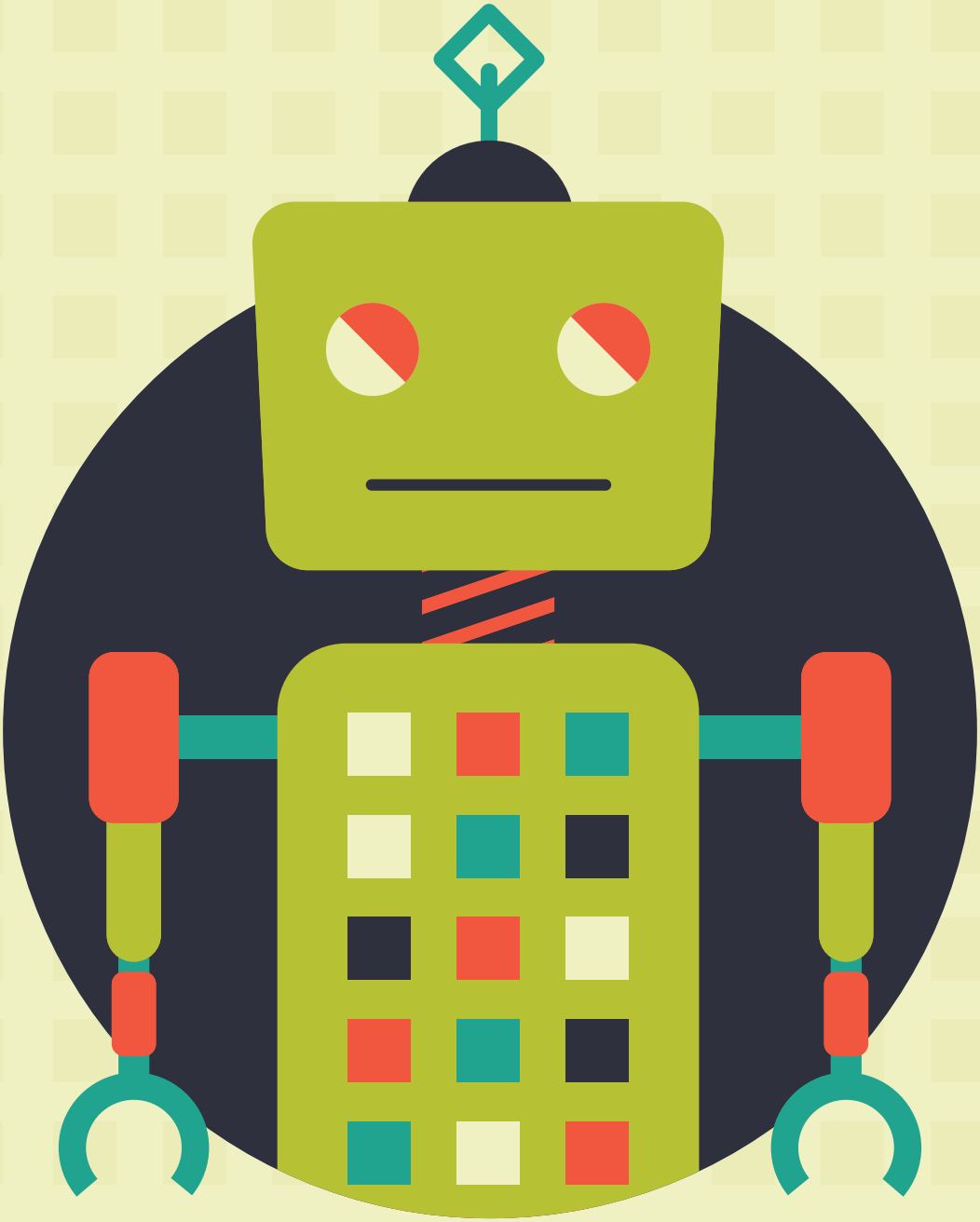

ОПП

LoftSchool
от мыслителя к создателю

Содержание

1

this

Создадим два объекта.

```
var a = {
        prop: 1,
        f: function() {
            console.log(this.prop);
        }
    },
    b = {
        prop: 2,
        f: function() {
            console.log(this.prop);
        }
    };
```

Если мы вызовем функцию **f()** у каждого из объектов, мы увидим чему равно значение свойства prop у каждого из объектов:

```
a.f();
b.f();
```

Теперь более подробно поговорим про ***this***.

