Занятие 5 (13). Ключевое слово THIS, NEW,

методы функций call(), apply(), bind()

THIS

**function declaration:**

f() {…}

**arrow function:**

let f = () => {}

|  |  |
| --- | --- |
| **Declaration f() {…}** | **Arrow let func = () => {}** |
| 1. Число аргументов не фиксировано, можно использовать Объект **arguments** — это подобный массиву объект, который содержит аргументы, переданные в функцию. 2. Значение **this** зависит от того от чьего имени была **вызвана** функция:   - **обычное выполнение:** this - глобальный объект window в браузере (*объект слева от точки)*  - **вызов как метода объекта:** this - это объект, которому принадлежит метод  - **использование call() и apply():** this - это первый аргумент  3. **Нужно явно задавать return** (либо не задавать его, вернется undefined)  4. **Есть методы функций**: call(), apply(), bind()  5. **Есть Prototype**  6. **Есть super()** | 1. Фиксированное число аргументов. Как и со значение this массив arguments для стрелочных функций будет браться из внешней функции. 2. Не имеет значения, как она была вызвана, значение **this** внутри стрелочной функции **всегда эквивалентно значения this внешней функции в момент ее объявления**. Функция не создает собственный контекст исполнения, она использует внешний. 3. Если стрелочная функция содержит в теле одну инструкцию, и ты опустил фигурные скобки, тогда выражение будет **возвращено автоматически**. 4. **Нет методов функций** call(), apply(), bind() 5. **Нет Prototype** 6. **Нет Super ()** |

**Совет по использованию**: в основном надо пользоваться **declaration**, **стрелочной** функцией только тогда, **когда важно сохранение контекста**, чтобы не привязывать каждый раз функцию к новому объекту.

В **setTimeout** и **setInterval** this привязывается к window, т.к. они оба являются браузерными API, но если браузерные API вызваны в методе объекта, то declaration примет this от window, а стрелочная примет от метода(this будет объект метода)

Объявление функции происходит до срабатывания setTimeout!

**this** **стрелочной** функции там, где она **написана**!!! Если стрелочная функция - метод объекта, что this будет window, т.к. объект не создает область видимости.

**this** **обычной** функции там, где она **вызвана**!!!

В **Typescript** this надо выносить в аргументы и там типизировать:

grunt.registerMultiTask('clean', function(**this: {files:any[]}**) {

this.files.forEach(function(f) { Delete(f); });

});

Метод bind() (возвращает новую привязанную функцию)

Допустим, у объекта есть метод. Как можно применить этот метод к другому объекту?

*let* obj1 = {

    name: 'Eugene',

    age: 32,

    sayName() {

        console.log(this.name);

    }

}

*let* obj2 = {

    name: 'Eugene',

    age: 32

}

Есть два варианта решения:

1. Напрямую передать ссылку: obj2.sayName = obj1.sayName. Но в этом случае мы расширяем объект! Мутируем его.
2. Использовать методы функций.

Метод **bind(контекст, аргументы)** ***создаёт новую******привязанную*** *функцию*, которая при вызове устанавливает в качестве контекста выполнения this предоставленное значение.

Например:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

let obj1 = { // объект1

name: 'Eugene',

age: 32,

sayName(arg1, arg2) {

console.log(this.name, arg1, arg2);

}

}

let obj2 = { // объект1

name: 'Eugene',

age: 32

}

let nameFunc = obj1.sayName.bind(obj1, 10, ‘hello’) // байндим с obj1 в качестве this и аргументы. В этом случае жестко привязывается контекст и аргументы.

nameFunc() // вызов функции даст ‘Eugene, 10, hello’

//или

let nameFunc1 = obj1.sayName.bind(obj1) // байндим с obj1 в качестве this, без аргументов. В этом случае аргументы передаем при вызове функции

nameFunc1(15, 'Mike')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Методы функций call(), apply() - делают вызов функции на месте

**Метод call() вызывает функцию с указанным значением this и индивидуально предоставленными аргументами через запятую.**

**Метод apply() вызывает функцию с указанным значением this и индивидуально предоставленными аргументами в виде массива**

