

JavaScript - объекты

ОСНОВЫ

Объекты, копирование объекта и ссылки, “this”,
опциональная цепочка “?.”, сборка мусора

Объект в JavaScript-е

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).

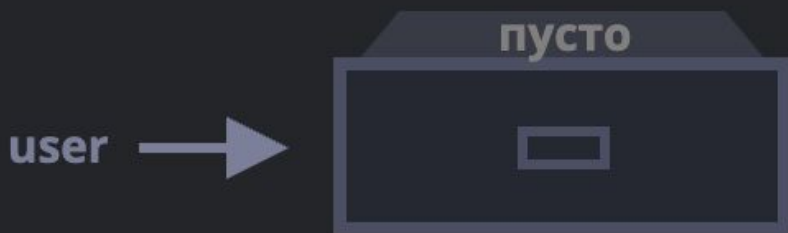
Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

В JavaScript "объект" - это как контейнер, который может содержать разные вещи, называемые свойствами. Давай представим, что это ящик, в который ты можешь положить разные игрушки. Каждая игрушка - это свойство объекта.

Пустой объект

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

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

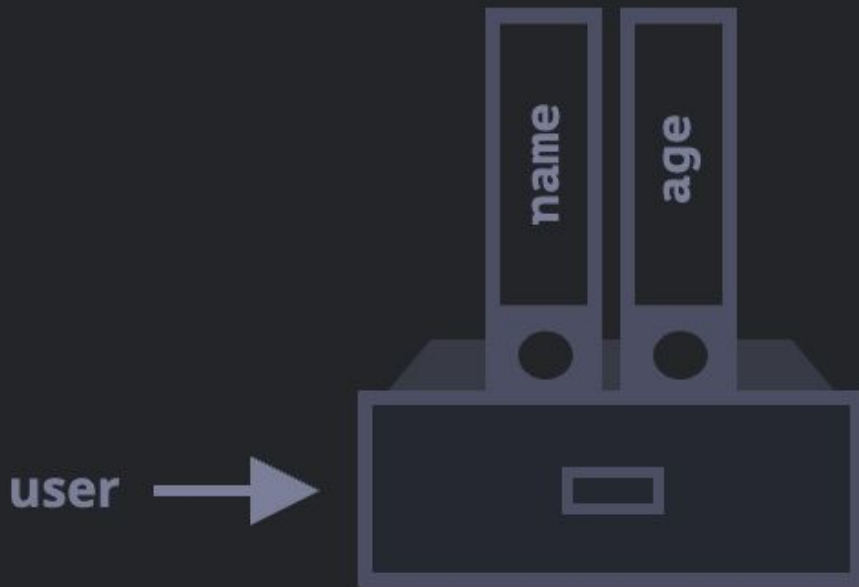


Литералы и свойства

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

В объекте `user` сейчас находятся два свойства:

- Первое свойство с именем `"name"` и значением `"John"`.
- Второе свойство с именем `"age"` и значением `30`.



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

Манипуляции с свойствами объекта

Для обращения к свойствам используется запись «через точку»;

Также можно добавить новое свойство в объект «через точку»;

Для удаления свойства мы можем использовать оператор delete;

Также удалить свойство можно присвоив ему значение undefined.

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

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

```
delete user.age;
```

Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает

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"];
```

```
let key = "likes birds";
```

```
// то же самое, что и user["likes birds"] = true;  
user[key] = true;
```

Свойство из переменной

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

Если название свойств совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств, используем специальные короткие свойства для упрощения этой записи.

```
function makeUser(name, age) {  
  return {  
    name: name,  
    age: age  
    // ...другие свойства  
  };  
}  
  
let user = makeUser("John", 30);  
alert(user.name); // John
```

```
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

Имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д. Но для свойств объекта такого ограничения нет. Нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов. Если использовать число 0 в качестве ключа, то оно превратится в строку "0".

