Практическая работа №10 «JavaScript. Объекты. Методы объекта. Конструктор. Символы. Преобразование объектов в примитивы. Строки»

Объекты

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.

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

- Ключи свойств должны быть строками или символами (обычно строками).
- Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

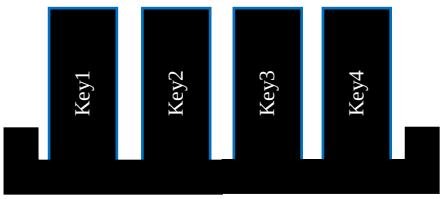
- Запись через точку: obj.property.
- Квадратные скобки obj["property"]. Квадратные скобки позволяют взять ключ из переменной, например, obj[varWithKey].

Дополнительные операторы:

- Удаление свойства: delete obj.prop.
- Проверка существования свойства: "key" in obj.
- Перебор свойств объекта: цикл for (let key in obj).

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
```

Можно добавить свойство с логическим значением:

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

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:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
};
```

Последнее свойство объекта может заканчиваться запятой. 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. 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"];
```

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.

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

```

Вычисляемые свойства

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства. Пример:

```
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 введёт слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

Мы можем использовать и более сложные выражения в квадратных скобках:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};

```

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

В большинстве случаев, когда имена свойств известны и просты, используется запись через точку. Если же нам нужно что-то более сложное, то мы используем квадратные скобки.

Свойство из переменной

В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем. Например:

```
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. Вместо name:name мы можем написать просто name:

```
function makeUser(name, age) {
  return {
    name, // то же самое, что и name: name
    age   // то же самое, что и age: age
    // ...
  };
}

```

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. 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)

```

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

```

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

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

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, 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
}
```

В JavaScript есть много других типов объектов:

- Array для хранения упорядоченных коллекций данных,
- Date для хранения информации о дате и времени,
- Error для хранения информации об ошибке.
- … и так далее.

Копирование объектов и ссылки

Одно из фундаментальных отличий объектов от примитивов заключается в том, что объекты хранятся и копируются «по ссылке», тогда как примитивные значения: строки, числа, логические значения и т.д. – всегда копируются «как целое значение». Это легко понять, если мы немного заглянем под капот того, что происходит, когда мы копируем значение. Здесь мы помещаем копию message во phrase:

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

В результате мы имеем две независимые переменные, каждая из которых хранит строку "Привет!". Объекты ведут себя иначе.

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

Объект хранится где-то в памяти, в то время как переменная имеет лишь «ссылку» на него. Когда мы выполняем действия с объектом, к примеру, берём свойство user.name, движок JavaScript просматривает то, что находится по этому адресу, и выполняет операцию с самим объектом.

При копировании переменной объекта копируется ссылка, но сам объект не дублируется.

Сравнение по ссылке

Два объекта равны только в том случае, если это один и тот же объект. Например, здесь a и b ссылаются на один и тот же объект, поэтому они равны:

```
let a = {};
let b = a; // копирование по ссылке

alert( a == b ); // true, обе переменные ссылаются на один и тот же объект
alert( a === b ); // true
```

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Однако, такие сравнения требуются очень редко и обычно они появляются в результате ошибок программиста.

Клонирование и объединение, Object.assign

В случае необходимости дублировать объект, создать независимую копию, клон, нужно создать новый объект и воспроизвести структуру существующего, перебрав его свойства и скопировав их на примитивном уровне. Например так:

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

let clone = {}; // новый пустой объект

// давайте скопируем все свойства user в него
for (let key in user) {
  clone[key] = user[key];
}

// теперь clone это полностью независимый объект с тем же содержимым
clone.name = "Pete"; // изменим в нём данные

alert( user.name ); // все ещё John в первоначальном объекте
```

Также мы можем использовать для этого метод Object.assign. Синтаксис:

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

- Первый аргумент dest — целевой объект.
- Остальные аргументы src1, ..., srcN (может быть столько, сколько необходимо) являются исходными объектами
- Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект dest. Другими словами, свойства всех аргументов, начиная со второго, копируются в первый объект.
- Возвращает объект dest.

Например, мы можем использовать его для объединения нескольких объектов в один:

```
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);

// теперь user = { name: "John", canView: true, canEdit: true }
```

Мы также можем использовать Object.assign для замены цикла for..in для простого клонирования:

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

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

```

Он копирует все свойства user в пустой объект и возвращает его. Также существуют и другие методы клонирования объекта. Например, с использованием оператора расширения clone = {...user}.