Бытует утверждение, что **this** указывает на объект, из которого вызывается текущая функция.

Это мнение является в корне неверным.

this указывает на объект, в контексте которого вызывается функция.

Может возникнуть вопрос - а какая разница?

А разница в том, что контекст функции мы можем менять.

То есть вызывая функцию **f()** из объекта A, мы можем сделать так, чтобы **this**, внутри этой функции, указывал на объект B.

Когда мы вызываем функцию из объекта, контекст этой функции равен тому объекту, из которого мы эту функцию вызываем. Но у каждой функции в JS есть три вспомогательные функции, которые позволяют менять контекст этой функции.

Немного запутанно звучит, но на самом деле всё становится понятно, если представить функцию как самый обычный объект, коим функция и является.

Так вот у этого объекта есть три метода: **bind**, **call** и **apply**.

Эти методы позволяют менять контекст у функции. Пользоваться ими чрезвычайно просто.

Начнем с **bind**.

2

Смена контекста

bind

Bind меняет контекст функции на указанный и возвращает новую функцию, с уже измененным контекстом. Не вызывает указанную функцию, а возвращает новую функцию, с измененным контекстом!

Вот как это работает. Для начала просто сохраним функцию из объекта в переменную

```
var newFunc = a.f;
```

Заметьте, мы не вызываем функцию, а просто сохраняем ее в переменную. Это важно.

Теперь вызовем то, что мы только что сохранили.

```
newFunc(); //выведется undefined
```

Всё потому, что функция потеряла свой контекст и теперь **this**, внутри этой функции указывает не на объект A, а на нечто другое, о чем будет рассказано позже.

Но стоит так же отметить, что в переменной newFunc и в объекте A одна и та же функция.

Просто если вызывать ее из объекта A, то контекстом ее является объект A, а если вызывать ее просто, как отдельную часть, то контекста нет. Точнее он есть, но об этом позже.

Вернемся к функции **bind** и применим ее.

Напомню, что bind возвращает новую функцию с указанным контекстом.

В переменную newFunc запишем результат функции **bind()**

```
var newFunc = a.f.bind(a);
```

Теперь вызовем то, что получилось.

```
newFunc();
```

Как видите, теперь, даже вызывая функцию без объекта, this внутри этой функции указывает на объект, который мы указали при использовании **bind**.

Еще более интересный пример: заменим функцию f в объекте A на ту же функцию, но с другим контекстом.

```
a.f = a.f.bind(b);

a.f();
b.f();
```

Теперь в обоих случаях выводится цифра два. Всё потому, что в первом случае мы заменили контекст на объект B, а во втором, сохранился родной контекст.

Может возникнуть вопрос: а почему бы просто не сделать так:

```
a.f = b.f;
```

То есть подменить функцию f в объекте A, той же самой функцией, но уже из объекта B, в надежде на то, что this этой функции тоже переедет из объекта в объект.

Тут важно понимать, что мы всего лишь заменили функцию, но как только мы будем вызывать эту функцию из объекта, контекстом этой функции станет тот объект, из которого мы эту функцию вызываем. Но функции для смены контекста позволяют это обойти, ведь применив метод bind, мы можем заменить контекст таким образом, чтобы не имело значения из какого объекта мы вызываем функцию.

Так же замечу, что изменить контекст мы можем всего один раз. Вот пример:

```
var newFunc = a.f.bind(b);
newFunc = newFunc.bind(a);

newFunc();
```

Как видим, вторая привязка контекста не сработала.

Это не значит, что у нас есть какие-то ограничения по количеству привязок контекста, это всего лишь значит, что у функции, которая получается в результате **bind**, мы больше не можем менять контекст.

Я надеюсь, что данные примеры стали наглядным доказательством того, что this внутри функции - это ее контекст, а не объект из которого мы эту функцию вызываем.

Кстати, в IE, bind доступна начиная с 9 версии. В более ранних версиях, метод bind эмулировался программно.

Отлично, теперь быстро пробежимся по двум оставшимся методам для смены контекста: **call** и **apply**.

call

Работают они по такому же принципу, как и **bind**, то есть меняют контекст функции, но если **bind** не вызывает функцию с новым контекстом, а просто возвращает ее, то **call** и **apply**, помимо смены контекста, еще и вызывают новую функцию и возвращают то, что возвратила бы эта функция. Но чем же тогда **call** и **apply** различаются между собой?

