Разработка мобильных приложений

Лекция 7

Сборка мусора (garbage collection)

- Одна из форм автоматического управления памятью. Специальный процесс, называемый сборщиком мусора (garbage collector), периодически освобождает память, удаляя объекты, которые уже не будут востребованы
- В JavaScript сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её.

Достижимость

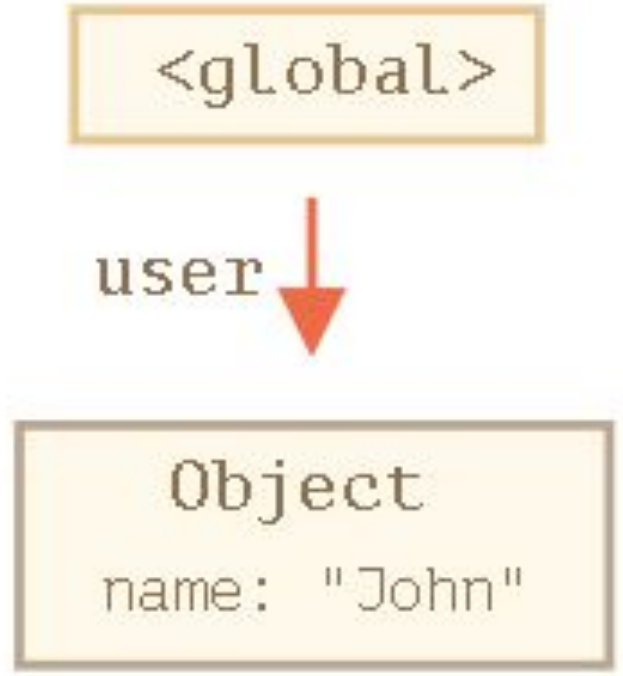
- Основной концепцией управления памятью в JavaScript является принцип достижимости.
- Достижимые значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

- Существует базовое множество достижимых значений, которые не могут быть удалены (далее **Корни**):
  - Локальные переменные и параметры текущей функции.
  - Переменные и параметры других функций в текущей цепочке вложенных вызовов.
  - Глобальные переменные.