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

```
let obj = {};  
obj.__proto__ = 5; // присвоим число  
alert(obj.__proto__); // [object Object], значение – это объект
```

```
// эти имена свойств допустимы  
let obj = {  
  for: 1,  
  let: 2,  
  return: 3  
};
```

```
let obj = {  
  0: "Тест" // то же самое что и "0": "Тест"  
};
```

```
// обе функции alert выведут одно и то же свойство  
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.

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

```
let user = {};
```

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

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

```
alert( "age" in user ); // true, user.age существует
```

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

```
let obj = {  
  test: undefined  
};
```

```
alert( obj.test ); // выведет undefined, значит свойство не существует?
```

```
alert( "test" in obj ); // true, свойство существует!
```

Цикл "for..in"

Для перебора всех свойств объекта используется цикл for..in.

```
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  
}
```

```
for (key in object) {  
  // тело цикла выполняется для каждого свойства объекта  
}
```

Особенность объекта объявленного как const

Объект, объявленный через const, может быть изменён.

Может показаться, что строка (*) должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление const защищает от изменений только саму переменную user, а не ее содержимое.

Определение const выдаст ошибку только если мы присвоим переменной другое значение: user=....

```
const user = {  
  name: "John"  
};
```

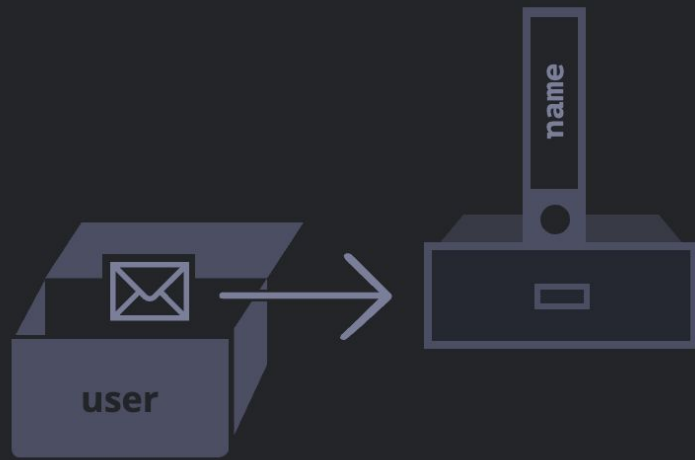
```
user.name = "Pete"; // (*)
```

```
alert(user.name); // Pete
```

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».

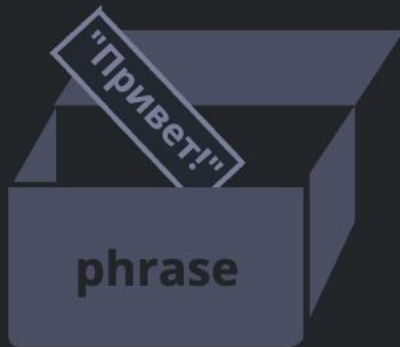
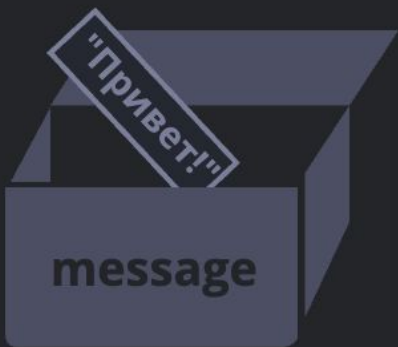
Переменная, которой присвоен объект, хранит не сам объект, а его «адрес в памяти» — другими словами, «ссылку» на него.



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

Пример с копированием примитивов

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



Здесь мы помещаем копию message во phrase

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

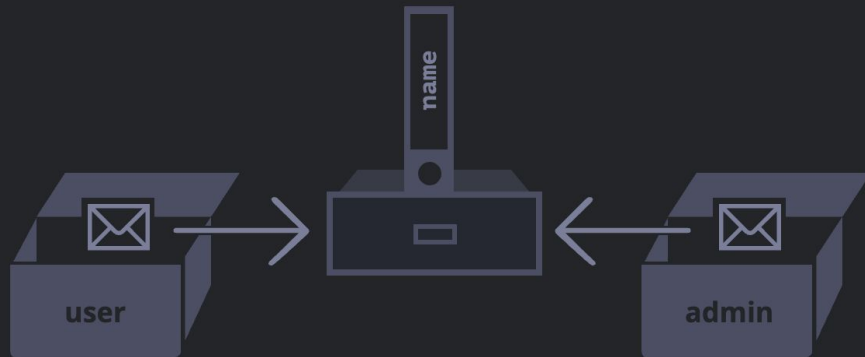
Копирование объекта по ссылке

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

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

Это как если бы у нас был шкафчик с двумя ключами, и мы использовали один из них (admin), чтобы войти в него и внести изменения. А затем, если мы позже используем другой ключ (user), мы все равно открываем тот же шкафчик и можем получить доступ к измененному содержимому.

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



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

