

Цели занятия

- подробно разобрать работу с объектами в JavaScript;
- разобрать что такое опциональная цепочка и для чего она используется;
- изучить комментарии и основы JSDoc;

Полезные материалы

- [Советы по стилю кода](#)
- [Подробнее про JSDoc](#)
- [Подробнее про объекты в JavaScript](#)

Объекты

Как мы уже знаем, 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. 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.

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

Литералы и свойства

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

```
let user = {  
  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`.

Для обращения к свойствам используется запись «через точку»:

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

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

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

Для удаления свойства мы можем использовать оператор `delete`:

```
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  
};
```

Но тогда и читать это свойство нужно не через точку, а с помощью квадратных скобок:

```
alert(user.likes birds); // Будет ошибка  
alert(user['likes birds']); // А вот так правильно
```

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.

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

```
alert("blabla" in user); // false, user.blabla не существует
```

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)".

Комментарии

Комментарии необходимы как опытным, так и начинающим разработчикам. Комментировать свой код необходимо по простой причине: ваш код рано или поздно перейдет в другие руки, и позволит другому программисту быстрее разобраться как он устроен. К тому же вы и сами спустя 1-2 недели можете забыть как работает ваш код, но за счет комментариев можно в него быстрее вникнуть и продолжить с ним работать.

Но далеко не все комментарии будут одинаково полезны, например, новички очень часто пишут в комментариях то, что делает их код:

```
// Если a делится на 2 без остатка, то прибавляем единицу к  
переменной s  
  
if (a % 2 === 0) {  
    s = s + 1;  
}
```

И в таком виде комментарий не несёт большой пользы, так как из кода это и так понятно; по сути мы словно перевели код с английского на русский. Но хороший комментарий должен помогать понять **ДЛЯ ЧЕГО** это условие было написано!

```
// Считаем кол-во чётных элементов a  
  
if (a % 2 === 0) {  
    s = s + 1;  
}
```

Но в хорошем коде количество «объясняющих» комментариев должно быть минимальным. Серьёзно, код должен быть таким, чтобы его можно было понять без комментариев.

Про это есть хорошее правило: «Если код настолько запутанный, что требует комментариев, то, может быть, его стоит переделать?»

Например, если оформить пример выше в виде функции, то само название функции подсказывало бы нам что она делает и дополнительный комментарий не понадобился.

Опциональная цепочка ‘?.’

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

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»).

Как мы можем избежать ошибки? 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... Но это довольно неэлегантно. Как вы можете видеть, `"user.address"` появляется в коде дважды. Для более глубоко

вложенных свойств это ещё менее красиво, поскольку потребуется больше повторений.

К примеру, давайте аналогично вычислим `user.address.street.name`.

```
let user = {}; // у пользователя нет адреса

alert(user.address ? (user.address.street ? user.address.street.name : null) : null);
```

Это просто ужасно, у кого-то могут даже возникнуть проблемы с пониманием такого кода.

Именно для решения этой проблемы не так давно в JavaScript добавили функционал, называющийся опциональной цепочкой и который позволяет безопасно получать свойства вложенных объектов. Выглядит это вот так:

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

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

JSDoc

Но помимо объясняющих комментариев для отдельных участков кода есть хорошая практика документировать каждую функцию в проекте, причём для этого даже был создан отдельный формат, который называется [JSDoc](#).

Например:

```
/**
 * Возвращает x, возведённое в n-ную степень.
 *
 * @param {number} x Возводимое в степень число.
 * @param {number} n Степень, должна быть натуральным числом.
 * @return {number} x, возведённое в n-ную степень.
 */
```

```
function pow(x, n) {  
    ...  
}
```

Здесь стоит обратить внимание на структуру комментария и ключевые слова внутри него, например, `@param` позволяет описать отдельный параметр функции, указать его тип в фигурных скобках и добавить описание, а слово `@return` позволяет описать результат, который возвращает данная функция (в данном случае мы сразу видим, что функция должна будет вернуть число, возведенное в n-ую степень)

Это шаблон для документирования функций, который часто используется именно в таком виде, но формат JSDoc позволяет структурировать комментарии не только для функций, подробнее об этом читайте в [этой статье](#).

Поиск внутри строки

Одна из самых частных потребностей при работе со строками это проверить, что в строке есть необходимые нам части или строка начинается/заканчивается определенными символами. Для этого у строк есть методы `includes()`, `startsWith()`, `endsWith()`

```
let message = 'Hello, frontenders!';  
  
// В строке есть указанная подстрока?  
alert( message.includes("frontenders") ); // true  
alert( message.includes("backenders") ); // false  
  
// Строка начинается на "Hello" ?  
alert( message.startsWith("Hello") ); // true  
  
// Строка заканчивается на "!" ?  
alert( message.endsWith("!") ); // true
```

```
// Строка заканчивается на "test" ?  
alert( message.endsWith("test") ); // false
```

Кроме этого существует ещё и метод [indexOf\(\)](#), но используется он реже, поэтому его изучение выходит за рамки занятия.