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

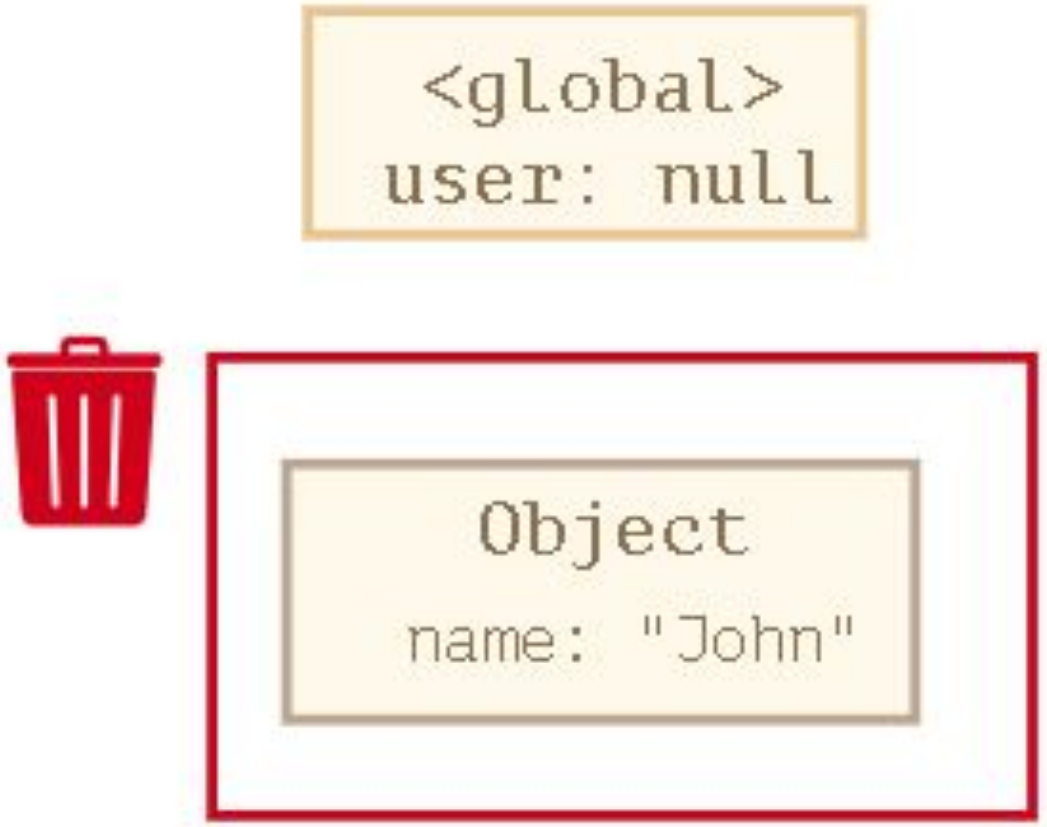
Простой пример

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



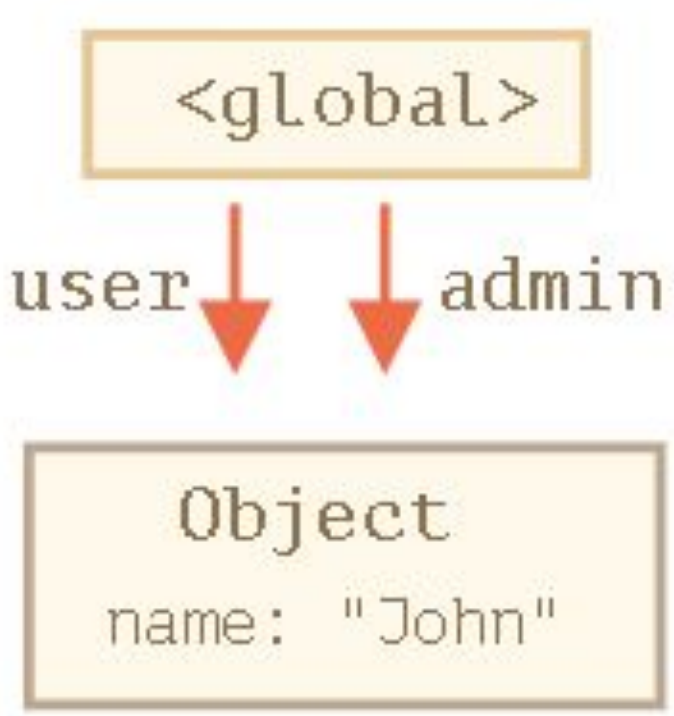
Удаляем ссылку на объект

```
user = null;
