



Факультет компьютерных наук

Департамент программной
инженерии

Москва
2025

Наследование



Функции-конструкторы

Функции-конструкторы – обычная функция, которая используется для создания объектов с определенной структурой и поведением.

Она позволяет:

- Создавать экземпляры объектов с одинаковой структурой
- Задавать начальные значения полей для каждого экземпляра
- Определять методы, общие для всех экземпляров

Для создания объекта и сохранения его в переменную используется ключевое слово `new`.

```
1  function Person(name, age) {
2      this.name = name;
3      this.age = age;
4
5      this.someMethod = function() {
6          return `Age is ${this.age}`;
7      }
8
9      this.anonymousMethod = () => {
10         return `Age is ${this.age}, but anonymous`;
11     }
12 }
13
14 const person1 = new Person('Alice', 30);
15 console.log(person1.name); // Alice
16 console.log(person1.someMethod()); // Age is 30
17
```



Что такое this в функциях-конструкторах

Для того, чтобы сослаться на какое-либо поле в создаваемом объекте, используется ключевое слово `this`.

При вызове функции-конструктора с ключевым словом `new` JavaScript работает следующим образом:

1. Создается новый объект согласно алгоритму, написанному в теле функции
2. Созданный объект становится целью ссылки `this`
3. Созданный объект возвращается из функции, если не указан другой `return`.

Если функцию вызвать без `new`, функция-конструктор поведет себя как обычная функция.

```
1  function Person(name, age) {
2      this.name = name;
3      this.age = age;
4
5      this.someMethod = function() {
6          return `Age is ${this.age}`;
7      }
8
9      this.anonymousMethod = () => {
10         return `Age is ${this.age}, but anonymous`;
11     }
12 }
13
14 const person1 = new Person('Alice', 30);
15 console.log(person1.name); // Alice
16 console.log(person1.someMethod()); // Age is 30
17
```



Прототип функции-конструктора

Прототип – свойство-объект функции, используемое для описания поведения создаваемых этой функцией объектов, **даже если они уже созданы**.

Основное удобство прототипов заключается в оптимизации памяти при хранении методов объектов: вместо того, чтобы хранить методы в самом объекте, JavaScript хранит их в функции-конструкторе, и на этот прототип будут ссылаться все созданные этой функцией объекты.

```
1  function Person(name) {
2      this.name = name;
3  }
4
5  // Добавляем метод в прототип
6  Person.prototype.sayHello = function() {
7      console.log(`Hello, my name is ${this.name}`);
8  };
9
10 const alice = new Person("Alice");
11 const bob = new Person("Bob");
12
13 alice.sayHello(); // "Hello, my name is Alice"
14 bob.sayHello();   // "Hello, my name is Bob"
15
```



Прототип объекта

Прототип объекта – это ссылка на другой объект, откуда данный объект может наследовать свойства и методы. Доступ к прототипу можно получить через свойство `__proto__` или метод `Object.getPrototypeOf()`

Прототип у объекта задаётся в момент его создания. В случае создания объекта через функцию-конструктор, то в `__proto__` попадает ссылка на то, что лежит в `prototype` функции-конструктора.

Если использовать литерал при создании объекта, то в прототип попадет пустой объект, прототипом которого в свою очередь будет `null`.

Поэтому `typeof null === 'object'`



```
1 function Animal(type) {
2     this.type = type;
3 }
4
5 Animal.prototype.makeSound = function() {
6     console.log(`${this.type} makes a sound`);
7 };
8
9 const dog = new Animal("Dog");
10
11 // Прототип объекта dog – это Animal.prototype
12 console.log(dog.__proto__ === Animal.prototype); // true
13
14 // dog унаследовал метод makeSound из Animal.prototype
15 dog.makeSound(); // "Dog makes a sound"
16
```



| | prototype | __proto__ |
|-----------------------|--|---|
| У кого есть это поле? | Только у функции | У любого объекта, у функции тоже |
| Роль | Определяет, что будет в __proto__ у создаваемых объектов | Ссылка на прототип объекта |
| Доступ | MyFunc.prototype | myObject.__proto__ или Object.getPrototypeOf(myObject) |
| Когда используется | При создании новых объектов через new | При доступе к <i>унаследованным</i> свойствам и методам |