Приведу очень грубый, но наглядный пример. Предположим, что функция f принимает два аргумента, два числа. Цель этой функции сложить эти два числа с полем **prop**:

```
var a = {
        prop: 1,
        f: function (a, b) {
            console.log(this.prop + a + b);
         }
    },
    b = {
        prop: 2,
        f: function (a, b) {
            console.log(this.prop + a + b);
         }
    };
```

На самом деле, мы можем вообще вынести эту функцию за объекты, чтобы не дублировать ее, а в свойствах объектов просто прописать ссылку на нее.

```
var summ = function(a, b) {
        console.log(this.prop + a + b);
    },
    a = {
        prop: 1,
        f: summ
    },
    b = {
        prop: 2,
        f: summ
    };

a.f(1, 1);
b.f(2, 2);
```

Как видите, при вызове функций, контекст подставляется динамически и поэтому мы наблюдаем довольно предсказуемое поведение.

Теперь применим метод **call**, но для наглядности, уже к функции summ:

```
summ.call(a, 1, 1);
summ.call(b, 2, 2);
```

Первым параметром передается контекст, в котором будет выполняться функция, а остальные переданные аргументы передадутся функции. Таким образом, второй аргумент call станет первым аргументом функции и так далее.

Теперь изменим функцию так, чтобы она не выводила в консоль результат, а возвращала его.

```
var summ = function(a, b) {
        return this.prop + a + b;
    }
```

Теперь **summ.call** будет возвращать результат, который мы можем вывести.

```
console.log(summ.call(a, 1, 1));
console.log(summ.call(b, 2, 2));
```

Это наглядно демонстрирует работу **call**.

apply

Осталась последняя функция - ***apply***. Она работает точно так же как и **call**, разница только в передаче аргументов в функцию.

Если в **call** мы передавали аргументы напрямую, то в **apply** нужно передавать всего лишь массив с аргументами, а **apply** сама развернет этот массив и передаст аргументы в функцию в нужном порядке.

```
console.log(summ.apply(a, [1, 1]));
console.log(summ.apply(b, [2, 2]));
```

Как видите, результат не изменился. В **apply** мы передали аргументы для функции в виде массива, а **apply** сама подставила элементы из массива на нужные позиции. Так, первый элемент массива стал первым аргументом функции и тд.

Отлично, мы познакомились с тройкой методов, для изменения контекста функции.

Теперь выясним чему равен контекст функции, если не вызывать ее из объекта и не применять к ней **bind**, **call** или **apply**.

Контекст по умолчанию

Вызовем функцию summ в самом обычном виде

```
summ(1,1);
```

а внутри функции выведем в консоль **this**

```
console.log(this);
```

и запустим скрипт в браузере, для большей наглядности.

Итак, если функции явно не указать контекст, то по умолчанию он равен глобальному объекту. Если запускать скрипт в браузере, то там глобальный объект всегда равен **window**.

Теперь нужно запомнить вот что:

Контекст у функции может быть всего лишь в трех случаях:

- либо когда мы вызываем функцию из объекта (через точку или квадратные скобки)
- либо когда используем **bind**, **call** или **apply**
- или если при вызове функции использовать ключевое слово **new** (разберем позднее)

Во всех остальных случаях контекст будет сброшен на глобальный объект.

Вот пример, с которым у разработчиков очень часто возникают затруднения:

```
var a = {
   prop: 1,
   f: function() {
       var func = function() {
           console.log(this.prop)
       };
       func();
   }
};

a.f();
```

Внутри функции f контекст, ясное дело, будет указывать на объект A, а какой будет контекст у функции func?

Вспомним то правило, о котором было сказано ранее:

Контекст у функции может быть всего лишь в трех случаях:

- либо когда мы вызываем функцию из объекта (через точку или квадратные скобки)
- либо когда используем **bind**, **call** или **apply**
- если при вызове функции использовать ключевое слово new (разберем позднее)

Функцию func мы вызываем как есть. Не из объекта и не используем **bind**, **call** или **apply**. Следовательно у нее нет контекста, а следовательно **this** будет указывать на **window**, поэтому если мы сейчас запустим скрипт, то увидим **undefined**.

Почему же с этим примером часто возникает недопонимание?