```



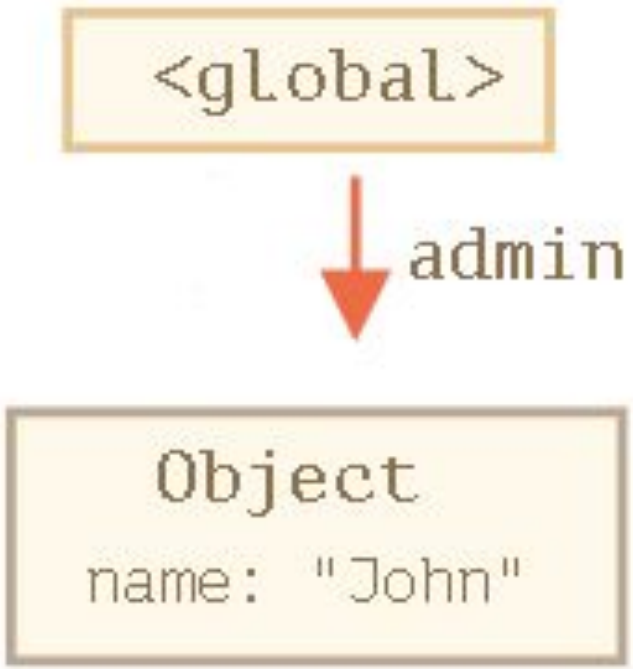
Две ссылки

```
// в user находится ссылка на объект
let user = {
    name: "John"
  };
let admin = user;
```



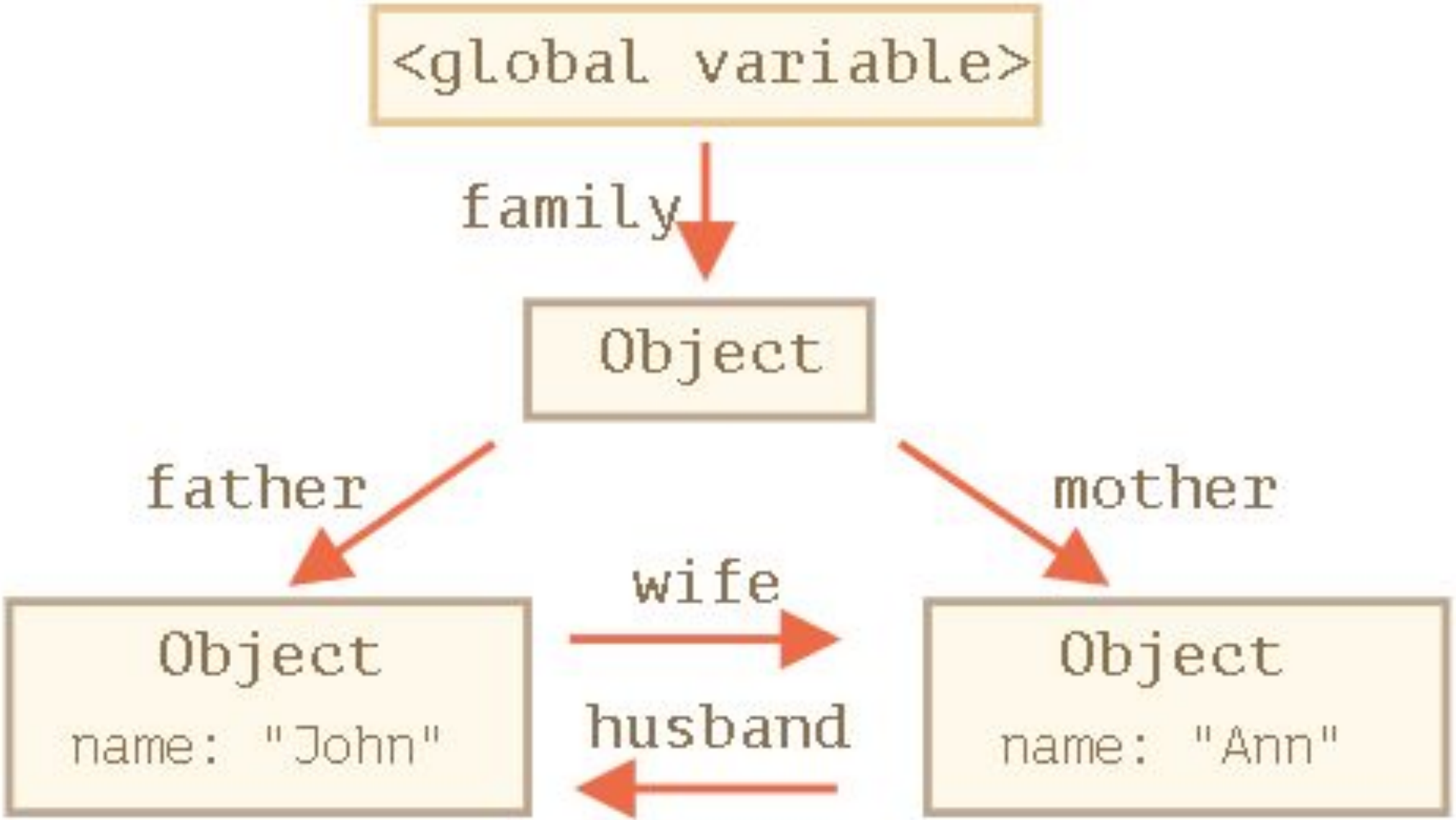
Удаляем ссылку на объект

```
user = null;
```

