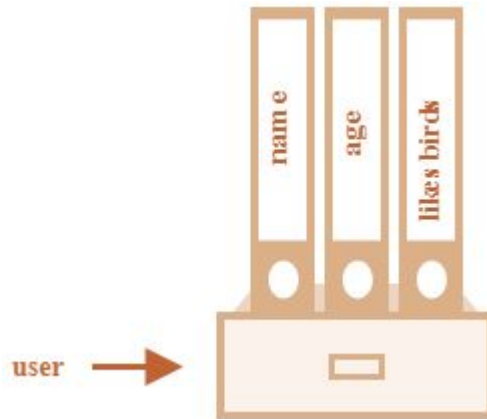


The slide features decorative geometric patterns in the corners. The top-left and bottom-right corners contain overlapping blue and dark blue shapes with gold outlines and a pattern of small blue dots. The text is centered on a white background.

JavaScript

Object, class, prototypes, inheritance

Объекты

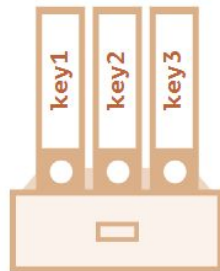


Объекты используются для хранения коллекций различных значений и более сложных сущностей.

В JavaScript объекты используются очень часто, это одна из основ языка.

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.



Пустой объект можно создать, используя один из двух вариантов синтаксиса:

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



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

```
1 let user = {      // объект
2   name: "John",   // под ключом "name" хранится значение "John"
3   age: 30          // под ключом "age" хранится значение 30
4 };
```

Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

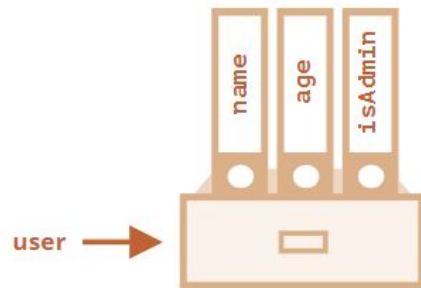
1. Первое свойство с именем **"name"** и значением **"John"**.
2. Второе свойство с именем **"age"** и значением **30**.

Для обращения к свойствам используется запись «через точку»:

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

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

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



Для удаления свойства мы можем использовать оператор **delete**:

```
1 delete user.age;
```

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

```
1 let user = {  
2   name: "John",  
3   age: 30,  
4   "likes birds": true // имя свойства из нескольких слов должно быть в кавычках  
5 };
```

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

```
1 // это вызовет синтаксическую ошибку
2 user.likes birds = true
```

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

```
1 let user = {};
2
3 // присваивание значения свойству
4 user["likes birds"] = true;
5
6 // получение значения свойства
7 alert(user["likes birds"]); // true
8
9 // удаление свойства
10 delete user["likes birds"];
```


Оператор in

В JavaScript оператор **in** используется для проверки наличия указанного свойства в объекте. Он возвращает **true**, если свойство существует в объекте, и **false**, если его нет.

```
const person = {  
  name: "John",  
  age: 30,  
  city: "New York"  
};
```

```
console.log("name" in person); // true
```

```
console.log("gender" in person); // false
```

Reference type

```
let person = {  
  firstName: 'John',  
  lastName: 'Doe'  
};
```



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»**.

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

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

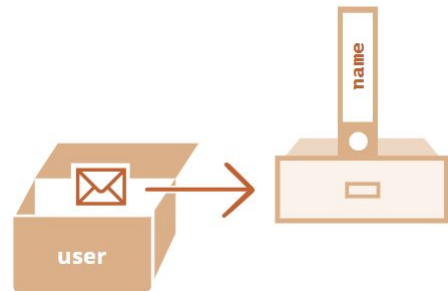


Объекты ведут себя иначе, чем примитивные типы.