Дело в том, что многие думают, что раз уж мы запускаем функцию из функции, которая имеет контекст, то у этой внутренней функции тоже будет контекст... как бы не так! Сразу вспоминаем то правило!

Решается такая задача очень просто, если вам необходимо передать контекст от основной функции во внутреннюю, то вы либо используете bind, call или apply либо сохраняете контекст в переменную и потом эту переменную используете во внутренней функции.

Разберем оба варианта.

```
func.call(this);
```

Здесь мы просто, так сказать, пробросили контекст во внутреннюю функцию, думаю здесь не нужно больше пояснений.

И второй вариант:

```
var that = this;

var func = function () {
   console.log(that.prop)
};
func();
```

Здесь мы просто сохранили текущий контекст функции в переменную и использовали эту переменную внутри вложенной функции.

О данном подходе еще иногда говорят - передали контекст через замыкание.

Кстати, если вы используете строгий режим, то контекст по умолчанию будет равер undefined.

Ну вот мы и разобрались с контекстом.

Осталось рассказать про одну маленькую особенность метода **bind**.

bind и аргументы

Если при вызове метода **bind**, передать ему дополнительные аргументы, как в случае с **call**, то **bind** запомнит эти аргументы и при вызове получившейся функции - будет подставлять эти аргументы. Вот так это выглядит:

```
var summ = function(a, b) {
   return this.prop + a + b;
};

var newFunc = summ.bind(a, 1, 1);
console.log(newFunc());
```

Как видите, мы вызвали новую функцию вообще без параметров. Параметры подставились автоматически, т.к. **bind** запомнил те значения, которые мы ему передали.

Здесь поведение **bind** совпадает с поведением **call**.

Помните, что нельзя изменить контекст у функции, которую вернул bind?

Но если вызывать **bind** повторно на той же самой функции, которую вернул **bind**, то мы получим довольно интересный эффект. Вот как это выглядит:

Если мы вызовем **bind** с одним аргументом

```
var newFunc = summ.bind(a, 1);
```

и попробуем вызвать получившуюся функцию, то функция вернет нам **NaN**.

Тут всё логично, тк мы передали в функцию summ только один аргумент, а второй аргумент так и остался не определен.

А теперь еще раз вызовем **bind** для полученной ранее функции и снова запустим ее

```
var newFunc = summ.bind(a, 1);
newFunc = newFunc.bind(a, 1);
newFunc();
```

Как видите, функция summ отработал как положено, как если бы мы передали ей оба аргумента.

Дело в том, что применяя bind над функцией, над которой уже был применен bind, происходит накопление аргументов.

То есть при первом вызове **bind** выставился первый аргумент, а при втором - второй.

Данную запись можно записать и короче

```
var newFunc = summ.bind(a, 1).bind(a, 1);
```

при втором вызове **bind**, контекст уже не играет роли, т.к., как говорилось ранее, учитывается только тот контекст, который был передан при первом вызове **bind**.

3

ООП

ООП (объектно-ориентированное программирование) - это концепция построения системы, в основе которой лежат объекты. То есть вся система разбивается на более мелки части и каждая такая часть является объектом со своим набором методов и свойств.

В ООП существует несколько базовых принципов. Давайте начнем с того, что проведем некоторые аналогии.

Представьте художника, который рисует портрет. Перед художником сидит человек, с которого художник рисует портрет. Каждый человек обладает своими особенностями строения тела или уникальными чертами лица. Задача художника - создать копию этого человека как минимум в единственном экземпляре, скопировав все его особенности. Теперь представьте, что художник нарисовал портрет не в одном экземпляре, а в нескольких, например в 4. Затем художнику захотелось на одном из экземпляров подрисовать человеку усы. Он берет кисть и начинает дорисовывать один из экземпляров. Теперь внимание: то, что художник подрисовал усы одному из экземпляров, не означает, что у оставшихся трех экземпляров тоже нарисовались усы. Это так же не означает, что у человека, с которого человек рисовал портрет, вдруг выросли усы. Логично, не правда ли?