Вложенное клонирование

До сих пор мы предполагали, что все свойства user примитивные. Но свойства могут быть и ссылками на другие объекты. Например, есть объект:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182

```

Теперь недостаточно просто скопировать clone.sizes = user.sizes, потому что user.sizes – это объект, он будет скопирован по ссылке. Таким образом, clone и user будут иметь общий объект sizes:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

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

alert( user.sizes === clone.sizes ); // true, тот же объект

// user и clone обладают общим свойством sizes
user.sizes.width++;       // изменяем свойства в первом объекте
alert(clone.sizes.width); // 51, видим результат в другом

```

Чтобы исправить это, мы должны использовать цикл клонирования, который проверяет каждое значение user[key] и, если это объект, тогда также копирует его структуру. Это называется «глубоким клонированием». Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед заново, возьмите готовую реализацию, например _.cloneDeep(obj) из библиотеки JavaScript lodash.

Сборка мусора

Управление памятью в JavaScript выполняется автоматически и незаметно. Основной алгоритм сборки мусора называется «алгоритм пометок» (от англ. «mark-and-sweep»). Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

1. Сборщик мусора «помечает» (запоминает) все корневые объекты.
2. Затем он идёт по ним и «помечает» все ссылки из них.
3. Затем он идёт по отмеченным объектам и отмечает их ссылки. Все посещённые объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
4. …И так далее, пока не будут посещены все достижимые (из корней) ссылки.

Иными словами, сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её. Объекты сохраняются в памяти, пока они достижимы. Если на объект есть ссылка – вовсе не факт, что он является достижимым (из корня): набор взаимосвязанных объектов может стать недоступен в целом, как мы видели в примере выше.

Движки JavaScript применяют множество оптимизаций, чтобы она работала быстрее и не задерживала выполнение кода. Вот некоторые из оптимизаций:

- Сборка по поколениям (Generational collection) – объекты делятся на два набора: «новые» и «старые». В типичном коде многие объекты имеют короткую жизнь: они появляются, выполняют свою работу и быстро умирают, так что имеет смысл отслеживать новые объекты и, если это так, быстро очищать от них память. Те, которые выживают достаточно долго, становятся «старыми» и проверяются реже.
- Инкрементальная сборка (Incremental collection) – если объектов много, и мы пытаемся обойти и пометить весь набор объектов сразу, это может занять некоторое время и привести к видимым задержкам в выполнения скрипта. Так что движок делит всё множество объектов на части, и далее очищает их одну за другой. Получается несколько небольших сборок мусора вместо одной всеобщей. Это требует дополнительного учёта для отслеживания изменений между частями, но зато получается много крошечных задержек вместо одной большой.
- Сборка в свободное время (Idle-time collection) – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора. Современные движки реализуют разные продвинутые алгоритмы сборки мусора.

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 функциями в свойствах.

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

Значение this – это объект «перед точкой», который используется для вызова метода. Например:

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

  sayHi() {
    // "this" - это "текущий объект".
    alert(this.name);
  }

};

user.sayHi(); // John
```

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

«this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Его можно использовать в любой функции, даже если это не метод объекта. Значение this вычисляется во время выполнения кода, в зависимости от контекста. Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» в вызовах:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// используем одну и ту же функцию в двух объектах
user.f = sayHi;
admin.f = sayHi;

// эти вызовы имеют  разное значение this
// "this" внутри функции - это объект "перед точкой"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (нет разницы между использованием точки или квадратных скобок для доступа к объекту)
```

Правило простое: если вызывается obj.f(), то во время вызова f, this – это obj. Так что, в приведённом выше примере это либо user, либо admin.