Как все связано

1. При объявлении функции JavaScript создает свойство самой функции `prototype`. В прототип попадают все методы, общие для всех создаваемых объектов.
2. Когда создается объект вызовом этой функции с `new`, `__proto__` этого объекта становится то, что лежит в `prototype` функции
3. Если вы добавляете ручками методы в `prototype`, они автоматически становятся доступны всем объектам, которые уже созданы, либо будут созданы этой функцией
4. При вызове метода объекта, JavaScript ищет вызываемый метод в самом объекте, затем в прототипе этого объекта, затем в прототипе прототипа и так далее. Если метод не найден ни в одном прототипе из цепочки, мы получим ошибку выполнения и аварийную остановку. Поэтому наследование, используемое JavaScript, называется **прототипным**

```
1 function Car() {}  
2 console.log(Car.prototype); // Пустой объект: {}
```



Как все связано

1. При объявлении функции JavaScript создает свойство самой функции `prototype`. В прототип попадают все методы, общие для всех создаваемых объектов.
2. Когда создается объект вызовом этой функции с `new`, `__proto__` этого объекта становится то, что лежит в `prototype` функции
3. Если вы добавляете ручками методы в `prototype`, ни автоматически становятся доступны всем объектам, которые уже созданы, либо будут созданы этой функцией
4. При вызове метода объекта, JavaScript ищет вызываемый метод в самом объекте, затем в прототипе этого объекта, затем в прототипе прототипа и так далее. Если метод не нашелся ни в одном прототипе из цепочки, мы получим ошибку выполнения и аварийную остановку. Поэтому наследование, используемое JavaScript, называется **прототипным**



```
1 function Car() {}  
2 console.log(Car.prototype); // Пустой объект: {}  
3  
4 const myCar = new Car();  
5 console.log(myCar.__proto__ === Car.prototype); // true
```



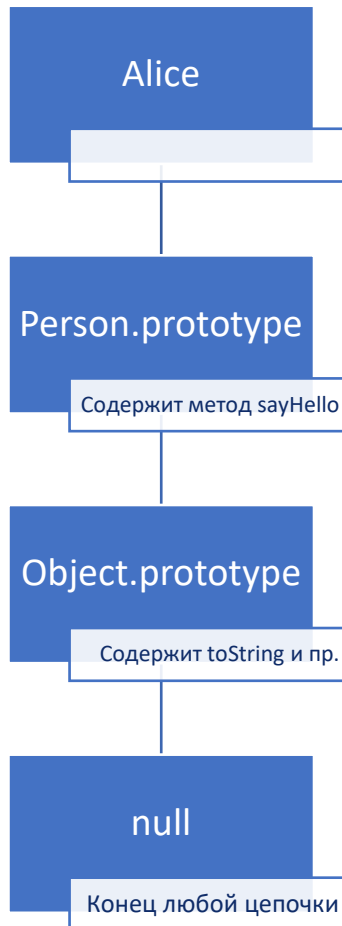

Как все связано

1. При объявлении функции JavaScript создает свойство самой функции `prototype`. В прототип попадают все методы, общие для всех создаваемых объектов.
2. Когда создается объект вызовом этой функции с `new`, `__proto__` этого объекта становится то, что лежит в `prototype` функции
3. Если вы добавляете ручками методы в `prototype`, ни автоматически становятся доступны всем объектам, которые уже созданы, либо будут созданы этой функцией
4. При вызове метода объекта, JavaScript ищет вызываемый метод в самом объекте, затем в прототипе этого объекта, затем в прототипе прототипа и так далее. Если метод не нашелся ни в одном прототипе из цепочки, мы получим ошибку выполнения и аварийную остановку. Поэтому наследование, используемое JavaScript, называется **прототипным**

```
1 function Car() {}
2 console.log(Car.prototype); // Пустой объект: {}
3
4 const myCar = new Car();
5 console.log(myCar.__proto__ === Car.prototype); // true
6
7 Car.prototype.drive = function() {
8     console.log("Driving!");
9 };
10 myCar.drive(); // "Driving!"
11
```