**Больше разницы нет!!**

Например:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

let obj1 = { // объект1

name: 'Eugene',

age: 32,

sayName(arg1, arg2) {

console.log(this.name, arg1, arg2);

}

}

let obj2 = { // объект1

name: 'Eugene',

age: 32

}

**// CALL**

obj1.sayName.call(obj2) // в консоль выведется ‘Eugene, undefined, undefined’

obj1.sayName.call(obj2, 20, 'Stan') // в консоль выведется ‘Eugene, 20, ‘Stan’’

**// APPLY**

obj1.sayName.apply(obj2) // в консоль выведется ‘Eugene, undefined, undefined’

obj1.sayName.apply(obj2, [20, 'Stan']) // в консоль выведется ‘Eugene, 20, ‘Stan’’

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**В реальных проектах на React в классовых компонентах контекст выполнения лучше привязывать стрелочными функциями. Т.е. в качестве метода пишем стрелочные функции и все.**

Если в качестве методов используется function declaration, то надо прописывать bind намеренно, чтобы сохранился контекст.

Как еще можно использовать call и apply ? Например, можно украсть метод у объекта и применить его к той сущности, которая не обладает таким методом. Например, псевдомассив из querySelectorAll или arguments:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[].filter.call(arr) - функция это тоже объект, от нее можно привязать this

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Занятие 6 (14). Классы

**SUMMARY:**

1. Constructor() - это обязательная часть класса, отвечает за создание будущего экземпляра объекта, т.е. создается this, и в этот this мы сетам параметры, которые может принимать конструктор. В Конструкторе нужно указывать ТОЛЬКО СВОЙСТВА ОБЪЕКТА. Методы лучше не указывать, т.к. каждая функция будет копироваться, а это сжирает ресурсы.
2. При наследовании родительский класс теряет возможность создавать this, поэтому пользуемся словом super() - это ссылка на родительский конструктор.
3. При наследовании можно переопределять методы, последним будет самый дочерний.
4. Главное отличие функций-конструкторов от классов в том, что у классов есть this!

Используется в основном в ООП.

Основной подход ООП - от общего к частному.

**Первый общий класс** - самолет. Далее делаем уточнения.

**Второй класс** **наследования** - Самолет грузовой или пассажирский, например. Но все они принадлежат к классу самолет.

**Третий класс наследования** - конкретная модель самолета.

При наследовании мы забираем общие вещи, но если что-то не подходит, мы эти методы/свойства **переопределяем**

Такой подход можно применить, например, при создании кнопки. Есть общий класс Кнопка, на основе ее создаются другие Кнопки и т.д.

*Класс* ***типизируется*** *внутри класса:*

*class* Test {

  name: *string* // типизация делается внутри класса

*constructor*(*name*: *string*) {

    this.sayHi = *function* () {}

  }

}

*let* testObj = new Test('Mike')

**Метод лучше объявлять вне конструктора,** чтобы не создавалось куча одинаковых копий функций:

*class* Test {

  name: *string* // типизация делается внутри класса

*constructor*(*name*: *string*) {

    this.name = *name*

    this.sayHi = *function* () { }

  }

  sayHi() { } // можно использовать function declaration

  sayBy = () *=>* { } // можно использовать стрелочные функциию

  //При этом this никогда не переназначится, будет всегда привязано к созданному объекту

}

*let* testObj = new Test('Mike')

**Наследование класса.**

Когда мы используем наследование классов, первым делом нужно использовать конструктор constructor() super(), но в данном случае *конструктор теряет способность создавать this, поэтому используется super() - является ссылкой на конструктор родителя. Вызывая super() мы вызываем конструктор родителя. This создает только базовый класс и чтобы к нему обратиться мы вызываем super()*

*class* Test {

  name: *string* // типизация делается внутри класса

*constructor*(*name*: *string*) {

    this.name = *name*

    this.sayHi = *function* () { }

  }

  sayHi() { }

}

*class* Test2 extends Test {

*constructor*(*name*: *string*) {

    super(*name*)

  }

  sayBye() {}

}

*let* testObj = new Test2('Mike')