У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы ссылаемся на this внутри такой функции, то оно берётся из внешней «нормальной» функции. Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Ilya

```

Это особенность стрелочных функций. Она полезна, когда мы на самом деле не хотим иметь отдельное this, а скорее хотим взять его из внешнего контекста.

- Функции, которые находятся в свойствах объекта, 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.
- Функция может быть скопирована между объектами (из одного объекта в другой).
- Когда функция вызывается синтаксисом «метода» – object.method(), значением this во время вызова является object.

Конструктор, оператор "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество похожих, однотипных объектов, таких как пользователи, элементы меню и так далее. Это можно сделать при помощи функции-конструктора и оператора "new".

Функция-конструктор

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

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна выполняться только с помощью оператора "new".

Например:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

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

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

Другими словами, new User(...) делает что-то вроде:

```
function User(name) {
  // this = {};  (неявно)

  // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;

  // return this;  (неявно)
}
```

Таким образом, let user = new User("Jack") возвращает тот же результат, что и:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Теперь, если нам будет необходимо создать других пользователей, мы можем просто вызвать new User("Ann"), new User("Alice") и так далее. Данная конструкция гораздо удобнее и читабельнее, чем многократное создание литерала объекта. Это и является основной целью конструкторов – реализовать код для многократного создания однотипных объектов.

Возврат значения из конструктора, return

Обычно конструкторы не имеют оператора return. Их задача – записать все необходимое в this, и это автоматически становится результатом.

Но если return всё же есть, то применяется простое правило:

- При вызове return с объектом, вместо this вернётся объект.
- При вызове return с примитивным значением, оно проигнорируется.

Другими словами, return с объектом возвращает этот объект, во всех остальных случаях возвращается this. К примеру, здесь return замещает this, возвращая объект:

```
function BigUser() {

  this.name = "John";

  return { name: "Godzilla" };  // <-- возвращает этот объект
}

alert( new BigUser().name );  // Godzilla, получили этот объект
```

А вот пример с пустым return (или мы могли бы поставить примитив после return, неважно):

```
function SmallUser() {

  this.name = "John";

  return; // <-- возвращает this
}

alert( new SmallUser().name );  // John

```

Обычно у конструкторов отсутствует return.

Создание методов в конструкторе

Использование конструкторов для создания объектов даёт большую гибкость. Функции-конструкторы могут иметь параметры, определяющие, как создавать объект и что в него записывать. Мы можем добавить к this не только свойства, но и методы. Например, new User(name) ниже создаёт объект с заданным name и методом sayHi:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "Меня зовут: " + this.name );
  };
}

let john = new User("John");

john.sayHi(); // Меня зовут: John

/*
john = {
   name: "John",
   sayHi: function() { ... }
}
*/

```

Для создания сложных объектов есть и более продвинутый синтаксис – классы.

Опциональная цепочка '?.'

Опциональная цепочка ?. — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует, или проблема «несуществующего свойства».

В качестве примера предположим, что у нас есть объекты user, которые содержат информацию о наших пользователях. У большинства наших пользователей есть адреса в свойстве user.address с улицей user.address.street, но некоторые из них их не указали. В таком случае, когда мы попытаемся получить user.address.street, а пользователь окажется без адреса, мы получим ошибку:

```
let user = {}; // пользователь без свойства "address"

alert(user.address.street); // Ошибка!