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

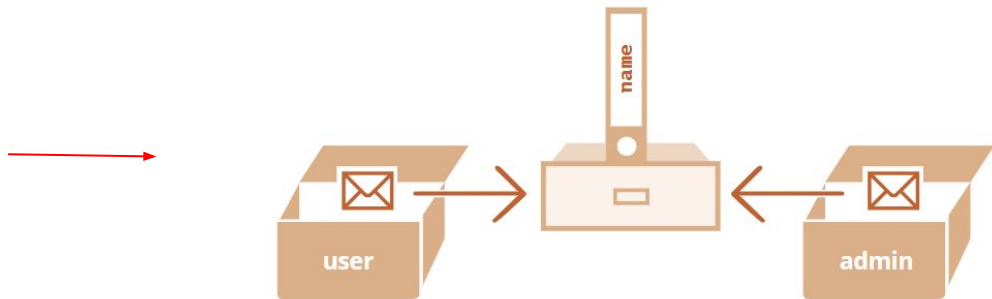
```
1 let user = {  
2   name: "John"  
3 };
```

Вот как это на самом деле хранится в памяти:



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

```
1 let user = { name: "John" };  
2  
3 let admin = user; // копируется ссылка
```



Ключевое слово - this



Сокращённая запись метода в объекте

```
1 // эти объекты делают одно и то же
2
3 user = {
4   sayHi: function() {
5     alert("Привет");
6   }
7 };
8
9 // сокращённая запись выглядит лучше, не так ли?
10 user = {
11   sayHi() { // то же самое, что и "sayHi: function(){...}"
12     alert("Привет");
13   }
14 };
```

Ключевое слово «this»

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.

```
1 let user = {  
2   name: "John",  
3   age: 30,  
4  
5   sayHi() {  
6     // "this" - это "текущий объект".  
7     alert(this.name);  
8   }  
9  
10 };  
11  
12 user.sayHi(); // John
```

Ключевое слово «this»

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:

```
1  let user = { name: "John" };
2  let admin = { name: "Admin" };
3
4  function sayHi() {
5    alert( this.name );
6  }
7
8  // используем одну и ту же функцию в двух объектах
9  user.f = sayHi;
10 admin.f = sayHi;
11
12 // эти вызовы имеют разное значение this
13 // "this" внутри функции - это объект "перед точкой"
14 user.f(); // John (this == user)
15 admin.f(); // Admin (this == admin)
```


Прототипное наследование

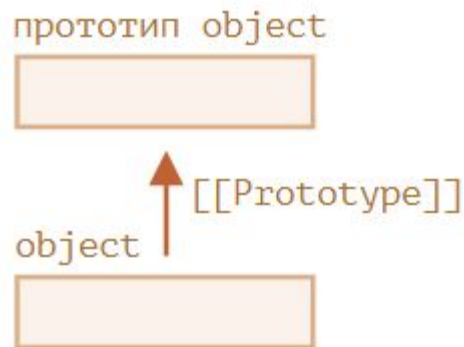


Часто в программировании мы часто хотим взять что-то и расширить.

Например, у нас есть объект **user** со своими свойствами и методами, и мы хотим создать объекты **admin** и **guest** как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта **user**, не копировать/переопределять его методы, а просто создать новый объект на его основе.

Прототипное наследование — это возможность языка, которая помогает в этом.

В JavaScript объекты имеют специальное скрытое свойство `[[Prototype]]` (так оно названо в спецификации), которое либо равно `null`, либо ссылается на другой объект. Этот объект называется «**прототип**»



Когда мы хотим прочитать свойство из **object**, а оно отсутствует, JavaScript автоматически берёт его из прототипа.

Свойство `[[Prototype]]` является внутренним и скрытым, но есть много способов задать его.

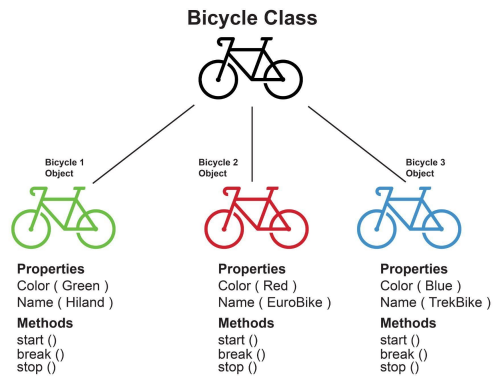
Одним из них является использование `__proto__`, например так:

```
1  let animal = {  
2    eats: true  
3  };  
4  let rabbit = {  
5    jumps: true  
6  };  
7  
8  rabbit.__proto__ = animal;
```

Если мы ищем свойство в **rabbit**, а оно отсутствует, JavaScript автоматически берёт его из **animal**.