Визуализация цепочки прототипов



```
1 function Person(name) {  
2     this.name = name;  
3 }  
4 Person.prototype.sayHello = function() {  
5     console.log(`Hello, my name is ${this.name}`);  
6 };  
7  
8 const alice = new Person("Alice");  
9
```



Первое задание на семинар

Описание:

Создайте цепочку прототипов, используя функции-конструкторы. Реализуйте базовый конструктор для описания устройства и расширьте его прототип, добавив новый функционал для конкретных типов устройств.

Инструкция:

1. *Создайте функцию-конструктор `Device` с параметрами:*
 1. *`name` (название устройства),*
 2. *`brand` (бренд устройства).*
2. *Добавьте в прототип `Device` метод `getInfo`, который возвращает строку: "Устройство: `[name]`, Бренд: `[brand]`".*
3. *Создайте функцию-конструктор `Phone`, которая наследует свойства и методы `Device` и добавляет параметр `os` (операционная система).*
4. *Установите прототипом для `Phone` прототип из `Device` при помощи метода `Object.create`*
5. *Расширьте прототип `Phone` методом `getFullInfo`, который возвращает строку: "Устройство: `[name]`, Бренд: `[brand]`, ОС: `[os]`".*
6. *Создайте несколько объектов `Phone` и проверьте работу методов `getInfo` и `getFullInfo`.*



Решение

```
1  // Базовый конструктор
2  function Device(name, brand) {
3      this.name = name;
4      this.brand = brand;
5  }
6
7  Device.prototype.getInfo = function () {
8      return `Устройство: ${this.name}, Бренд: ${this.brand}`;
9  };
10
11 // Конструктор-наследник
12 function Phone(name, brand, os) {
13     Device.call(this, name, brand); // Вызов конструктора родителя
14     this.os = os;
15 }
16
17 // Устанавливаем прототип для наследования
18 Phone.prototype = Object.create(Device.prototype);
19
20 // Расширяем прототип Phone
21 Phone.prototype.getFullInfo = function () {
22     return `Устройство: ${this.name}, Бренд: ${this.brand}, ОС: ${this.os}`;
23 };
24
25 // Создаём объекты
26 const device1 = new Phone("Galaxy S21", "Samsung", "Android");
27 const device2 = new Phone("iPhone 13", "Apple", "iOS");
28
29 console.log(device1.getInfo());
30 // "Устройство: Galaxy S21, Бренд: Samsung"
31 console.log(device1.getFullInfo());
32 // "Устройство: Galaxy S21, Бренд: Samsung, ОС: Android"
33
34 console.log(device2.getInfo());
35 // "Устройство: iPhone 13, Бренд: Apple"
36 console.log(device2.getFullInfo());
37 // "Устройство: iPhone 13, Бренд: Apple, ОС: iOS"
```



Потеря контекста выполнения

Контекст (`this`) часто теряется в обработчиках событий в JavaScript из-за того, как именно функции передаются и вызываются в контексте DOM-элементов. Когда вы передаёте метод объекта как обработчик события, контекст (`this`) внутри этого метода больше не ссылается на объект, а вместо этого указывает на элемент, который вызвал событие.