```

Это ожидаемый результат. JavaScript работает следующим образом. Поскольку user.address имеет значение undefined, попытка получить user.address.street завершается ошибкой. Во многих практических случаях мы бы предпочли получить здесь undefined вместо ошибки (что означало бы «улицы нет»). В веб-разработке мы можем получить объект, соответствующий элементу веб-страницы, с помощью специального вызова метода, такого как document.querySelector('.elem'), и он возвращает null, когда такого элемента нет.

```
// document.querySelector('.elem') равен null, если элемента нет
let html = document.querySelector('.elem').innerHTML; // ошибка, если он равен null

```

Если нам необходимо избежать ошибки и просто принять html = null в качестве результата, очевидным решением было бы проверить значение с помощью if или условного оператора ?, прежде чем обращаться к его свойству, вот так:

```
let user = {};

alert(user.address ? user.address.street : undefined);
```

Это работает, тут нет ошибки. Однако поиск элемента user.address здесь вызывается дважды, что не очень хорошо. Для более глубоко вложенных свойств это ещё менее красиво, поскольку потребуется больше повторений. Работая в команде, у кого-то могут даже возникнуть проблемы с пониманием такого кода.

Есть немного лучший способ написать это, используя оператор &&:

```
let user = {}; // пользователь без адреса

alert( user.address && user.address.street && user.address.street.name ); // undefined (без ошибки)
```

Проход при помощи логического оператора И && через весь путь к свойству гарантирует, что все компоненты существуют (если нет, вычисление прекращается), но также не является идеальным. Как вы можете видеть, имена свойств по-прежнему дублируются в коде. Именно для этого нужна опциональная цепочка ?.

Опциональная цепочка

Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если значение перед ?. равно undefined или null.

Другими словами, value?.prop:

- работает как value.prop, если значение value существует,
- в противном случае (когда value равно undefined/null) он возвращает undefined.

Вот безопасный способ получить доступ к user.address.street, используя ?.:

```
let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)
```

Код лаконичный и понятный, в нем вообще нет дублирования.

Не злоупотребляйте опциональной цепочкой!

К примеру, если, в соответствии с логикой нашего кода, объект user должен существовать, но address является необязательным, то нам следует писать user.address?.street, но не user?.address?.street. В этом случае, если вдруг user окажется undefined, мы увидим программную ошибку по этому поводу и исправим её. В противном случае, если слишком часто использовать ?., ошибки могут замалчиваться там, где это неуместно, и их будет сложнее отлаживать.

Сокращённое вычисление

Как было сказано ранее, ?. немедленно останавливает вычисление, если левая часть не существует. Так что если после ?. есть какие-то вызовы функций или операции, то они не произойдут. Например:

```
let user = null;
let x = 0;

user?.sayHi(x++); // нет "user", поэтому выполнение не достигает вызова sayHi и x++

alert(x); // 0, значение не увеличилось
```

Другие варианты применения: ?.(), ?.[]

Опциональная цепочка ?. — это не оператор, а специальная синтаксическая конструкция, которая также работает с функциями и квадратными скобками. Например, ?.() используется для вызова функции, которая может не существовать. В приведённом ниже коде у некоторых наших пользователей есть метод admin, а у некоторых его нет:

```
let userAdmin = {
  admin() {
    alert("Я админ");
  }
};

let userGuest = {};

userAdmin.admin?.(); // Я админ

userGuest.admin?.(); // ничего не произойдет (такого метода нет)
```

Здесь в обеих строках мы сначала используем точку (userAdmin.admin), чтобы получить свойство admin, потому что мы предполагаем, что объект user существует, так что читать из него безопасно. Затем ?.() проверяет левую часть: если функция admin существует, то она запускается (это так для userAdmin). В противном случае (для userGuest) вычисление остановится без ошибок.

Мы можем использовать ?. для безопасного чтения и удаления, но не для записи

Опциональная цепочка ?. не имеет смысла в левой части присваивания. Например:

```
let user = null;

user?.name = "John"; // Ошибка, не работает
// то же самое что написать undefined = "John"
```

Синтаксис опциональной цепочки ?. имеет три формы:

- obj?.prop – возвращает obj.prop если obj существует, в противном случае undefined.
- obj?.[prop] – возвращает obj[prop] если obj существует, в противном случае undefined.
- obj.method?.() – вызывает obj.method(), если obj.method существует, в противном случае возвращает undefined.

Как мы видим, все они просты и понятны в использовании. ?. проверяет левую часть на null/undefined и позволяет продолжить вычисление, если это не так.