Так вот, переходим от аналогии к коду. Человек сидящий перед художником, это класс, который обладает своими свойствами и особенностями, а несколько портретов, которые художник нарисовал, это экземпляры(объекты) этого класса. Из выше сказанного следует, что экземпляры класса никак не могут повлиять на сам класс. Если мы меняем какие-то свойства экземпляра, то они меняются только внутри этого экземпляра, но не у класса. Примером могут служить те же усы, которые художник дорисовал одному их экземпляров. Так же, меняя свойства одного из экземпляров, мы никак не затрагиваем те же свойства у других экземпляров этого класса.

Теперь давайте разберем, что означает фраза “класс наследует от другого класса”.

Представь того же художника, но теперь он рисует портрет не с человека из первого примера, а с его ребенка.

Логично предположить, что ребенок похож на своих родителей какими-то чертами лица или характера. Но тем не менее, у него есть свои особенности, которые отличают его от родителей. Например: цвет глаз. У родителей - карие, а у их ребенка - голубые.

В случае с классами то же самое: есть класс-родитель, который обладает какими-то свойствами, и есть другой класс, производный от него. Производный класс называют дочерним классом. Дочерний класс наследует все свойства родительского класса и имеет возможность переопределить любое из этих свойств так, как ему нужно.

Опять же аналогия: если ребенок вдруг поранится, это ведь не значит, что автоматически поранятся и его родители. Или, если ребенок научился играть на гитаре, это ведь не значит, что его родители тоже автоматически научились это делать. Следовательно - изменение свойств дочерного класса никак не влияет на свойства родительского. Но не наоборот! Если у одного из родителей, например, есть ямочка на подбородке, то высока вероятность, что у ребенка она будет.

С классами то же самое, наделив родительский класс свойствами, эти свойства будут и у дочернего класса.

В JS нет понятия классов. За то есть понятие “**Конструктор**”.

Конструктор - это функция, которая должна создать объект.

4

Конструкторы

Создадим самую обычную функцию:

```
var F = function() {
};
```

Теперь интересный момент - если перед вызовом этой функции поставить ключевое слово **new**, то функция автоматически вернет новый пустой объект.

Убедимся в этом

```
var obj = new F();

console.log(obj);
```

Есть еще один интересный момент - **this** внутри этой функции будет равен как раз этому пустому объекту.

Соответственно, внутри функции мы можем выполнять над этим объектом разные манипуляции, например, создавать новые свойства

```
var F = function(name) {
   this.name = name;
};

var obj = new F('Сергей');
```

Теперь запустим пример и видим, что возвращаемый объект изменился в соответствии с тем, какие манипуляции мы проделали с ним в функции.

И заметьте, что никакого return внутри функции не нужно. При использовании ключевого слова **new**, перед вызовом функции, функция автоматически возвратит новый объект.

Добавим еще один объект и еще один вызов

```
var obj = new F('Сергей'),
    obj2 = new F('Катя');

console.log(obj);
console.log(obj2);
```

Запустим пример. Из этого понятно, что каждый раз, при вызове функции с ключевым словом **new**, возвращается новый объект, со своим набором свойств.

Фактически, внутри этой функции, мы конструируем наш новый объект и поэтому такие функции, носят название "***Конструктор***".

Свойство constructor

У каждого объекта есть свойство ***constructor***, которое указывает на функцию, в которой этот объект был создан.

Выведем свойство **constructor** нашего объекта

```
console.log(obj.constructor);
```

Как видите, в консоли вывелась та самая функция, которая использовалась для создания данного объекта. То есть свойство constructor содержит ссылку на функцию-конструктор данного объекта.

Таким образом, если у каждого типа данных в JS есть такое свойство и мы можем к нему обратиться.

Прототип

Теперь предположим, что мы хотим сконструировать объект так, чтобы у этого объекта был ряд методов... Можно поступить так

```
var F = function (name) {
    this.name = name;

    this.setName = function (name) {
        this.name = name;
    };

    this.getName = function () {
        return this.name;
    };
};
```

Здесь мы добавляем к конструируемому объекту два новых метода. Первый метод устанавливает новое значение для свойства name, а второй методы возвращает текущее значение свойства **name**.

А вместо **this.name = name** используем метод **setName()**

Вот так можно использовать эти методы в дальнейшем

```
var obj = new F('Сергей');

console.log(obj.getName());
obj.setName('Андрей');
console.log(obj.getName());
```

Запустим скрипт и увидим, что сначала выводится имя Сергей, так как именно это имя было выставлено при конструировании объекта.

Затем мы устанавливаем свойству name новое значение “Андрей” и выводим его.