При этом у созданного объекта будет метод sayBye и унаследованный sayHi

*class* Test {

*constructor*(*name*: *string*) {

    this.name = *name*

     }

  sayHi() { }

}

*class* Test2 extends Test {

  //@ts-ignore

*constructor*(*name*) {

    super(*name*)

  }

  sayBye() {}

}

*class* Test3 extends Test2 {

  //@ts-ignore

*constructor*(*name*, *age*) {

    super(*name*)

    this.age = *age*

  }

  sayBye() {}

  sayNo() {}

}

*let* testObj = new Test3('Mike', 30)

Метод sayBye перезаписывается от класса с самого верхнего уровня, т.е. от Test3.

Если нам нужно, чтобы метод не переопределялся, его можно взять у родительского класса (только на одну ступень в глубину). В этом случае Test3 будет выглядеть следующим образом:

*class* Test3 extends Test2 {

*constructor*(*name*, *age*) {

    super(*name*)

     this.age = *age*

  }

  sayBye() {

    super.sayBye()

   }

  sayNo() { }

}

Если свойства не создаются, то слово **super() можно не использовать:**

*class* Helper {

  helperMethod1() { }

  helperMethod2() { }

}

*class* SuperHelper extends Helper {

  helperMethod3() { }

}

*let* obj = new SuperHelper()

В этом случае obj будет обладать методами от обоих классов

Модификаторы доступа Typescript

В JS модификаторы доступа есть только внутри классов.

**По дефолту все свойства - public**

*class* Test {

*constructor*(public *name*: *string*) {

    this.name = *name*

  }

}

Модификатор **private.** К свойствам и методам private можно обратиться только с помощью метода, внешнего доступа напрямую к нему нет. в JS приватность обозначается решеткой хеш , например **#**variableName = 10

*class* Test {

  private param = 10

*constructor*(public *name*: *string*) {

    this.name = *name*

  }

  showParam() { this.param }

}

**Фильтрация.** Например есть массив студентов. *Экземпляры классов не могут воздействовать на другие экземпляры (SOLID), поэтому сортировку надо прописать в родительском классе. Такая функциональность будет только на классе, но ее не будет в экземпляре! Делается это через модификатор* **static.**

- В методе с static , this указывает на сам класс!

- В обычном методе this указывает на экземпляр класса! Поэтому надо обращаться к классу, а не к this.

- Метод static может обратиться только к static свойствам и методам!

*class* Test {

  private param = 10

*constructor*(public *name*: *string*) {

    this.name = *name*

  }

  showParam() {

    console.log(this.param);

    console.log(Test.staticParam); // не видит static параметры, обращаемся через имя класса

  }

  static staticParam = 58 // переменной не будет в инстансе

  static testMethod() {

    this.staticParam // видит только static параметры

  } // метода не будет в инстансе

}

Интерфейсы Typescript (для работы с классами)

Интерфейс в ООП не о типизации, а о структуре, которую должен иметь класс(объект). Используется со словом **implements**

*interface* Itest {

  name: *string*

  sayHi: *Function*

}

*class* Test implements Itest {

  // из интерфейса берeтся только поля, не типизация

  name: *string*

*constructor* (*name*: *string*) {

    this.name = *name*

  }

  sayHi() {}

  sayBye() {} // ошибки не будет, даже если этой функции нет в интерфейсе,

  //т.к. он не запрещает расширение класса

}

Можно имплементировать даже сразу несколько интерфейсов:

*interface* Itest {

  name: *string*

  sayHi: *Function*

}

*interface* Itest2 {

  name: *string*

  sayNo: *Function*

}

*class* Test implements Itest, Itest2 {

  // из интерфейса берeтся только поля, не типизация

  name: *string*

*constructor*(*name*: *string*) {

    this.name = *name*

  }

  sayHi() { }

  sayNo() { }

}

**!Интерфейсы могут наследоваться друг от друга**

**--ТЕОРИЯ из ИСТОЧНИКОВ---**

Классы используются при ООП подходе программирования. Обычные функции там не используются. Любая функция в ООП - это метод какого-нибудь класса.