Еще контекст всегда теряется, когда функция передается как коллбек.

```
1  const user = {
2    name: "Alice",
3    greet() {
4      console.log(this.name);
5    }
6  };
7
8  setTimeout(user.greet, 1000);
```

```
1  const user = {
2    name: "Alice",
3    greet() {
4      console.log(`Hi, my name is ${this.name}`);
5    }
6  };
7
8  const button = document.getElementById("btn");
9
10 // Назначаем метод объекта как обработчик
11 button.addEventListener("click", user.greet);
```

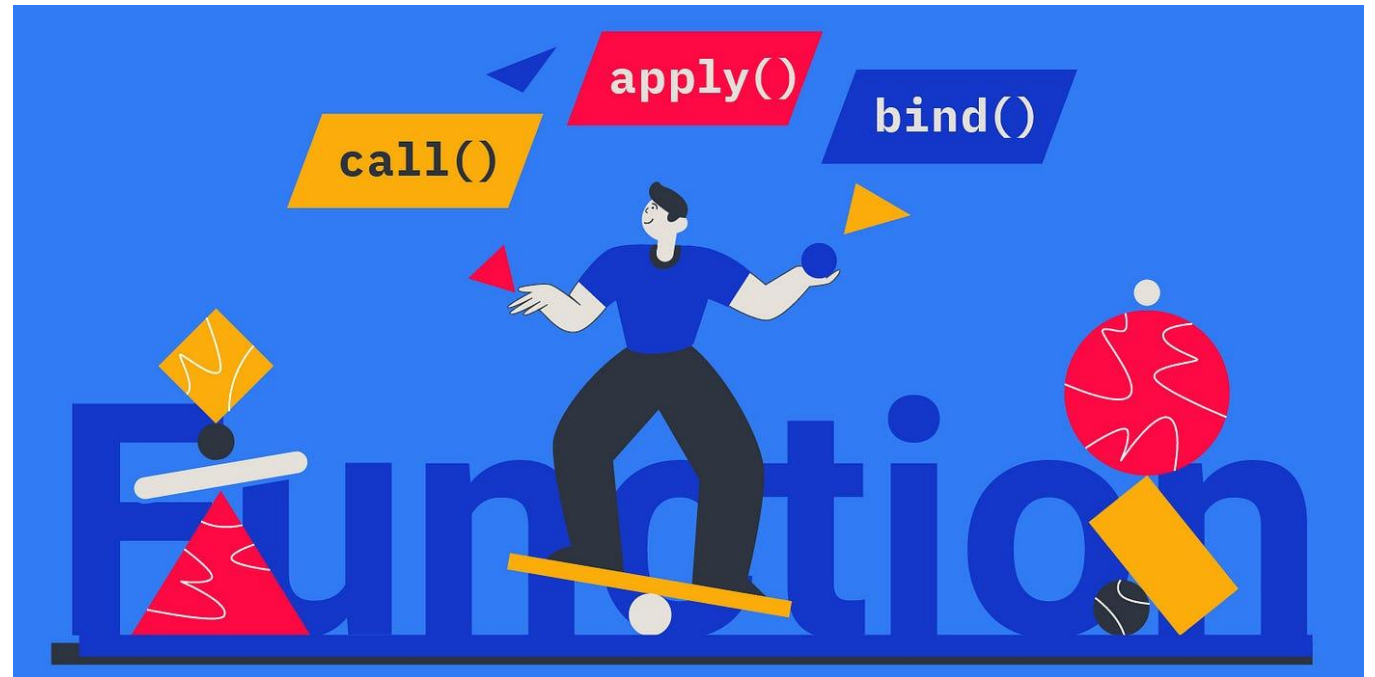


Как восстанавливать контекст

Всего в JS существует 3 метода, явно указывающих на контекст выполнения функций:

- `apply`
- `bind`
- `call`

А еще можно использовать стрелочные функции, у которых нет своего собственного контекста, в качестве значения `this` они берут его из окружения вызова.





Как восстанавливать контекст: `apply()`

Метод **`apply()`** инстанса любой функции вызывает функцию с переданным значением `this` и аргументами для вызываемой функции в виде массива.