```
let admin = user;
```

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

```
alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"
```

Сравнение по ссылке

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

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

```
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
```

Клонирование и объединение

Итак, копирование объектной переменной создаёт ещё одну ссылку на тот же объект.

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

Это тоже выполнимо, но немного сложнее, потому что в JavaScript для этого нет встроенного метода. Но на самом деле в этом редко возникает необходимость, копирования по ссылке в большинстве случаев вполне хватает.

Но если мы действительно этого хотим, то нам нужно создать новый объект и воспроизвести структуру существующего, перебрав его свойства и скопировав их на примитивном уровне.

```
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
```

Вложенное клонирование

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

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`.

Также мы можем использовать глобальный метод `structuredClone()`, который позволяет сделать полную копию объекта. К сожалению он поддерживается только современными браузерами.

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

Методы объекта

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.

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

ООП вскользь (теория и знакомство)

Когда мы пишем наш код, используя объекты для представления сущностей реального мира, — это называется объектно-ориентированным программированием или сокращённо: «ООП».

ООП является большой предметной областью и интересной наукой самой по себе. Как выбрать правильные сущности? Как организовать взаимодействие между ними? Это — создание архитектуры, и на эту тему есть отличные книги, такие как «Приемы объектно-ориентированного проектирования. Паттерны проектирования» авторов Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес или «Объектно-ориентированный анализ и проектирование с примерами приложений» Гради Буча, а также ещё множество других книг.

Методы объекта пример

```
let user = {  
  // ...  
};  
  
// сначала, объявляем  
function sayHi() {  
  alert("Привет!");  
}  
  
// затем добавляем в качестве метода  
user.sayHi = sayHi;  
  
user.sayHi(); // Привет!
```

```
let user = {  
  name: "John",  
  age: 30  
};  
  
user.sayHi = function() {  
  alert("Привет!");  
};  
  
user.sayHi(); // Привет!
```

Сокращённая запись метода

мы можем
пропустить
ключевое слово
"function" и просто
написать sayHi().

```
// эти объекты делают одно и то же
```

```
user = {  
  sayHi: function() {  
    alert("Привет");  
  }  
};
```

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

```
user = {  
  sayHi() { // то же самое, что и "sayHi: function(){...}"  
    alert("Привет");  
  }  
};
```

Ключевое слово «this» в методах

Как правило, методу объекта обычно требуется доступ к информации, хранящейся в объекте, для выполнения своей работы.

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

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

Преимущество this

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

Технически также возможно получить доступ к объекту без ключевого слова `this`, обратившись к нему через внешнюю переменную (в которой хранится ссылка на этот объект).

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

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

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

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

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

  sayHi() {
    alert(user.name); // "user" вместо "this"
  }
};
```

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

  sayHi() {
    alert(user.name); // приведёт к ошибке
  },
};

You, Moments ago • Uncommitted changes
let admin = user;
user = null; // перезапишем переменную для наглядности, теперь она не хранит ссылку на объект.

admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

Опциональная цепочка '?.'

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

Проблема «несуществующего свойства»

Это ожидаемый результат. JavaScript работает следующим образом. Поскольку `user.address` имеет значение `undefined`, попытка получить `user.address.street` завершается ошибкой.

Очевидным решением было бы проверить значение с помощью `if` или условного оператора `?`, прежде чем обращаться к его свойству.

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

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.