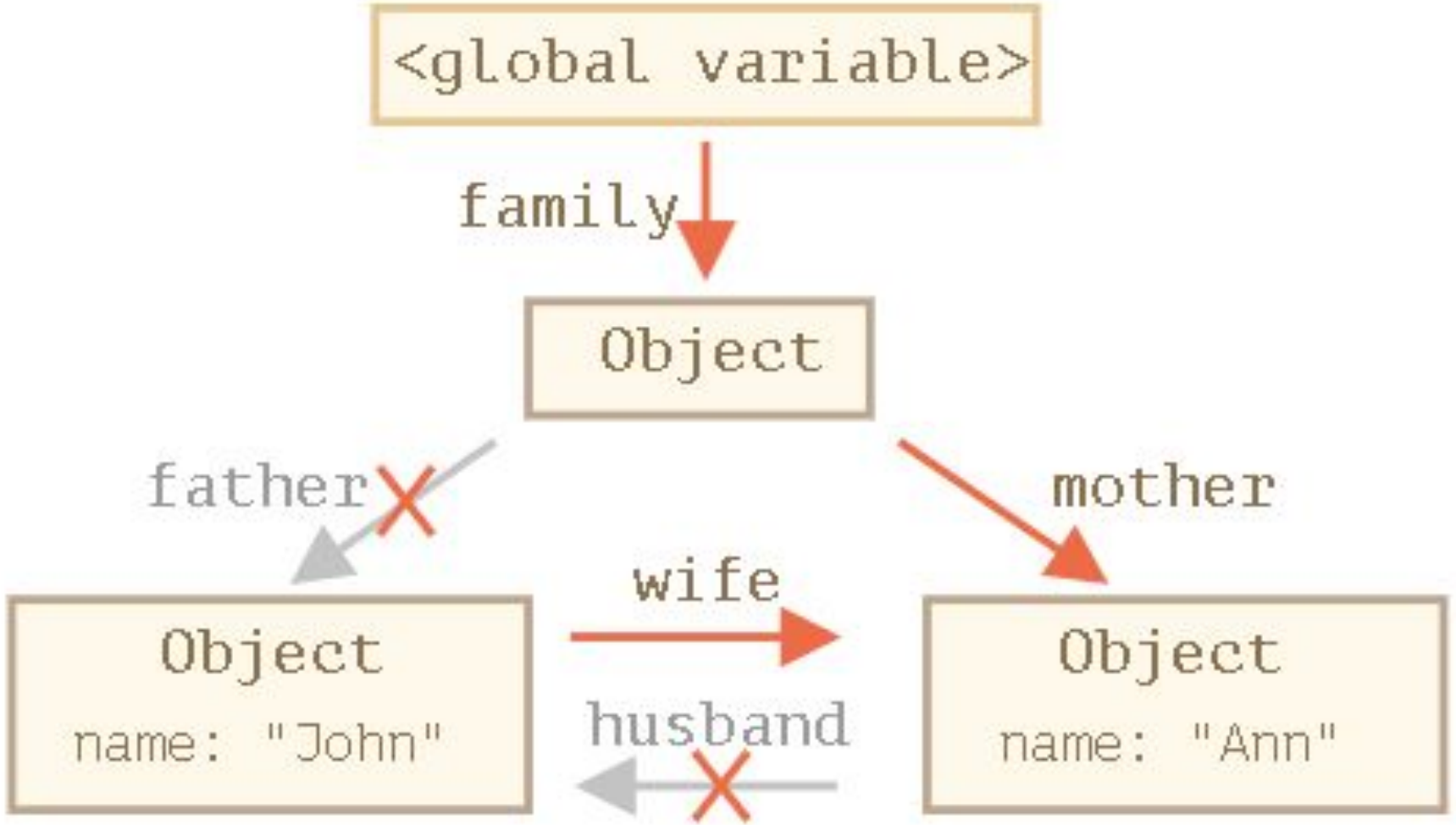


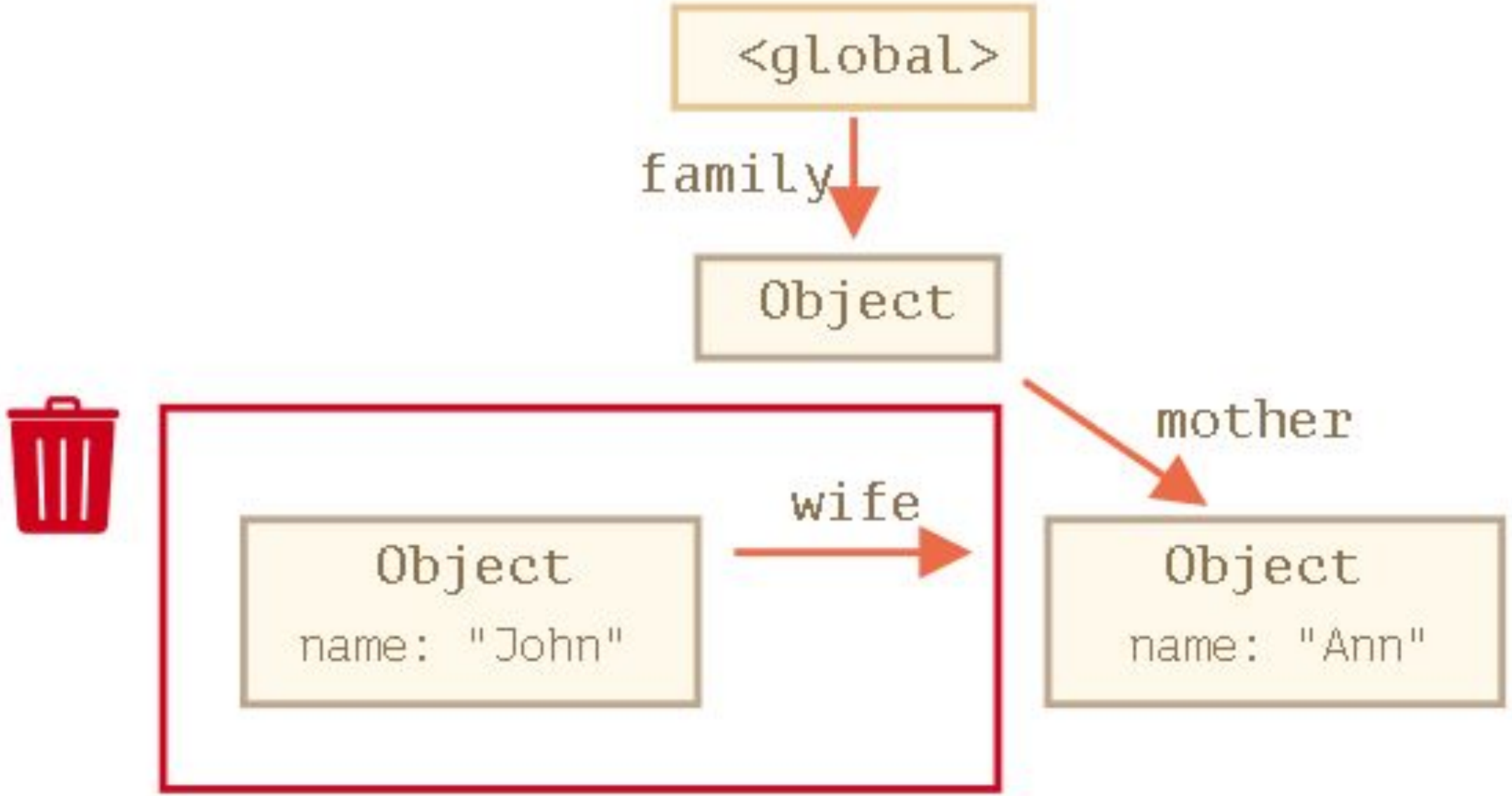
Взаимосвязанные объекты

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

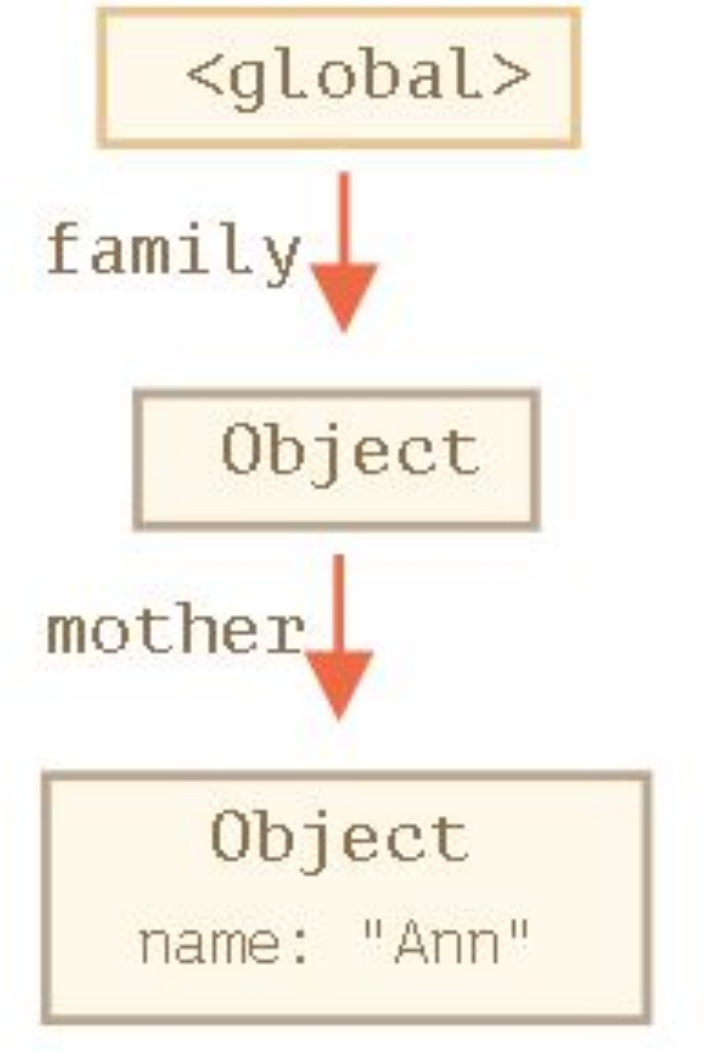

<global variable>
family
Object
father
mother
wife
husband
Object
name: "John"
Object
name: "Ann"

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




<global>
family
Object
mother
wife
Object
name: "John"
Object
name: "Ann"