Первый аргумент указывает, что будет значением `this` для этой функции. Если мы не в строгом режиме, `null` и `undefined` будут заменены глобальным объектом, а все примитивы будут заменены эквивалентным объектом.

Второй аргумент указывает на массив (или любой итерируемый объект), в котором находятся аргументы для вызова указанной функции.



```
1  const numbers = [5, 2, 9, 1];
2  const max = Math.max.apply(null, numbers);
3  console.log(max); // 9
```



```
1  const array = ["a", "b"];
2  const elements = [0, 1, 2];
3  array.push.apply(array, elements);
4  console.info(array); // ["a", "b", 0, 1, 2]
```



Как восстанавливать контекст: call()

Метод **call()** инстанса любой функции вызывает функцию с переданным значением `this` и аргументами для вызываемой функции в виде перечисления.

Первый аргумент указывает, что будет значением `this` для этой функции. Если мы не в строгом режиме, `null` и `undefined` будут заменены глобальным объектом, а все примитивы будут заменены эквивалентным объектом.

Второй и последующий аргументы указывают на аргументы для вызова указанной функции.



```
1  function greet() {  
2      console.log(this.animal,  
3                  "typically sleep between",  
4                  this.sleepDuration  
5      );  
6  }  
7  
8  const obj = {  
9      animal: "cats",  
10     sleepDuration: "12 and 16 hours",  
11 };  
12  
13 greet.call(obj);  
14 // cats typically sleep between 12 and 16 hours  
15
```




Как восстанавливать контекст: `bind()`

Метод **`bind()`** инстанса любой функции возвращает ссылку на новую функцию, во время вызова которой в качестве `this` выступает заданное значение. У созданной функции будет отсутствовать `prototype`.

Первый аргумент указывает, что будет значением `this` для этой функции. Если мы не в строгом режиме, `null` и `undefined` будут заменены глобальным объектом, а все примитивы будут заменены эквивалентным объектом.

Второй и последующий аргументы указывают на аргументы для вызова указанной функции.

```
1  const user = {
2      name: "Alice",
3      greet() {
4          console.log(`Hi, ${this.name}`);
5      }
6  };
7
8  document.getElementById("btn")
9      .addEventListener(
10     "click",
11     user.greet.bind(user)
12 );
13
```



Задание на семинар

Описание: напишите функцию, которая вычисляет площадь прямоугольника, используя `this` для доступа к ширине и высоте. Затем вызовите эту функцию с разными контекстами, используя `call`, `apply` и `bind`.

Инструкция:

1. Создайте функцию `calculateArea`, которая возвращает произведение `this.width` и `this.height`.
2. Создайте два объекта:
 1. `rect1` с шириной 10 и высотой 20.
 2. `rect2` с шириной 5 и высотой 8.
3. Используйте:
 1. `call` для вызова функции с контекстом `rect1`.
 2. `apply` для вызова функции с контекстом `rect2`.
 3. `bind` для создания новой функции с привязанным контекстом `rect1`.



Решение

```
1  const calculateArea = function () {  
2      return this.width * this.height;  
3  };  
4  
5  const rect1 = { width: 10, height: 20 };  
6  const rect2 = { width: 5, height: 8 };  
7  
8  // Использование call  
9  console.log(calculateArea.call(rect1)); // 200  
10  
11 // Использование apply  
12 console.log(calculateArea.apply(rect2)); // 40  
13  
14 // Использование bind  
15 const calculateAreaForRect1 = calculateArea.bind(rect1);  
16 console.log(calculateAreaForRect1()); // 200  
17
```