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

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

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

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

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

alert( user.address && user.address.street && user.address.street.name );
```

Опциональная цепочка

Опциональная цепочка `?.` останавливает процесс, если значение перед `?.` равно `undefined`, если значение перед `?.` равно `null`.

Далее в этой статье, для краткости, мы будем использовать `value?.prop` вместо `value != null ? value.prop : undefined` «существует», если оно не является `null` или `undefined`.

Другими словами, `value?.prop`:

работает как `value.prop`, если значение `value` существует,

в противном случае (когда `value` равно `undefined/null`) он возвращает `undefined`.

```
let user = {}; // пользователь без адреса
alert( user?.address?.street ); // undefined (без ошибки)
```

```
let user = null;
alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

Не злоупотребляйте опциональной цепочкой

Нам следует использовать `?.` только там, где нормально, что чего-то не существует.

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.

Переменная перед ?. должна быть объявлена

Если переменной `user` вообще нет, то `user?.anything` приведёт к ошибке:

Переменная должна быть объявлена (к примеру, как `let/const/var user` или как параметр функции). Опциональная цепочка работает только с объявленными переменными.

```
// ReferenceError: user is not defined  
user?.address;
```

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"
```

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.

Синтаксис ?.[] также работает, если мы хотим использовать скобки [] для доступа к свойствам вместо точки .. Как и в предыдущих случаях, он позволяет безопасно считывать свойство из объекта, который может не существовать.

Также мы можем использовать ?. с delete.

```
delete user?.name; // удаляет user.name если пользователь существует
```

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

```
let userGuest = {};
```

```
userAdmin.admin?.(); // Я админ
```

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

```
let key = "firstName";
```

```
let user1 = {  
  firstName: "John"  
};
```

```
let user2 = null;
```

```
alert( user1?.[key] ); // John
```

```
alert( user2?.[key] ); // undefined
```

Сборка мусора

Управление памятью в JavaScript выполняется автоматически и незаметно. Мы создаём примитивы, объекты, функции... Всё это занимает память.

Но что происходит, когда что-то больше не нужно? Как движок JavaScript обнаруживает, что пора очищать память?

Основной концепцией управления памятью в JavaScript является принцип достижимости.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

Существует базовое множество достижимых значений, которые не могут быть удалены.

Например:

- Выполняемая в данный момент функция, ее локальные переменные и параметры.
- Другие функции в текущей цепочке вложенных вызовов, их локальные переменные и параметры.
- Глобальные переменные.
- (некоторые другие внутренние значения)

Эти значения мы будем называть корнями.

Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

- Например, если в глобальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Далее вы познакомитесь с подробными примерами на эту тему.

В движке JavaScript есть фоновый процесс, который называется сборщиком мусора. Он отслеживает все объекты и удаляет те, которые стали недоступными.

```
1 // в user находится ссылка на объект
2 let user = {
3   name: "John"
4 };
```

<global>

user
↓

Object
name: "John"