Тип данных Symbol

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки или символы. Ни числа, ни логические значения не подходят, разрешены только эти два типа данных.

Символы

«Символ» представляет собой уникальный идентификатор. Создаются новые символы с помощью функции Symbol():

```
// Создаём новый символ - id
let id = Symbol();
```

При создании, символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

```
// Создаём символ id с описанием (именем) "id"
let id = Symbol("id");
```

Символы гарантированно уникальны. Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет. Например, вот два символа с одинаковым описанием – но они не равны:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false

```

Символы в JavaScript имеют свои особенности, и не стоит думать о них, как о символах в Ruby или в других языках программирования.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов. Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в for..in, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования. Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.
2. Существует множество системных символов, используемых внутри JavaScript, доступных как Symbol.*. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать Symbol.iterator для итераторов, Symbol.toPrimitive для настройки преобразования объектов в примитивы и так далее.

Символы не преобразуются автоматически в строки

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция alert принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически.

Это – языковая «защита» от путаницы, ведь строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга. Если же мы действительно хотим вывести символ с помощью alert, то необходимо явно преобразовать его с помощью метода .toString(), вот так:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), теперь работает
```

«Скрытые» свойства

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы. Например, мы работаем с объектами user, которые принадлежат стороннему коду. Мы хотим добавить к ним идентификаторы. Используем для этого символьный ключ:

```
let user = {
  name: "Вася"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // мы можем получить доступ к данным по ключу-символу
```

Так как объект user принадлежит стороннему коду, и этот код также работает с ним, то нам не следует добавлять к нему какие-либо поля. Это небезопасно.

Символы в литеральном объекте

Если мы хотим использовать символ при литеральном объявлении объекта {...}, его необходимо заключить в квадратные скобки.

```
let id = Symbol("id");

let user = {
  name: "Вася",
  [id]: 123 // просто "id: 123" не сработает
};

```

Это вызвано тем, что нам нужно использовать значение переменной id в качестве ключа, а не строку «id».

Символы игнорируются циклом for…in

Свойства, чьи ключи – символы, не перебираются циклом for..in. Например:

```
let id = Symbol("id");
let user = {
  name: "Вася",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (свойства с ключом-символом нет среди перечисленных)

// хотя прямой доступ по символу работает
alert( "Напрямую: " + user[id] );

```

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше символьное свойство. Object.keys(user) также игнорирует символы. А вот Object.assign, в отличие от цикла for..in, копирует и строковые, и символьные свойства:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

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

alert( clone[id] ); // 123

```

Здесь нет никакого парадокса или противоречия. Так и задумано. Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать все свойства (включая такие свойства с ключами-символами, как, например, id в примере выше).

Глобальные символы

Итак, как мы видели, обычно все символы уникальны, даже если их имена совпадают. Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство. Для этого существует глобальный реестр символов. Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам гарантированно будет возвращаться один и тот же символ. Для чтения (или, при отсутствии, создания) символа из реестра используется вызов Symbol.for(key). Он проверяет глобальный реестр и, при наличии в нём символа с именем key, возвращает его, иначе же создаётся новый символ Symbol(key) и записывается в реестр под ключом key. Например:

```
// читаем символ из глобального реестра и записываем его в переменную
let id = Symbol.for("id"); // если символа не существует, он будет создан

// читаем его снова и записываем в другую переменную (возможно, из другого места кода)
let idAgain = Symbol.for("id");

// проверяем -- это один и тот же символ
alert( id === idAgain ); // true

```

Символы, содержащиеся в реестре, называются глобальными символами. Если вам нужен символ, доступный везде в коде – используйте глобальные символы.

Symbol.keyFor

Для глобальных символов, кроме Symbol.for(key), который ищет символ по имени, существует обратный метод: Symbol.keyFor(sym), который, наоборот, принимает глобальный символ и возвращает его имя. К примеру:

```
// получаем символ по имени
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// получаем имя по символу
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id

```

Внутри метода Symbol.keyFor используется глобальный реестр символов для нахождения имени символа. Так что этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт undefined.

Системные символы

Существует множество «системных» символов, использующихся внутри самого JavaScript, и мы можем использовать их, чтобы настраивать различные аспекты поведения объектов. Эти символы перечислены в спецификации в таблице Well-known symbols:

- Symbol.hasInstance
- Symbol.isConcatSpreadable
- Symbol.iterator
- Symbol.toPrimitive
- …и так далее.

В частности, Symbol.toPrimitive позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву.

Преобразование объектов в примитивы

JavaScript не позволяет настраивать, как операторы работают с объектами. В отличие от некоторых других языков программирования, таких как Ruby или C++, мы не можем реализовать специальный объектный метод для обработки сложения (или других операторов). В случае таких операций, объекты автоматически преобразуются в примитивы, затем выполняется сама операция над этими примитивами, и на выходе мы получим примитивное значение.

Это важное ограничение: результатом obj1 + obj2 (или другой математической операции) не может быть другой объект!

В реальных проектах нет математики с объектами. Если она всё же происходит, то за редким исключением, это из-за ошибок в коде. Однако данную тему того, как объект преобразуется в примитив и как это можно настроить стоит рассмотреть:

1. Это позволит нам понять, что происходит в случае ошибок в коде, когда такая операция произошла случайно.
2. Есть исключения, когда такие операции возможны и вполне уместны. Например, вычитание или сравнение дат (Date объекты). Мы встретимся с ними позже.

Преобразование объекта в примитив вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве значения.

Существует всего 3 типа для этого:

- "string" (для alert и других операций, которым нужна строка)
- "number" (для математических операций)
- "default" (для некоторых других операторов, обычно объекты реализуют его как "number")

Алгоритм преобразования таков:

1. Сначала вызывается метод obj[Symbol.toPrimitive](hint), если он существует,
2. В случае, если хинт равен "string"
3. происходит попытка вызвать obj.toString() и obj.valueOf(), смотря что есть.
4. В случае, если хинт равен "number" или "default"
5. происходит попытка вызвать obj.valueOf() и obj.toString(), смотря что есть.
6. Все эти методы должны возвращать примитив (если определены).

На практике часто бывает достаточно реализовать только obj.toString() в качестве универсального метода для преобразований к строке, который должен возвращать удобочитаемое представление объекта для целей логирования или отладки.

Методы примитивов

JavaScript позволяет нам работать с примитивными типами данных – строками, числами и т.д., как будто они являются объектами. У них есть и методы.

Ключевые различия между примитивами и объектами:

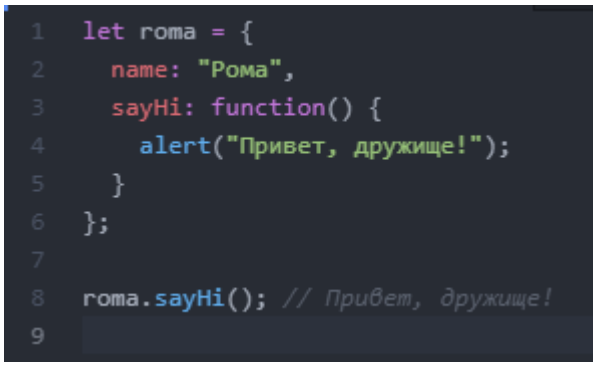
Примитив

- Это – значение «примитивного» типа.
- Есть 7 примитивных типов: string, number, boolean, symbol, null, undefined и bigint.

Объект

- Может хранить множество значений как свойства.
- Объявляется при помощи фигурных скобок {}, например: {name: "Рома", age: 30}. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что мы можем хранить функцию как одно из свойств объекта.



Здесь мы создали объект roma с методом sayHi. Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы. Однако у этих возможностей есть обратная сторона! Объекты «тяжелее» примитивов. Они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Примитив как объект

Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean, Symbol и BigInt. Таким образом, они имеют разный набор методов. К примеру, существует метод str.toUpperCase(), который возвращает строку в верхнем регистре.

