* и передают ему опции, можно рассматривать их как укороченные варианты вызова.

[Старые IE](https://learn.javascript.ru/intl" \l "starye-ie): в IE10- рекомендуется использовать полифил, например библиотеку <https://github.com/andyearnshaw/Intl.js>.