Так можно наполнять объект новыми метода.

Но помимо этого способа, есть еще один - наполнять объект методами через свойство prototype у функции-конструктора.

У любой функции-конструктора есть свойство **prototype** и по умолчанию это просто пустой объект

```
console.log(F.prototype);
```

Но давайте попробуем наполнить этот объект методами, то есть перенесем объявление методов из функции-конструктора в свойство **prototype** этого конструктора

```
var F = function (name) {
   this.setName(name);
};

F.prototype.setName = function (name) {
   this.name = name;
};

F.prototype.getName = function () {
   return this.name;
};
```

и теперь снова попробуем написать тот же код, что и раньше

```
var obj = new F('Сергей');

console.log(obj.getName());
obj.setName('Андрей');
console.log(obj.getName());
```

Как видите, результат остался тем же, что и раньше.

Первым делом, когда конструктор конструирует объект, он автоматически добавляет к этому объекту служебное свойство **__proto__**, которое указывает на свойство **prototype** функции-конструктора.

В этом легко убедиться, выполнив следующий код

```
console.log(obj.__proto__ === F.prototype);
```

Как видите, вывелось **true**

Следовательно, в этом свойстве **__proto__** будет всё то, что мы записывали в **prototype**, ведь это один и тот же объект.

То есть наши методы будут не напрямую в объекте **obj**, а в свойстве **__proto__** внутри него.

Но как же тогда происходит вызов функций?

Поиск свойств в прототипах

Дело в том, что когда мы пытаемся обратиться к какому либо свойству объекта, сначала, это свойство ищется в самом объекте.

Если внутри объекта свойство не было найдено, поиск продолжается в том самом свойстве **__proto__** и так далее.

То есть, вызывая функцию **getName**, сначала интерпретатор ищет ее в самом объекте, и так как функции там нет, то поиск продолжается в свойстве **__proto__** и только там интерпретатор находит эту функцию и выполняет её.

А для разработчика создается впечатление, как будто функция вызывается напрямую из объекта.

Кстати, свойство constructor хранится не в самом объекте, а в свойстве **prototype** конструктора... Подумайте над этим...

5

Наследование

Представим, что нам необходимо добавить еще один тип объектов, которые обладали бы теми же методами, что и объекты типа F, но помимо этого, имели бы еще пару дополнительных методов (например методы для установки и чтения возраста), которые были бы не доступы для F

Вот что приходит на ум в первую очередь

Создаем новый конструктор, который, помимо имени, принимает еще и возраст

```
var F2 = function(name, age) {
};
```

Затем копируем прототип из F в F2

```
F2.prototype = F.prototype;
```

И расширяем прототип у F2 парой новый функций.

```
F2.prototype.setAge = function (age) {
   this.age = age;
};

F2.prototype.getAge = function () {
   return this.age;
};
```

Таким образом, у F2 теперь есть все методы от F и, плюс ко всему, мы добавили парочку новых

Теперь корректно сконструируем объект типа F2

```
var F2 = function(name, age) {
   this.setName(name);
   this.setAge(age);
};
```

Теперь создаем один объект типа F, а второй объект типа F2

```
var obj = new F('Сергей'),
    obj2 = new F2('Андрей', 30);
console.log(obj.getName());
console.log(obj2.getName(), obj2.getAge());
```

Запустим и увидим, что всё работает как надо.

Но всё не совсем так....

Дело в том, что у F тоже появились методы для работы с возрастом. Мы можем в этом убедиться

```
obj.setAge(26);
console.log(obj.getName(), obj.getAge());
```

Но как же так? Ведь предполагалось, что к методам управления возрастом, будут иметь доступ только объекты типа F2.

Ошибка заключается в том, что мы присвоили прототипу F2 прототип от F

Когда в JS один объект присваивается другому - никакого копирования или клонирования объектов не происходит. Вместо этого, оба объекта указывают на одни и те же данные и изменяя один объект, мы тем самым меняем и другой. Как это и произошло при присваивании **prototype**.

Обойти это довольно просто.

Правильное наследование

Для этого напишем функцию, которая будет реализовывать правильное наследование - создавать между прототипами прослойку из пустого объекта таким образом, чтобы изменения в прототипе потомка не влияли на родительский прототип.