Классы

Классы в JavaScript предоставляют синтаксический сахар для работы с объектно-ориентированным программированием (ООП). Они упрощают создание объектов, работу с наследованием, инкапсуляцией и полиморфизмом. Синтаксис классов **не вводит** новую объектно-ориентированную модель, а предоставляет более простой и понятный способ создания объектов и организации наследования. Давайте разберём ключевые аспекты классов.

```
function Car(name, speed) {  
  this.name = name;  
  this.speed = speed;  
}  
  
Car.prototype.showSpeed = function() {  
  console.log(this.speed);  
}  
  
let audi = new Car('audi', 200);  
audi.showSpeed();
```

Before ES6

```
class Car {  
  constructor(name, speed, color) {  
    this.name = name;  
    this.speed = speed;  
  }  
  
  showSpeed() {  
    console.log(this.speed);  
  }  
}  
  
let audi = new Car('audi', 300, 'red');  
audi.showSpeed();
```

After ES6



Определение классов

Классы можно объявлять также как и функции двумя способами: декларативно и экспрессивно (при помощи выражения).

Если воспользоваться декларативным способом, то потребуется использовать ключевое слово `class` и указать имя класса. Обычно класс именуют с заглавной буквы, но это формальность.

Для создания объекта класса сперва в коде указывается описание класса, затем уже выполняется инстанцирование. Поднятие (хойстинг) классов, в отличие от функций, не происходит.



```
1 class Rectangle {  
2     constructor(height, width) {  
3         this.height = height;  
4         this.width = width;  
5     }  
6 }  
7
```



```
1 var p = new Rectangle(); // ReferenceError  
2  
3 class Rectangle {}
```



Определение классов

Второй способ определения класса — **class expression (выражение класса)**. Можно создавать именованные и безымянные выражения. В первом случае имя выражения класса находится в локальной области видимости класса и может быть получено через свойства самого класса, а не его экземпляра.

С таким способом объявления класса поднятия также не будет.

```
1 // безымянный
2 var Rectangle = class {
3     constructor(height, width) {
4         this.height = height;
5         this.width = width;
6     }
7 };
8 console.log(Rectangle.name);
9 // отобразится: "Rectangle"
10
11
12 // именованный
13 var Rectangle = class Rectangle2 {
14     constructor(height, width) {
15         this.height = height;
16         this.width = width;
17     }
18 };
19 console.log(Rectangle.name);
20 // отобразится: "Rectangle2"
```



Описание класса: конструктор

Метод `constructor` — специальный метод, необходимый для создания и инициализации объектов, созданных, с помощью класса. В классе может быть только один метод с именем `constructor`. Исключение типа `SyntaxError` будет выброшено, если класс содержит более одного вхождения метода `constructor`. Если конструктор не определен, то будет использован конструктор по умолчанию (пустой конструктор)

Поскольку классы являются синтаксическим сахаром, на самом деле они все такие же функции, следовательно у них также есть поле `prototype`, благодаря которому можно описывать дополнительные свойства у объектов класса.



```
1 class Rectangle {  
2     constructor(height, width) {  
3         this.height = height;  
4         this.width = width;  
5     }  
6 }
```



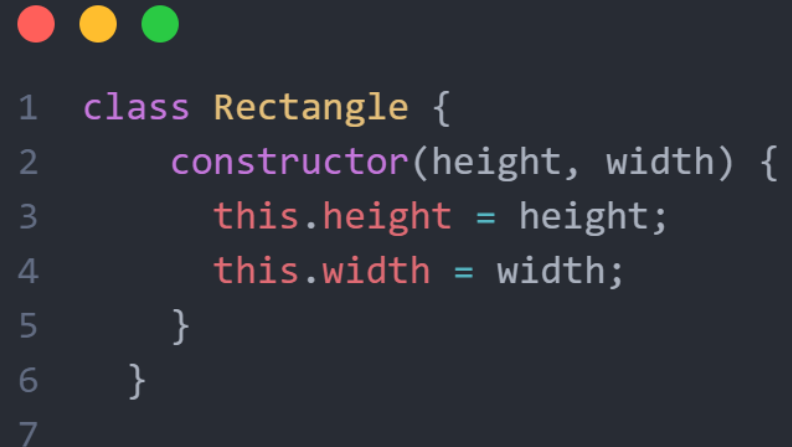
```
1 function Animal(name) {  
2     this.name = name;  
3 }  
4 Animal.prototype.speak = function () {  
5     console.log(`${this.name} издаёт звук.`);  
6 };
```