```
let str = "Привет";

alert( str.toUpperCase() ); // ПРИВЕТ

```

Вот, что на самом деле происходит в str.toUpperCase():

1. Строка str – примитив. В момент обращения к его свойству, создаётся специальный объект, который знает значение строки и имеет такие полезные методы, как toUpperCase().
2. Этот метод запускается и возвращает новую строку (показывается в alert).
3. Специальный объект удаляется, оставляя только примитив str.

Получается, что примитивы могут предоставлять методы, и в то же время оставаться «лёгкими».

Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта. Однако, он всё же должен придерживаться спецификаций и работать так, как будто он его создаёт. Число имеет собственный набор методов. Например, toFixed(n) округляет число до n знаков после запятой.

```
let num = 1.23456;

alert( num.toFixed(2) ); // 1.23

```

Строки

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков. Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

Кавычки

В JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

```
let single = 'single-quoted';
let double = "double-quoted";
let backticks = `backticks`;

```

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в ${…}:

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.

```

Ещё одно преимущество обратных кавычек — они могут занимать более одной строки:

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList); // список гостей, состоящий из нескольких строк

```

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в документации. Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

Спецсимволы

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

```
// перевод строки добавлен с помощью символа перевода строки
let str1 = "Hello\nWorld";

// многострочная строка, созданная с использованием обратных кавычек
let str2 = `Hello
World`;

alert(str1 == str2); // true

```

Есть и другие, реже используемые спецсимволы:

\n	Перевод строки

\r	В текстовых файлах Windows для перевода строки используется комбинация символов \r\n, а на других ОС это просто \n. Это так по историческим причинам, ПО под Windows обычно понимает и просто \n.
\', \"	Кавычки
\\	Обратный слеш
\t	Знак табуляции
\b, \f, \v	Backspace, Form Feed и Vertical Tab — оставлены для обратной совместимости, сейчас не используются.

Как вы можете видеть, все спецсимволы начинаются с обратного слеша, \ — так называемого «символа экранирования». Он также используется, если необходимо вставить в строку кавычку. К примеру:

```
1  alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш — \' — иначе она бы обозначала окончание строки. Разумеется, требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Так что мы можем применить и более элегантное решение, использовав для этой строки двойные или обратные кавычки:

```
1  alert( `I'm the Walrus!` ); // I'm the Walrus!
```

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Вы можете явно видеть это в выводах alert в примерах выше. Если нам надо добавить в строку собственно сам обратный слеш \ необходимо добавить перед ним ещё один обратный слеш.

```
1  alert( `The backslash: \\` ); // The backslash: \
```

Длина строки

Свойство length содержит длину строки:

```
1  alert( `My\n`.length ); // 3
```

Обратите внимание, \n — это один спецсимвол, поэтому тут всё правильно: длина строки 3.

length — это свойство

Бывает так, что люди с практикой в других языках случайно пытаются вызвать его, добавляя круглые скобки: они пишут str.length() вместо str.length. Это не работает. Так как str.length — это числовое свойство, а не функция, добавлять скобки не нужно.

Доступ к символам

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: str.charAt(pos). Первый символ занимает нулевую позицию:

```
let str = `Hello`;

// получаем первый символ
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// получаем последний символ
alert( str[str.length - 1] ); // o
```

Квадратные скобки — современный способ получить символ, в то время как charAt существует в основном по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt — пустую строку:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (пустая строка)
```

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана — она такая навсегда. Можно создать новую строку и записать её в ту же самую переменную вместо старой. Например:

```
let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi
```

Изменение регистра

Методы toLowerCase() и toUpperCase() меняют регистр символов:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Если мы захотим перевести в нижний регистр какой-то конкретный символ:

```
1   alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Поиск подстроки

Существует несколько способов поиска подстроки.