Вот как это выглядит

```
function inherit(child, parent) {
   var Temp = function() {
   };
   Temp.prototype = parent.prototype;
   child.prototype = new Temp();
}
```

Теперь применим эту функцию

```
inherit(F2, F);
```

То есть наследуем F2 от F

Запускаем старый код и наблюдаем ошибку, в которой сказано, что у объекта obj нет метода **setAge**.

Убрав вызов методов **setAge** и **getAge** из obj1 - ошибка пропадает.

Отлично! Это то, чего мы и хотели.

Теперь давайте наконец посмотрим как работает функция **inherit**. Напомню, что ее задача - создать между прототипом родителя и прототипом наследника прослойку, чтобы изменения в прототипе наследника не влияли на прототип родителя.

Первым делом мы создаем новый конструктор с именем **Temp**

Затем прототипом этого конструктора устанавливаем прототип родителя.

Затем в прототип наследника записываем результат работы конструктора.

А теперь давайте вспомним что делает конструктор.

Конструктор создает новый пустой объект, а внутри этого объекта создает служебное свойство **__proto__** которое является копией свойства **prototype** конструктора

Следовательно **new Temp** вернет нам пустой объект со служебным свойством **__proto__** в котором будет прототип родителя.

И этот пустой объект станет прототипом для наследника.

Соответственно, сколько бы мы потом не меняли прототип наследника, это никак не повлияет на прототип родителя, ведь по сути, мы добавляем новые методы в пустой объект.

Вызов унаследованных свойств

Как же тогда из объектов типа F2 вызываются методы, унаследованные от F?

А помните, что сначала метод ищется в самом объекте и если не был там найден, ищется в **__proto__**?

Так вот, вызывая **obj2.getName** произойдет вот что.

- сначала интерпретатор попытается найти **getName** внутри объекта **obj2**
- там он его не найдет, после этого, он попытается найти его в свойстве **__proto__** этого объекта, но и там он его не найдет, т.к. там будет два метода для управления возрастом.
- но внутри **__proto__** есть еще одно свойство **__proto__** которое будет указывать уже на прототип конструктора F, в котором и будет метод **getName**

То есть цепочка вызова метода будет вот такая

```
obj2.__proto__.__proto__.getName.call(obj2);
```

Запустим.... как видите, ничего не изменилось, всё прекрасно работает.

Кстати, **call** мы тут использовали потому, что в такой вариации, вызов метода происходит напрямую из прототипа F

А следовательно и контекст у **getName** будет равняться прототипу F, который является просто набором методов и не имеет свойства **name**. Свойство **name** есть только в самом объекте **obj2**. И чтобы переопределить контекст, мы использовали **call**.

Конечно, в здравом уме никто так вызывать функции не будет. Показано это просто для того, чтобы вы четко понимали себе механизм того, как это работает.

В текущей версии JS, всю функцию **inherit** можно сильно укоротить

```
function inherit(child, parent) {
   child.prototype = Object.create(parent.prototype);
}
```

Object.create как раз создает ту самую прослойку на основе переданного прототипа родителя.

Запустим... ничего не изменилось... всё работает как работало.

Теперь предположим, что нам необходимо возможность вызывать одноименные методы родителя... Например мы хотим переопределить поведение метода **setName**.

То есть если вызывать **setName** из объектов типа F он будет вести себя одним образом, а вызывая тот же метод из объектов типа F2, будет вести себя по-другому.

Например, при вызове метода **setName** из F2, мы хотим не просто установить имя, но и вывести уведомление об этом.

Первое, что приходит на ум - просто скопировать код метода **setName** в прототип F2 и немного откорректировать его

```
F2.prototype.setName = function (name) {
   this.name = name;
   console.log('имя [', name, '] установлено');
};
```

Теперь, если запустить код, то при каждой попытке установить имя для объектов типа F2, мы будем видеть уведомление об этом.

При этом тот же самый метод для объектов типа F ведет себя по-старому.

Казалось бы всё отлично, но есть один нюанс - дублирование кода.

Какая нам разница как работает метод **setName** у родителя?

Мы хотим сохранить поведение родителя, но добавить к нему уведомление.

Соответственно, нам нужно как-то вызвать метод родителя и потом вывести уведомление.

Вместо дублированного кода вызываем метод родителя, но контекст заменим на текущий объект

```
F.prototype.setName.call(this, name);
```

Думаю не нужно объяснять что тут произошло... про смену контекста уже было сказано достаточно.