Описание класса: поля

Поля в классах можно объявлять в любом месте: в описании класса, конструкторе или методе. Если поле описывается в теле класса, достаточно указать его имя по правилам наименования переменной. Можно задать начальное значение полю при помощи оператора присвоения. Если присвоение не выполнять, то до выполнения конструктора значением поля будет undefined.

Все поля и свойства по умолчанию имеют модификатор доступа public и доступны извне. Если название поля или свойства начинается с # - его модификатор доступа становится private и теперь оно доступно только внутри класса. Других модификаторов доступа не существует.



```
1  class Rectangle {  
2      constructor(height, width) {  
3          this.height = height;  
4          this.width = width;  
5      }  
6  }  
7
```




Описание класса: геттеры и сеттеры

Геттер позволяет определить метод, который будет вызываться при чтении значения свойства. Сеттер позволяет определить метод, который будет вызываться при установке значения свойства.

Когда использовать геттеры и сеттеры:

- Для создания **вычисляемых свойств** (например, площадь прямоугольника).
- Когда необходимо **скрыть внутреннюю реализацию** свойства и предоставить контролируемый доступ.

```
1  class Example {
2      constructor(value) {
3          this._value = value;
4      }
5
6      get value() {
7          return this._value; // Возвращаем значение
8      }
9
10     set value(newValue) {
11         if (newValue < 0) {
12             console.error("Значение не может быть отрицательным");
13         } else {
14             this._value = newValue; // Устанавливаем новое значение
15         }
16     }
17 }
18
19 console.log(obj.value); // 10
20 obj.value = 20;
21 console.log(obj.value); // 20
22 obj.value = -5; // "Значение не может быть отрицательным"
```



Описание класса: статика

Ключевое слово **static**, определяет *статический* метод или свойства для класса. Статические методы и свойства вызываются без инстанцирования их класса, и не могут быть вызваны у экземпляров класса.

Статические методы, часто используются для создания служебных функций для приложения, в то время как статические свойства полезны для кеширования в рамках класса, фиксированной конфигурации или любых других целей, не связанных с репликацией данных между экземплярами.

```
1  class Point {
2      constructor(x, y) {
3          this.x = x;
4          this.y = y;
5      }
6
7      static displayName = "Точка";
8      static distance(a, b) {
9          const dx = a.x - b.x;
10         const dy = a.y - b.y;
11
12         return Math.hypot(dx, dy);
13     }
14 }
15
16 const p1 = new Point(5, 5);
17 const p2 = new Point(10, 10);
18 p1.displayName; //undefined
19 p1.distance; //undefined
20 p2.displayName; //undefined
21 p2.distance; //undefined
22
23 console.log(Point.displayName); // "Точка"
24 console.log(Point.distance(p1, p2)); // 7.0710678118654755
25
```



Описание класса: перегрузка методов

JavaScript не поддерживает традиционную перегрузку методов (как, например, в Java или C++), но вы можете имитировать её с помощью проверки аргументов внутри метода.

```
1  class Calculator {
2    calculate(a, b) {
3      if (b === undefined) {
4        // Если второй аргумент отсутствует,
5        // возвращаем квадрат числа
6        return a * a;
7      }
8      // Иначе возвращаем сумму
9      return a + b;
10   }
11 }
12
13 const calc = new Calculator();
14 console.log(calc.calculate(5)); // 25
15 console.log(calc.calculate(5, 3)); // 8
16
```



Описание класса: наследование

Классы могут наследовать друг друга с помощью ключевого слова `extends`. При наследовании можно:

- Переопределять методы;
- Использовать `super` для вызова методов и конструктора родительского класса.