```
1 user = null;
```

<global>
user: null



Object
name: "John"

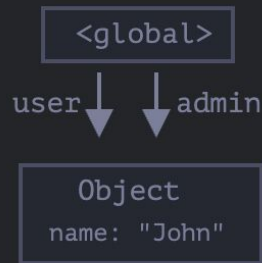
Две ссылки

Представим, что мы скопировали ссылку из `user` в `admin`:

Теперь, если мы сделаем то же самое:

...то объект `John` всё ещё достигим через глобальную переменную `admin`, поэтому он находится в памяти. Если бы мы также перезаписали `admin`, то `John` был бы удален.

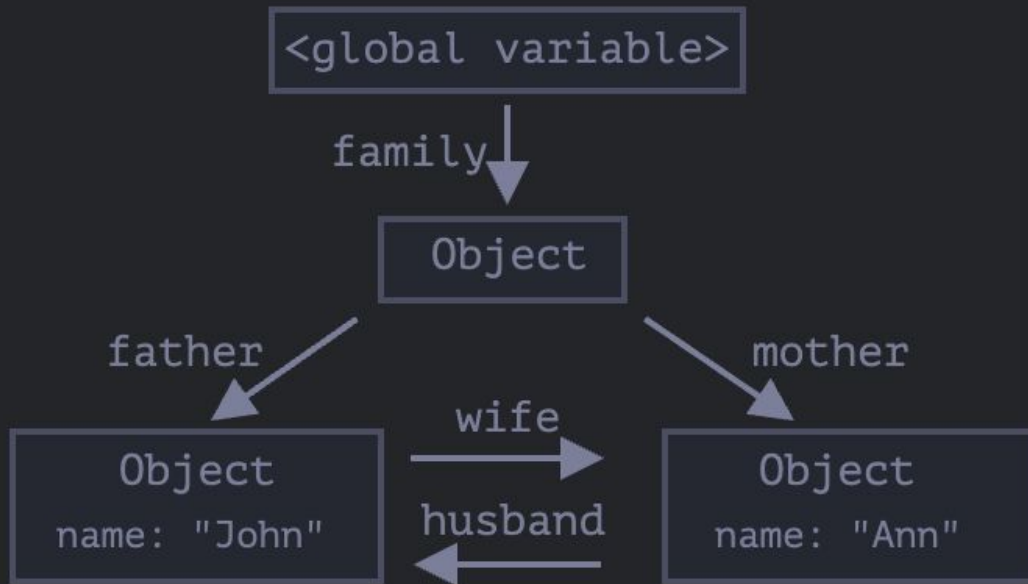
```
1 // в user находится ссылка на объект
2 let user = {
3   name: "John"
4 };
5
6 let admin = user;
```



```
1 user = null;
```

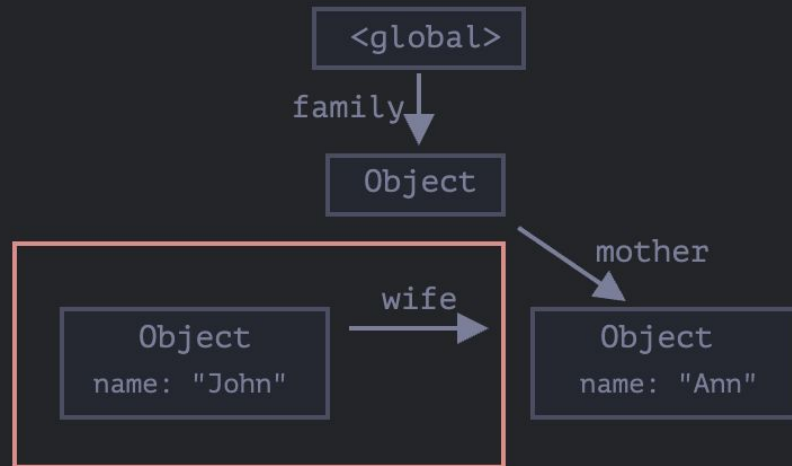
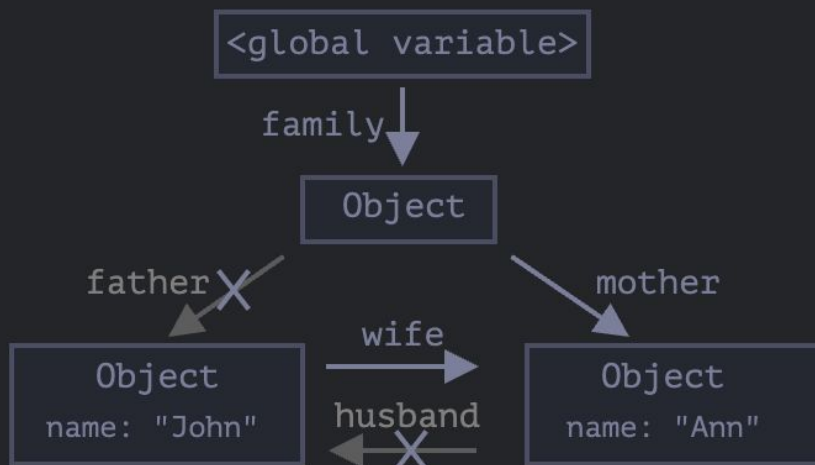
Взаимосвязанные объекты