В объектно-ориентированном программировании **класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы). В JS это разновидность функции.**

Служит для избегания дублирования кода, когда нужно создавать много объектов одинакового вида.

Синтаксис:

**class MyClass {**

**prop = value; // свойство**

**constructor(...) { // конструктор**

**// ...**

**}**

**method(...) {} // метод**

**get something(...) {} // геттер**

**set something(...) {} // сеттер**

**[Symbol.iterator]() {} // метод с вычисляемым именем (здесь - символом)**

**// ...**

**}**

Затем используйте вызов **new MyClass()** для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод **constructor(),** в нём мы можем инициализировать объект.

Геттеры/Сеттеры

Как и в литеральных объектах, в классах можно объявлять вычисляемые свойства, геттеры/сеттеры и т.д.

**class User {**

**constructor(name) {**

**// вызывает сеттер**

**this.name = name;**

**}**

**get name() {**

**return this.\_name;**

**}**

**set name(value) {**

**if (value.length < 4) {**

**alert("Имя слишком короткое.");**

**return;**

**}**

**this.\_name = value;**

**}**

**}**

**let user = new User("Иван");**

**alert(user.name); // Иван**

**user = new User(""); // Имя слишком короткое.**

Можно добавлять **свойства** к классу:

**class User {**

**name = "Аноним";**

**sayHi() {**

**alert(`Привет, ${this.name}!`);**

**}**

**}**

**new User().sayHi();**

Наследование классов

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}.

// Наследуем от Animal указывая "extends Animal"

**class Rabbit extends Animal {**

**hide() {**

**alert(`${this.name} прячется!`);**

**}**

**}**

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать методы класса Animal как и все животные.

[Переопределение методов](https://learn.javascript.ru/class-inheritance" \l "pereopredelenie-metodov)

Если мы определим свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

**class Rabbit extends Animal {**

**stop() {**

**// ...будет использован для rabbit.stop()**

**}**

**}**

…Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово **"super"** для таких случаев.

* **super.method(...)** вызывает родительский метод.
* **super(...)** вызывает родительский конструктор (работает только внутри нашего конструктора).

Проверка классов “instanceof”

Оператор **instanceof**  позволяет проверить, к какому классу принадлежит объект, *с учётом наследования, возвращает TRUE/FALSE и просматривает для проверки цепочку прототипов*

*Для* ***объектов*** *используется* ***instanceof*** *(возвращает true/false)*

*Для* ***примитивов*** *используется* ***typeof*** *(возвращает string)*

**class Rabbit {}**

**let rabbit = new Rabbit();**

**// это объект класса Rabbit?**

**alert( rabbit instanceof Rabbit ); // true**

Инкупсуляция

Инкапсуляция в ООП - отделение внутреннего интерфейса объекта от внешнего.

Это дает:

- **Защита для пользователей**, чтобы они не выстрелили себе в ногу. Если пользователь класса изменит вещи, не предназначенные для изменения извне – последствия непредсказуемы.

**- Поддерживаемость**. Если мы чётко отделим внутренний интерфейс, то разработчик класса сможет свободно менять его внутренние свойства и методы, даже не информируя пользователей…

**- Сокрытие сложности.** Всегда удобно, когда детали реализации скрыты, и доступен простой, хорошо документированный внешний интерфейс.

Для сокрытия внутреннего интерфейса мы используем защищённые или приватные свойства:

*Защищённые поля имеют префикс \_.* Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с \_, только из его класса и классов, унаследованных от него.

*Приватные поля имеют префикс #.* JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса.

Public, private, protected (Typescript)

**public**

Можно использовать Typescript, чтобы определить, какие методы или свойства будут видимы вне класса или нет, для этого используется слово **public.**

**! Все методы и свойства имеют по умолчанию видимость public!**

class Greeter {

public greet() {

console.log("hi!");

}

}

const g = new Greeter();

g.greet();

**protected**

Для того, чтобы у метода или свойства видимость была только из подкласса, в котором они были определены, используется слово **protected**

class Greeter {

public greet() {

console.log("Hello, " + this.getName());

}

protected getName() {

return "hi";

}

}

class SpecialGreeter extends Greeter {

public howdy() {

// OK to access protected member here

console.log("Howdy, " + this.getName());

}

}

const g = new SpecialGreeter();

g.greet(); // OK

g.getName();

Property 'getName' is protected and only accessible within class 'Greeter' and its subclasses.