После сборки мусора

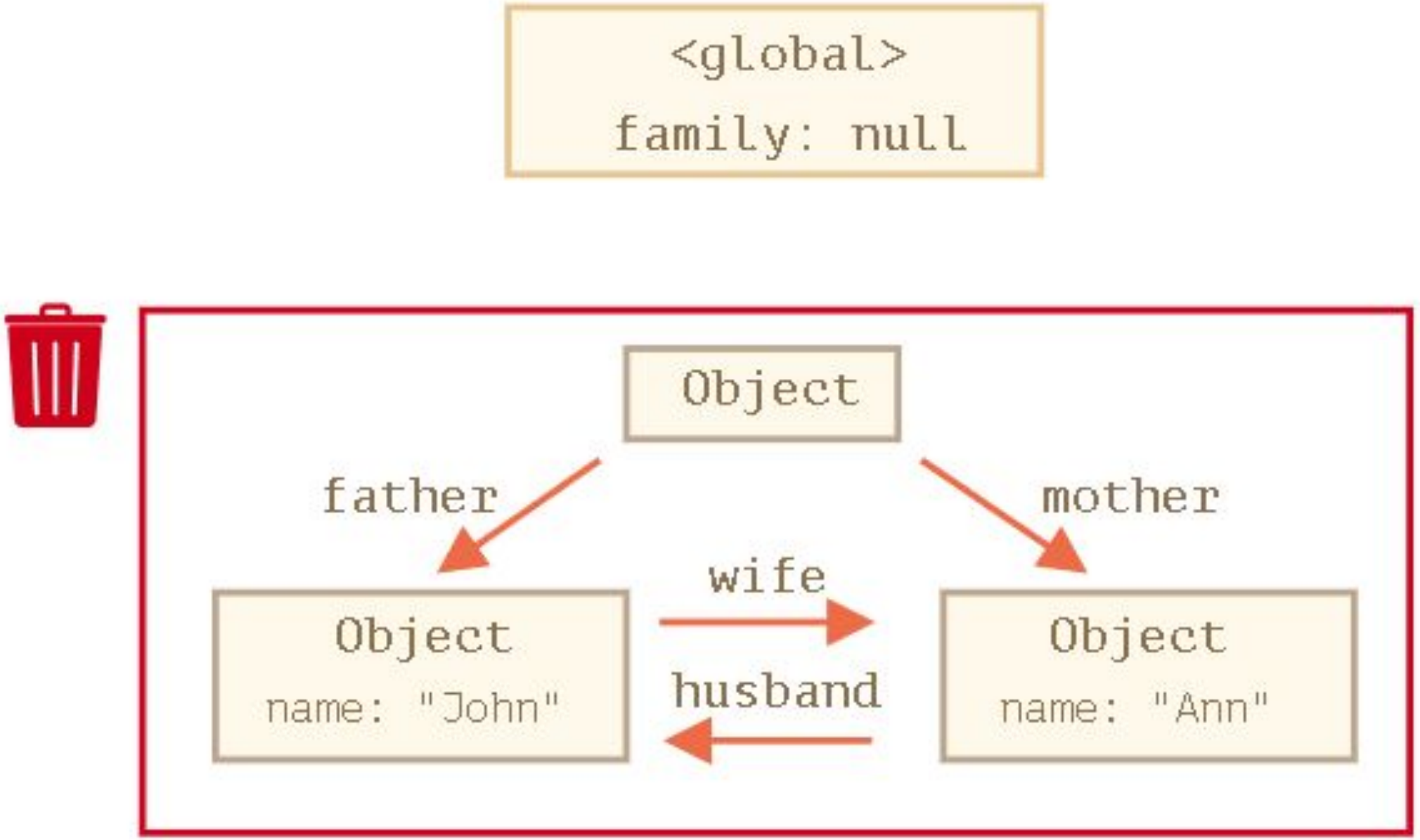


- Недостаточно удалить только одну из этих ссылок, потому что все объекты останутся достижимыми.
- Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым.

Недостижимый «остров»

- Вполне возможна ситуация, при которой целый «остров» связанных объектов может стать недостижимым и удалится из памяти.

```
family = null;
```


<global>
family: null
Object
father
mother
wife
husband
Object
name: "John"
Object
name: "Ann"

Алгоритм сборщика мусора

- Сборщик мусора «помечает» (запоминает) все корневые объекты.
- Затем он идёт по их ссылкам и помечает все найденные объекты.
- Затем он идёт по ссылкам помеченных объектов и помечает объекты, на которые есть ссылка от них. Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
- …И так далее, пока не будут посещены все ссылки (достижимые от корней).
- Все непомеченные объекты удаляются.

Методы объекта

- Функциональное выражение:

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

- Создали объект
- Объявили функцию
- Присвоили функцию объекту
- Можем вызывать функцию

Сокращённая запись метода

```
// эти объекты делают одно и то же (одинаковые методы)
user = {
    sayHi: function() {
      alert("Привет");
    }
};
user = {
    sayHi() { // то же самое, что и "sayHi: function()"
        alert("Привет");
    }
};
```

Ключевое слово «this» в методах

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

- Значение this – это объект «перед точкой», который использовался для вызова метода.

```
let user = {
    name: "Джон",
    age: 30,
    sayHi() {
        // this - это "текущий объект"
        alert(this.name);
    }
};
user.sayHi();
```

Джон

```
let user = {
    name: "Джон",
    age: 30,
    sayHi() {
      alert( user.name );
    }
};
let admin = user;
user = null;
admin.sayHi();
```

Ошибка! user не ссылается на объект!

«this» не является фиксированным

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

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

«this» не является фиксированным

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

```
let user = { name: "Джон" };
let admin = { name: "Админ" };
function sayHi() {
  alert( this.name );
}
user.f = sayHi;
admin.f = sayHi;
user.f(); // Джон  (this == user)
admin.f(); // Админ  (this == admin)
admin['f'](); // Админ
```

Последствия свободного this

- Вы, скорее всего, привыкли к идее "фиксированного this" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод).
- В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Некоторые хитрые способы вызова метода приводят к потере значения this

```
let user = {
    name: "Джон",
    hi() { alert(this.name); },
    bye() { alert("Пока"); }
  };

user.hi();
(user.name == "Джон" ? user.hi : user.bye)();
```

Джон

Ошибка!

```
let user = {
    name: "Джон",
    hi() { alert(this.name); }
}
let hi = user.hi;
hi();
```

Ошибка!

У стрелочных функций нет «this»

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

Илья

Функция-конструктор

- Функции-конструкторы являются обычными функциями. Но есть два соглашения:
  - Имя функции-конструктора должно начинаться с большой буквы.
  - Функция-конструктор должна вызываться при помощи оператора "new".

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

  let user = new User("Вася");
  alert(user.name);
  alert(user.isAdmin);
```

Вася

false

- Создаётся новый пустой объект, и он присваивается this.
- Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
- Возвращается значение this.

```
function User(name) {
    // this = {};  (неявно)
    // добавляет свойства к this
    this.name = name;
    this.isAdmin = false;
    // return this;  (неявно)
  }
```

Возврат значения из конструктора return

- Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в this, который в итоге станет результатом.
- Но если return всё же есть, то применяется простое правило:
- При вызове return с объектом, будет возвращён объект, а не this.
- При вызове return с примитивным значением, примитивное значение будет отброшено.
- Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this.

```
function BigUser() {
    this.name = "Вася";
    return { name: "Godzilla" };
    // возвращает этот объект
}
alert( new BigUser().name );
 // Godzilla, получили этот объект
```

```
function SmallUser() {
    this.name = "Вася";
    return; // <-- возвращает this
}
alert( new SmallUser().name );  // Вася
```

Создание методов в конструкторе

- Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.
- В this мы можем добавлять не только свойства, но и методы.
- Например new User(name) создаёт объект с данным именем name и методом sayHi:

```
function User(name) {
    this.name = name;
    this.sayHi = function() {
      alert( "Меня зовут: " + this.name );
    };
}
let vasya = new User("Вася");
vasya.sayHi();
```

Меня зовут: Вася

Замыкание

- Это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но в JavaScript, все функции изначально являются замыканиями

Вложенные функции

- Функция называется «вложенной», когда она создаётся внутри другой функции.