Если вы создаете класс, который наследуется от другого класса, и в новом классе нет конструктора, JS сам вызовет конструктор родительского класса. Если вам нужно объявить новый конструктор, вы обязаны вызвать родительский конструктор при помощи `super()`

```
1  class Animal {
2      constructor(name) {
3          this.name = name;
4      }
5
6      speak() {
7          console.log(`${this.name} makes a noise.`);
8      }
9  }
10
11 class Dog extends Animal {
12     speak() {
13         super.speak(); // Вызов родительского метода
14         console.log(`${this.name} barks.`);
15     }
16 }
17
18 const dog = new Dog("Buddy");
19 dog.speak();
20 // Buddy makes a noise.
21 // Buddy barks.
22
```



Описание класса: абстрактные классы

Абстрактные подклассы, или mix-ins, — это шаблоны для классов. У класса в ECMAScript может быть только один родительский класс, поэтому множественное наследование невозможно. Функциональность должен предоставлять родительский класс.

Для реализации mix-ins в ECMAScript можно использовать функцию, которая в качестве аргумента принимает родительский класс, а возвращает подкласс, его расширяющий:

```
1  const CanFly = Base =>
2    class extends Base {
3      fly() {
4        console.log(`${this.name} is flying!`);
5      }
6    };
7
8  class Bird {
9    constructor(name) {
10      this.name = name;
11    }
12  }
13
14  class Eagle extends CanFly(Bird) { }
15
16  const eagle = new Eagle("Eagle");
17  eagle.fly(); // "Eagle is flying!"
```



Задание на семинар

Описание: создайте класс для описания банковского счета. Используйте все основные возможности классов: конструкторы, методы, статические методы, наследование, приватные свойства, геттеры и сеттеры.

Инструкция:

1. Создайте базовый класс BankAccount с полями:
 1. owner (владелец счета),
 2. balance (баланс).
2. Реализуйте:
 1. Приватное свойство balance для хранения баланса.
 2. Геттер getBalance для получения значения баланса.
 3. Сеттер setBalance для изменения баланса (баланс не может быть отрицательным).
 4. Метод deposit(amount) для пополнения баланса.
 5. Метод withdraw(amount) для снятия денег (если недостаточно средств, вывести ошибку).
3. Добавьте статический метод compareAccounts(account1, account2), который сравнивает два счета по балансу.
4. Создайте класс-наследник SavingsAccount, который добавляет возможность начисления процентов (applyInterest(rate)).
5. Создайте объекты и продемонстрируйте работу всех методов.



Решение

```
1 class BankAccount {
2     #balance = 0;
3
4     constructor(owner, initialBalance) {
5         this.owner = owner;
6         this.#balance = initialBalance;
7     }
8
9     get balance() {
10         return this.#balance;
11     }
12
13     set balance(value) {
14         if (value < 0) {
15             console.error("Баланс не может быть отрицательным");
16             return;
17         }
18         this.#balance = value;
19     }
20
21     deposit(amount) {
22         this.#balance += amount;
23     }
24
25     withdraw(amount) {
26         if (this.#balance < amount) {
27             console.error("Недостаточно средств");
28             return;
29         }
30         this.#balance -= amount;
31     }
32
33     static compareAccounts(account1, account2) {
34         return account1.balance - account2.balance;
35     }
36 }
```

```
1 class SavingsAccount extends BankAccount {
2     applyInterest(rate) {
3         this.balance += this.balance * (rate / 100);
4     }
5 }
6
7 // Создание объектов
8 const account1 = new BankAccount("Alice", 500);
9 const account2 = new SavingsAccount("Bob", 1000);
10
11 account1.deposit(200);
12 account2.applyInterest(5);
13
14 console.log(account1.balance); // 700
15 console.log(account2.balance); // 1050
16
17 console.log(BankAccount.compareAccounts(account1, account2)); // -350
```