str.indexOf

Первый метод — str.indexOf(substr, pos). Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений. Например:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, потому что подстрока 'Widget' найдена в начале
alert( str.indexOf('widget') ); // -1, совпадений нет, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1, подстрока "id" найдена на позиции 1 (..idget with id)

```

Необязательный второй аргумент позволяет начать поиск с определённой позиции. Например, первое вхождение "id" — на позиции 1. Для того, чтобы найти следующее, начнём поиск с позиции 2:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12

```

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

```
let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Найдено тут: ${foundPos}` );
  pos = foundPos + 1; // продолжаем со следующей позиции
}

```

includes, startsWith, endsWith

Более современный метод str.includes(substr, pos) возвращает true, если в строке str есть подстрока substr, либо false, если нет. Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:

```
alert( "Midget".includes("id") ); // true
alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3
```

Методы str.startsWith и str.endsWith проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

```
alert( "Widget".startsWith("Wid") ); // true, "Wid" – начало "Widget"
alert( "Widget".endsWith("get") ); // true, "get" – окончание "Widget"
```

Получение подстроки

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

метод	выбирает…	отрицательные значения
slice(start, end)	от start до end (не включая end)	можно передавать отрицательные значения
substring(start, end)	между start и end	отрицательные значения равнозначны 0
substr(start, length)	length символов, начиная от start	значение start может быть отрицательным

Все эти методы эффективно выполняют задачу. Формально у метода substr есть небольшой недостаток: он описан не в собственно спецификации JavaScript, а в приложении к ней — Annex B. Это приложение описывает возможности языка для использования в браузерах, существующие в основном по историческим причинам. Таким образом, в другом окружении, отличном от браузера, он может не поддерживаться. Однако на практике он работает везде. Из двух других вариантов, slice более гибок, он поддерживает отрицательные аргументы, и его короче писать. Так что, в принципе, можно запомнить только его.

Сравнение строк

Строки сравниваются посимвольно в алфавитном порядке. Тем не менее, есть некоторые нюансы.

1. Строчные буквы больше заглавных:
   ```
   alert( 'a' > 'Z' ); // true
   ```
2. Буквы, имеющие диакритические знаки, идут «не по порядку»:

```
alert( 'Österreich' > 'Zealand' ); // true
```

Это может привести к своеобразным результатам при сортировке названий стран: нормально было бы ожидать, что Zealand будет после Österreich в списке. Чтобы разобраться, что происходит, давайте ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в UTF-16. Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

Правильное сравнение

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты. Поэтому браузеру нужно знать, какой язык использовать для сравнения. К счастью, все современные браузеры поддерживают стандарт ECMA 402, обеспечивающий правильное сравнение строк на разных языках с учётом их правил. Для этого есть соответствующий метод.

Вызов str.localeCompare(str2) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

- Отрицательное число, если str меньше str2.
- Положительное число, если str больше str2.
- 0, если строки равны.

Например:

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

У этого метода есть два дополнительных аргумента, которые указаны в документации. Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

Практическое задание

1. Создайте простую проверку (капчу) для проверки пользователя. Код должен включать в себя генерацию букв разного регистра. Если пользователь вводит правильное значение кнопка отправки формы работает. Если же пользователь ошибается ему предлагается другая капча с генерацией чисел. Пользователю показывается пример с двумя случайными числами N + M и ему необходимо ввести сумму двух чисел. Если true, кнопка отправки формы становится активной, в противном случае выводится сообщение об ошибке. Также, с помощью функции isEmpty(obj), которая возвращает true, если у объекта нет свойств, иначе false, введите проверку на пустой ввод.

2. Создайте «корзину» или элемент с похожим функционалом для вашего сайта. Создайте 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() должен использовать prompt для считывания нового числа и прибавления его к value.

Другими словами, свойство value представляет собой сумму всех введённых пользователем значений, с учётом начального значения startingValue.

3. Для карточек с текстом необходимо создать функцию truncate(str, maxlength), которая проверяет длину строки str и, если она превосходит maxlength, заменяет конец str на "…", так, чтобы её длина стала равна maxlength. Результатом функции должна быть та же строка, если усечение не требуется, либо, если необходимо, усечённая строка.