```
function sayHiBye(firstName, lastName) {
    function getFullName() {
      return firstName + " " + lastName;
    }
    alert( "Hello, " + getFullName() );
    alert( "Bye, " + getFullName() );
}
```

Вложенные функции

- Вложенная функция может быть возвращена: либо в качестве свойства нового объекта (если внешняя функция создаёт объект с методами), либо сама по себе. И затем может быть использована в любом месте. Не важно где, она всё так же будет иметь доступ к тем же внешним переменным.

```
// функция-конструктор возвращает новый объект
function User(name) {
    // методом объекта становится вложенная функция
    this.sayHi = function() {
      alert(name);
    };
}
let user = new User("John");
user.sayHi();
// у кода метода "sayHi" есть доступ к внешней переменной "name"
```

```
function makeCounter() {
    let count = 0;
    return function() {
      return count++;
      // есть доступ к внешней переменной "count"
    };
}
let counter = makeCounter();
alert( counter() );
alert( counter() );
alert( counter() );
```

0

1

2

Массивы

- Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

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

Объявление

```
let arr = [];
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива

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

В массиве могут храниться элементы любого типа.

```
// разные типы значений
let arr = [ 'Яблоко', { name: 'Джон' },
   true, function() { alert('привет'); } ];

// получить элемент с индексом 1 (объект) и
// затем показать его свойство
alert( arr[1].name ); // Джон

// получить элемент с индексом 3 (функция) и выполнить её
arr[3](); // привет
```

Висячая запятая

- Список элементов массива, как и список свойств объекта, может оканчиваться запятой:

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

Методы pop/push, shift/unshift

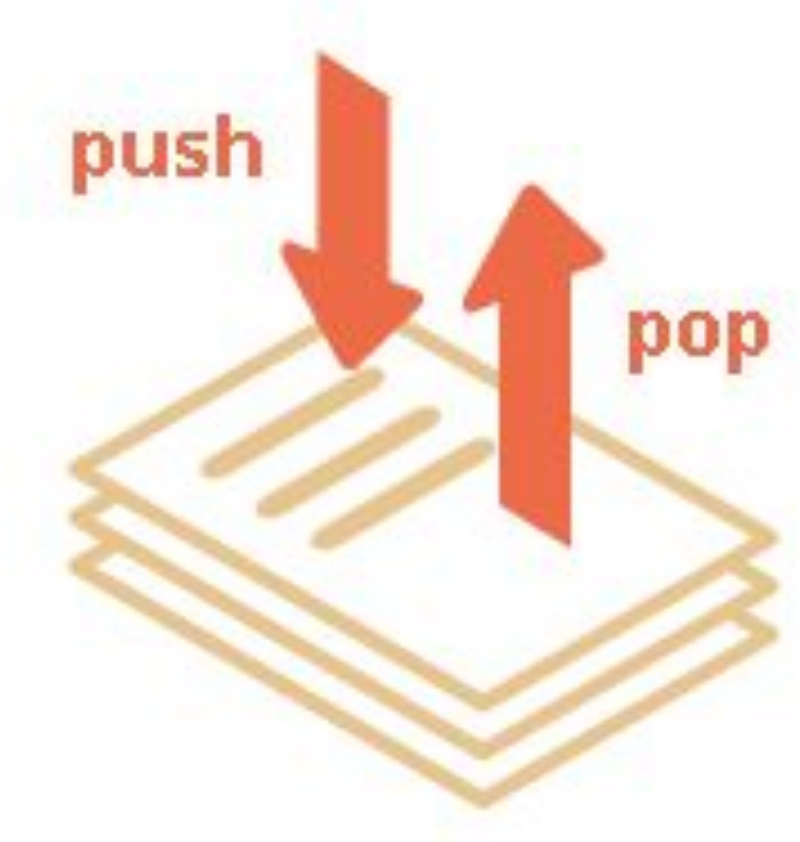
- Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

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

- Массивы поддерживают обе операции.
- На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

- Существует и другой вариант применения для массивов – структура данных, называемая стек.
- Она поддерживает два вида операций:
  - push добавляет элемент в конец.
  - pop удаляет последний элемент.

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



pop

- Удаляет последний элемент из массива и возвращает его:

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

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

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

push

- Добавляет элемент в конец массива

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

fruits.push("Груша");

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

shift

- Удаляет из массива первый элемент и возвращает его:

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

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

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

unshift

- Добавляет элемент в начало массива:

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

fruits.unshift('Яблоко');

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

Методы push и unshift могут добавлять сразу несколько элементов:

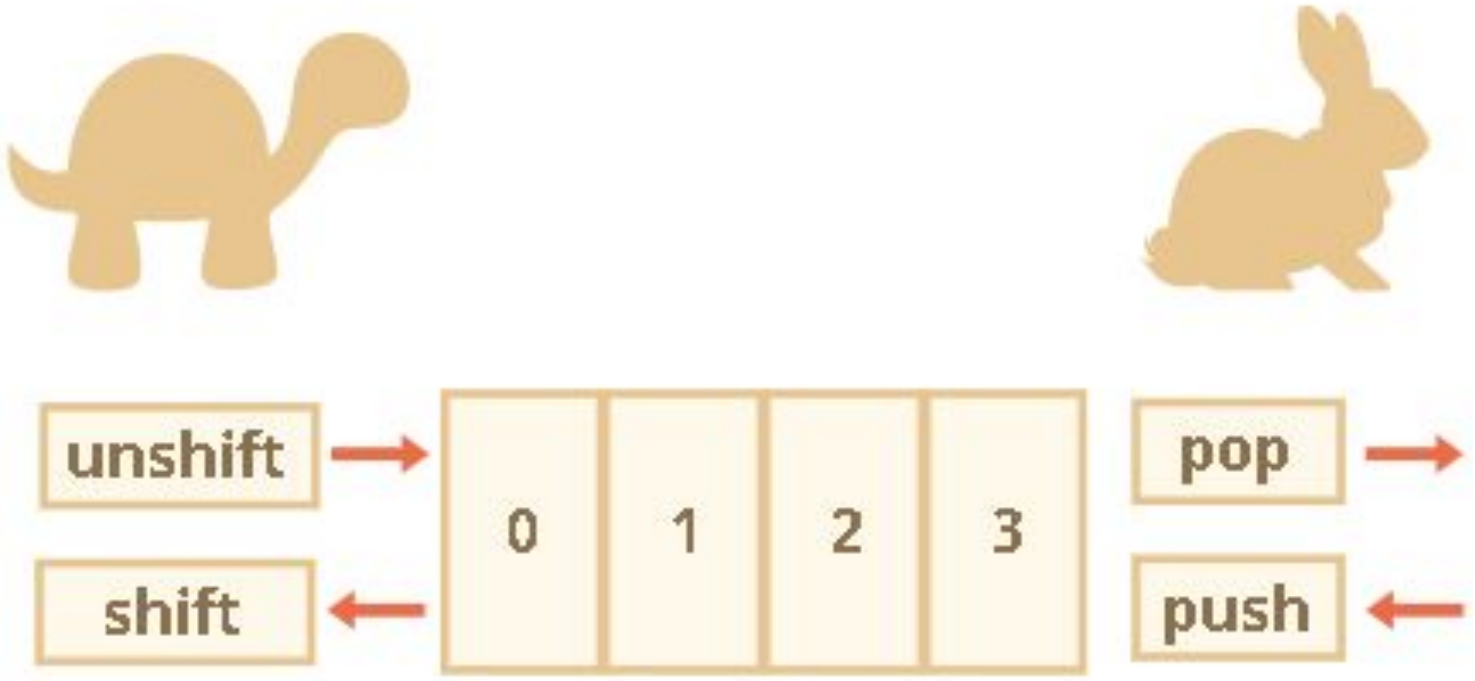
```
let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");
fruits.unshift("Ананас", "Лимон");

// ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша
"]
alert( fruits );
```

Эффективность

- Методы push/pop выполняются быстро, а методы shift/unshift – медленно.



Перебор элементов

- Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

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

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

Другой вариант цикла, for..of:

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

// проходит по значениям
for (let fruit of fruits) {
  alert( fruit );
}
```

Технически, так как массив является объектом, можно использовать и вариант for..in

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

for (let key in arr) {
  alert( arr[key] ); // Яблоко, Апельсин, Груша
}
```

- Цикл for..in выполняет перебор всех свойств объекта, а не только цифровых.
- В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые выглядят, как массив. То есть, у них есть свойство length и индексы, но они также могут иметь дополнительные нечисловые свойства и методы, которые нам обычно не нужны. Тем не менее, цикл for..in выведет и их. Поэтому, если нам приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

- Цикл for..in оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест. Но мы всё же должны представлять разницу.
- В общем, не следует использовать цикл for..in для массивов.

Дата и время

- Встречайте новый встроенный объект: Date. Он содержит дату и время, а также предоставляет методы управления ими.
- Например, его можно использовать для хранения времени создания/изменения, для измерения времени или просто для вывода текущей даты.

Создание

- Для создания нового объекта Date нужно вызвать конструктор new Date() с одним из следующих аргументов:

```
new Date()
```

- Без аргументов – создать объект Date с текущими датой и временем:

```
let now = new Date();
alert( now ); // показывает текущие дату и время
```

```
new Date(milliseconds)
```

- Создать объект Date с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// теперь добавим 24 часа и получим 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

- Датам до 1 января 1970 будут соответствовать отрицательные таймстампы, например:

```
// 31 декабря 1969 года
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

Получение компонентов даты

- Существуют методы получения года, месяца и т.д. из объекта Date:

getFullYear()

Получить год (4 цифры)

getMonth()

Получить месяц, от 0 до 11.

getDate()

Получить день месяца, от 1 до 31, что несколько противоречит названию метода.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Получить, соответственно, часы, минуты, секунды или миллисекунды.

JSON

- JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом.

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

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js’]
};
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"],
}*/
```

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

user = JSON.parse(user);

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