Таким образом нам не важно как именно работает метод родителя. Мы просто вызываем его и затем добавляем свое поведение.

А что, если мы не знаем от кого мы унаследовались? Ведь сейчас у нас жестко прописан F и если мы унаследуем F2 от какого-либо другого конструктора, то F придется менять.

Чтобы придать коду больше универсальности, сначала выясним от какого прототипа мы унаследовались, а потом обратимся к нужному методу этого прототипа

```
this.__proto__.__proto__.setName.call(this, name);
```

__proto__ это прототип текущего объекта.

Прототипом текущего объект, как вы помните из функции **inherit**, является пустой объект, который был сконструирован при помощи конструктора, прототипом которого является прототип родителя.

Но человек прибывающий в здравом уме тоже не будет писать таким образом вызов родительского метода.

Лучше запомнить где-нибудь конструктор родителя и потом обращаться к нему

```
function inherit(child, parent) {
   child.prototype = Object.create(parent.prototype);
   child.parent = parent;
}

F2.parent.prototype.setName.call(this, name);
```

Запустим, видим, что ничего не изменилось.

Есть еще более универсальный вариант, при котором нам даже не обязательно знать имя текущего конструктора... мы можем получить его средствами JS

```
this.constructor.parent.prototype.setName.call(this,
name);
```

Но вот ведь незадача... помните, что свойство constructor каждого объекта хранится не в самом объекте, а в прототипе конструктора объекта?

Так вот, когда мы в функции **inherit** записали в свойство prototype объект-прослойку, мы не создали свойство constructor и теперь по правилам поиска свойств в объекте, о которых было сказано выше, свойство **constructor** будет найдено только в прототипе родителя.

Соответственно у нас не будет доступа к свойству **constructor** конструктора F2

Чтобы предотвратить это, нам необходимо явно указать конструктор унаследованного объекта в функции **inherit**

```
function inherit(child, parent) {
   child.prototype = Object.create(parent.prototype);
   child.prototype.constructor = child;

   child.parent = parent;
}
```

Теперь можем запустить и убедиться, что всё работает и ничего не поломалось.

Но и это еще не всё.... есть еще более простой и читаемый способ получить прототип родителя

```
function inherit(child, parent) {
   child.prototype = Object.create(parent.prototype);
   child.prototype.constructor = child;
   child.prototype.parent = parent;
}

this.parent.prototype.setName.call(this, name);
```

То есть просто записываем конструктор родителя в прототип потомка и всё!

Запустим код и убедимся, что ничего не сломалось

Может возникнуть вопрос: “Зачем нужно было приводить столько разных способов вызова родительских методов, если в итоге мы пришли к простому и лаконичному?”

Это было сделано лишь для того, чтобы вы смогли в деталях понять принцип работы объектов в JS

6

Паттерн “Модуль”

Модуль - это такая конструкция, которая позволяет скрывать детали своей внутренней реализации, а открыть только определенные ее части.

Фактически, модуль - это функция, которая возвращает объект с набором методов и эти методы имеют доступ к “внутренностям” той функции через механизм замыканий. И ТОЛЬКО через эти методы можно получить доступ к этим “внутренностям”. Другими словами: модуль позволяет ограничивать видимость своих данных извне (инкапсуляция)

Классический пример - счетчик:

```
var Counter = function() {
   var counter = 0;

   return {
       inc: function() {
           counter++;
       },
       dec: function() {
           counter--;
       },
       get: function() {
           return counter;
       }
   }
};

var counter1 = Counter();

counter1.inc();
counter1.inc();
counter1.inc();
counter1.dec();

console.log(counter1.get());
```

Объявлена функция **Counter**.

Внутри нее - переменная counter, которой присвоен 0.

Так же, функция **Counter** возвращает объект с тремя методами, для увеличения, уменьшения и получения счетчика.

Все три метода имеют доступ к переменной counter через замыкание.

Возвращая объект с методами, мы, таким образом, скрываем переменную counter от внешнего мира, и доступ к ней имею только методы объекта.

В таких случаях говорят, что переменная counter является приватной, то есть доступной только в пределах **Counter** и получить к ней прямого доступа извне нельзя.

Это и есть паттерн “Модуль”, механизм позволяющий скрывать детали своей реализации.