```
function marry(man, woman) {  
  woman.husband = man;  
  man.wife = woman;  
  
  return {  
    father: man,  
    mother: woman  
  }  
}  
  
let family = marry({  
  name: "John"  
}, {  
  name: "Ann"  
});
```

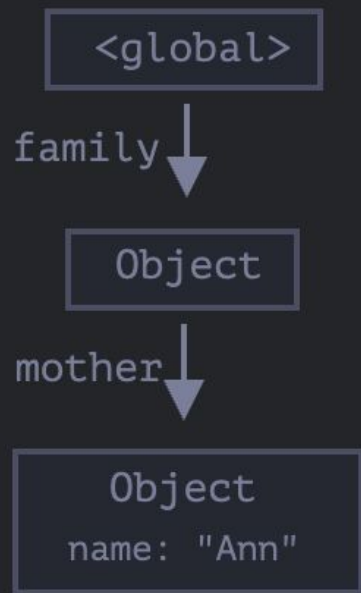


Удаление сложных связей

```
delete family.father;  
delete family.mother.husband;
```

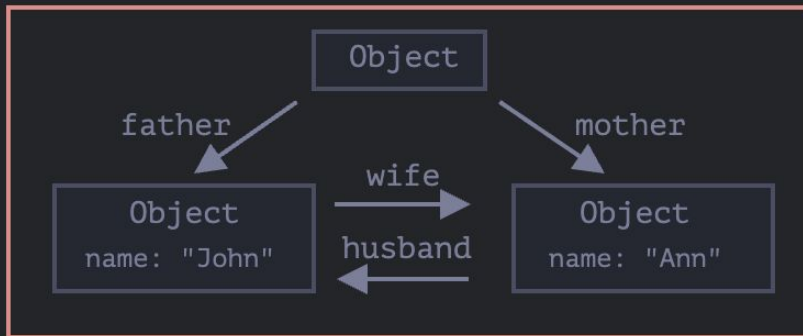
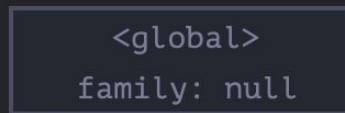


После сборки мусора:



Недостижимый «остров»

Структура в памяти теперь станет такой:



```
family = null;
```

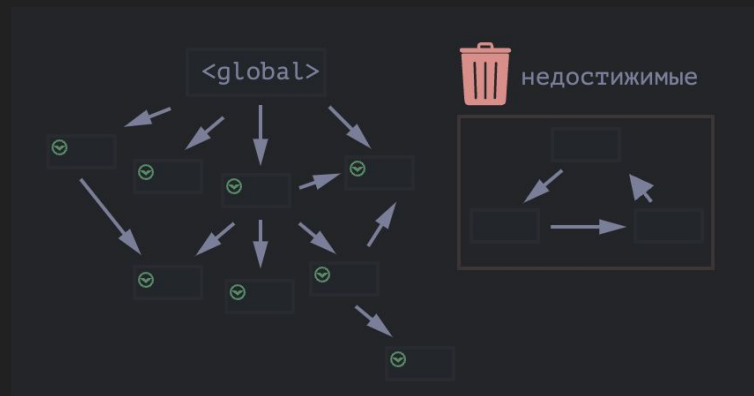
Внутренние алгоритмы

Основной алгоритм сборки мусора называется «алгоритм пометок» (от англ. «mark-and-sweep»).

Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

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

Визуализация алгоритма



Главное, что нужно знать:

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.

НО в целом все происходит “под-капотом” в JavaScript-е и сам сбор мусора не реализуется в рамках написания кода, это нужно просто понимать и знать

Ресурсы

Объекты - [ТЫК](#)

Копирование объектов и ссылка - [ТЫК](#)

Метод объекта “this” - [ТЫК](#)

Опциональная цепочка “?.” - [ТЫК](#)

Lodash документация (на английском) - [ТЫК](#)

Сборка мусора - [ТЫК](#)