**Private**

Слово **Private** не дает возможности доступа к методам и свойствам даже из подклассов

class Base {

private x = 0;

}

const b = new Base();

// Can't access from outside the class

console.log(b.x);

Property 'x' is private and only accessible within class 'Base'.

Занятие 7 (15). \_ \_Proto\_ \_, Prototype

**Важное:**

**- \_ \_proto\_ \_ (ссылка на прототип класса) есть у всех объектов, примитивов, всех функций, классов и пр. У одинаковых по типу объектов proto будут равны. Proto показывает, от какого класса или функции-конструктора был создан инстанс.** Например, proto двух объектов равны, двух массивов равны, двух функций равны, двух строк равны и т.д. Другими словами, когда сущность создана с вызовом одинакового класса new Object, то у них proto будут равны.

**- prototype** есть только у **class** или **function (у стрелочных функций нет!). Это значит, что все, что не имеет prototype не может выступать в качестве средства для создания объектов.**

**- Каждый prototype –** это независимый объект, сам по себе, с определенным набором свойств и методов

- Proto любого объекта ссылается на prototype класса (функции-конструктора), с помощью которой этот объект был создан(сконструирован)

**- В JavaScript все объекты имеют скрытое свойство [[Prototype]], которое является либо другим объектом, либо null.**

**- Если компилятор не находит свойства или метода у объекта, только после этого он пытается найти его у прототипа!**

**- прототипы никак не влияют на this.** Т.е. Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.

**- Цикл for..in проходит не только по собственным, но и по унаследованным свойствам объекта.**

**Видео по теме:** [**https://www.youtube.com/watch?v=b55hiUlhAzI**](https://www.youtube.com/watch?v=b55hiUlhAzI)

\_ \_proto\_ \_  и [[Prototype]] – это свойства объекта

\_ \_proto\_ \_ — не то же самое, что [[Prototype]]. Это геттер/сеттер для него.

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal; // (\*)

или

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

// walk взят из прототипа

rabbit.walk(); // Animal walk

**Применение**

1. Например, у нас уже есть много созданных объектов obj1, obj2, obj3 и т.д., которые мы создавали с помощью одного конструктора. Потом нам понадобилось добавить ко всем объектам метод или свойство. Используя принцип наследование, мы можем задать метод или свойство только прототипу, и оно появится у всех экземпляров этого прототипа:

function Test(*name*) { // функция - конструктор this.name = *name*}

Test.prototype.sayHi() = function(){} // появится у всех экземпляров Test (ссылка на функцию, не копия. Это экономит ресурсы)

1. Библиотека создала объект testObj, который мы сами не можем создать. Но мы хотим такой же объект newOb сами. Для этого можно воспользоваться \_\_proto\_\_:

Устаревший синтаксис:

let newObj = new testObj.\_\_proto\_\_.construtor('Mike')

Современный синтаксис:

let objProto = Object.getPrototypeOf(testObj)  
let newObj = new objProto.constructor("Mike")

При этом создастся объект того же класса, с той же функциональностью. Т.е. через прототип инстанса мы можем добраться до конструктора и создать еще один инстанс.

**Задача СВОЙ BIND (встречается на собеседовании)**

Задача: используя методы apply или call напишите свой метод bind.

Bind() – это метод функции.

Прототип функции – это Function. Чтобы у инстанса прототипа появился метод bind(), его нужно расширить функционал Function.

Решение:

Function.prototype.customBind = function(*context*, ...*args*) {  
 const self = this  
 return function(...*args2*: any) {  
 return self.apply(*context*, [...*args*, ...*args2*])  
 }  
}

Применим:

let obj = {  
 name: 'Mike',  
 sayName() {  
 console.log(this.name);  
 }  
}  
  
let obj2 = {  
 name: 'Stan'  
}  
  
let bindedFunc = obj.sayName.customBind(obj2)()