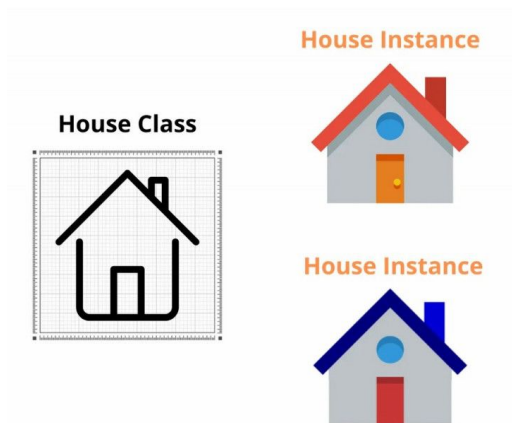
```
1  let animal = {  
2    eats: true  
3  };  
4  let rabbit = {  
5    jumps: true  
6  };  
7  
8  rabbit.__proto__ = animal; // (*)  
9  
10 // теперь мы можем найти оба свойства в rabbit:  
11 alert( rabbit.eats ); // true (**)  
12 alert( rabbit.jumps ); // true
```

Классы



На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё.

В JavaScript есть и более продвинутая конструкция «**class**», которая предоставляет новые возможности для создания объектов одного вида.



Синтаксис

Для создания класса используется ключевое слово `class`. В классе можно определить конструктор и методы.

```
class Person {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    sayHello() {  
        console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);  
    }  
}
```

Конструктор - это специальный метод в классе, который вызывается при создании экземпляра объекта. Конструкторы используются для инициализации объекта, устанавливая начальные значения его свойств и выполняя другие необходимые действия.

Создание экземпляра класса

Чтобы создать экземпляр класса, используется ключевое слово **new** и вызов конструктора.

```
const person1 = new Person("John", 25);
```

```
const person2 = new Person("Alice", 30);
```

```
person1.sayHello(); // "Hello, my name is John and I'm 25 years old."
```

```
person2.sayHello(); // "Hello, my name is Alice and I'm 30 years old."
```

Наследование классов

Наследование классов – это способ расширения одного класса другим классом.

Таким образом, мы можем добавить новый функционал к уже существующему.

Для наследования используется ключевое слово **extends**

Шаг 1. Создаём класс Animal

```
1 class Animal {  
2   constructor(name) {  
3     this.speed = 0;  
4     this.name = name;  
5   }  
6   run(speed) {  
7     this.speed = speed;  
8     alert(`${this.name} бежит со скоростью ${this.speed}.`);  
9   }  
10  stop() {  
11    this.speed = 0;  
12    alert(`${this.name} стоит неподвижно.`);  
13  }  
14 }  
15  
16 let animal = new Animal("Мой питомец");
```

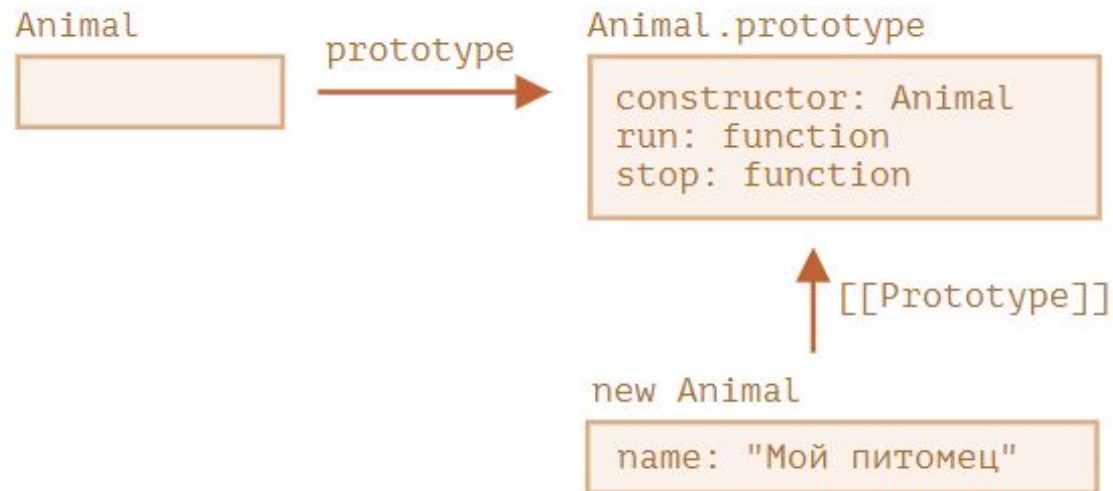
Наследование классов

Шаг 2. Создаём class Rabbit, который наследуется от Animal:

```
1 class Rabbit extends Animal {  
2   hide() {  
3     alert(`${this.name} прячется!`);  
4   }  
5 }  
6  
7 let rabbit = new Rabbit("Белый кролик");  
8  
9 rabbit.run(5); // Белый кролик бежит со скоростью 5.  
10 rabbit.hide(); // Белый кролик прячется!
```

Наследование классов

Схема наследования



Ключевое слово `super` в наследовании классов

Ключевое слово `super` используется в контексте классов и наследования. Оно предоставляет доступ к методам родительского класса из дочернего класса и используется для вызова конструктора родительского класса и доступа к его методам.

Конструкторы в наследуемых классах должны обязательно вызывать `super(...)`, и (!) делать это перед использованием `this..`

Разница в следующем:

Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его `this`.

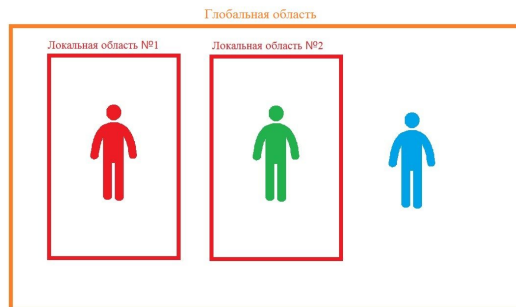
Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.


```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
}
```

```
class Rabbit extends Animal {  
    constructor(name, earLength) {  
        super(name);  
        this.earLength = earLength;  
    }  
}
```

```
let rabbit = new Rabbit("Белый кролик", 10);  
alert(rabbit.name); // Белый кролик  
alert(rabbit.earLength); // 10
```


Области видимости переменных





Область видимости определяет, где в коде программы будут доступны переменные и функции. В JavaScript есть два типа области видимости — глобальная и локальная

Переменные **let** и **const** ведут себя аналогично, в примерах будем употреблять **let**



Блоки кода

Если переменная объявлена внутри блока кода {...}, то она видна только внутри этого блока.

```
function showMessage() {  
  let message = "Hello";  
  alert(message); // Hello  
}
```

```
// Вызываем функцию  
showMessage();
```

```
alert(message); // ReferenceError: message is not defined
```

Для **if**, **for**, **while** и т.д. переменные, объявленные в блоке кода {...}, также видны только внутри:

```
1  for (let i = 0; i < 3; i++) {  
2    // переменная i видна только внутри for  
3    alert(i); // 0, потом 1, потом 2  
4  }  
5  
6  alert(i); // Ошибка, нет такой переменной!
```

Устаревшее ключевое слово

`var`

Устаревшее ключевое слово "var".

Обычно var не используется в современных скриптах, но всё ещё может скрываться в старых.

На первый взгляд, поведение var похоже на let. Например, объявление переменной:

```
1 function sayHi() {  
2   var phrase = "Привет"; // локальная переменная, "var" вместо "let"  
3  
4   alert(phrase); // Привет  
5 }  
6  
7 sayHi();  
8  
9 alert(phrase); // Ошибка: phrase не определена
```

Для «var» не существует блочной области видимости

Область видимости переменных var ограничивается либо функцией, либо, если переменная глобальная, то скриптом. Такие переменные доступны за пределами блока (if, for, while и т.д).

```
1  if (true) {  
2    var test = true; // используем var вместо let  
3  }  
4  
5  alert(test); // true, переменная существует вне блока if
```

Так как var игнорирует блоки, мы получили глобальную переменную **test**.