# Прототипное наследование, свойство \_\_proto\_\_

[Смотреть материал на видео](https://www.youtube.com/watch?v=13e7nw4Pm9c&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

<https://proproprogs.ru/javascript_oop/prototipnoe-nasledovanie>

Ранее у нас были занятия по JavaScript стандарта ES6+ где, помимо всего прочего, мы говорили об объектах. И отмечали, что если нам нужно описать некую сущность: назначить ей разные свойства и поведение (методы), то это можно сделать так:

"use strict";

let geom = {

         name: "фигура",

         sp: {x: 0, y: 0},

         ep: {x: 100, y: 20}

};

Также мы говорили, что для перебора всех свойств и методов объекта используется цикл for in:

**for**(let key **in** geom) {

         console.log(key+": "+geom[key]);

}

А дальше сделаем следующий, новый для нас ход: объявим новый объект, который бы расширял функционал уже существующего geom. Это можно сделать с помощью специального свойства

**\_\_proto\_\_**

которое есть у каждого объекта JavaScript. Именно через него устанавливается связь с базовым (расширяемым) объектом. И в нашем случае это будет выглядеть так:

let rect = {

   draw() {

    console.log("Рисование прямоугольника: " +

**this**.sp.x+","+**this**.sp.y+","+**this**.ep.x+","+**this**.ep.y);

   }

};

rect.\_\_proto\_\_ = geom;

В результате, объект rect получает доступ ко всем свойствам объекта geom и дополнительно объявляет метод draw. Это мы увидим при следующем переборе:

**for**(let key **in** rect) {

         console.log(key+": "+rect[key]);

}

А, вот если закомментировать строку:

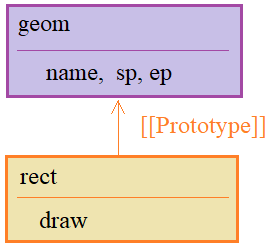
*//rect.\_\_proto\_\_ = geom;*

то останется только один метод draw. Давайте уберем комментарий и выполним метод:

rect.draw()

Увидим в консоли сообщение:

Рисование прямоугольника: 0,0,100,20



Здесь следует отметить, что спецификация языка JavaScript предполагает существование свойства \_\_proto\_\_ только в браузерной среде. В других средах – это необязательное свойство но, тем не менее, оно существует во всех популярных средах, в том числе, и серверных, таких как Node.js.

Возвращаясь к нашим объектам, можно сказать, что в них реализуется прототипный механизм наследования. И эту цепочку можно продолжить, например, добавив вот такой объект:

let info = {

         getInfo() {

                   console.log(**this**.name);

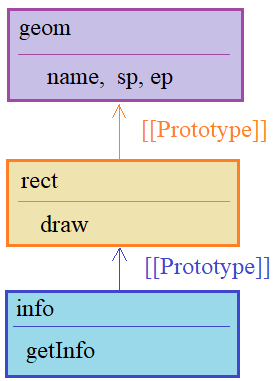
         },

         \_\_proto\_\_: rect

}

info.getInfo()

Теперь у нас info наследуется от rect, а rect – от geom:



Разумеется, когда мы используем свойство \_\_proto\_\_, то ему можно присваивать либо другой объект, либо значение null. Все остальные типы будут попросту игнорироваться. Также из этой схемы видно, что множественное наследование реализовать не получится, т.к. свойство \_\_proto\_\_ ссылается лишь на один определенный объект (либо ни на одного при значении null).

Далее, из объекта, указанного в прототипе можно выполнять только чтение свойств и методов. Например, если в объекте rect создать свойство name:

rect.name = "прямоугольник";

то это никак не скажется на свойстве объекта geom:

console.log(geom.name)          *// фигура*

То есть, алгоритм такой: сначала происходит поиск атрибутов внутри текущего объекта и если они не обнаруживаются, то поиск переходит к объекту, указанному в свойстве \_\_proto\_\_.

**Свойства-аксессоры set и get**

В любом объекте можно создавать специальные свойства, которые будут срабатывать в момент присвоения (set) и чтения (get) данных из объекта. Они определяются по следующему синтаксису:

set <имя метода>([параметры]) {…} – свойство для записи значений в объект;

get <имя метода>([параметры]) {…} – свойство для чтения значений из объекта.

Давайте в качестве примера в объекте geom пропишем два таких свойства:

let geom = {

         name: "фигура",

         sp: {x: 0, y: 0},

         ep: {x: 100, y: 20},

**get** nameGeom() {**return** **this**.name; },

**set** nameGeom(name) {**this**.name = name; },

};

И, далее, можно с ними работать, следующим образом:

console.log(rect.nameGeom);     *// чтение свойства*

rect.nameGeom = "Прямоугольник"; *// запись свойства*

console.log(rect.nameGeom);

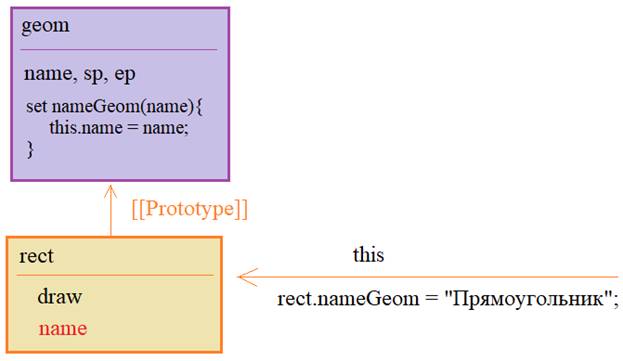
Причем, благодаря наличию свойств-аксессоров, при наследовании объекта в rect не создается новое свойство nameGeom, когда ему присваиваются какие-либо данные, так как JavaScript «понимает», что имеется сеттер с именем nameGeom в базовом объекте и именно его следует использовать в такой ситуации.

**Поведение this при прототипном наследовании**

Следующий момент в этом примере, на который, возможно, кто-то из вас уже обратил внимание: на какой из двух объектов (geom или rect) ссылается this при вызове сеттера:

rect.nameGeom = "Прямоугольник";

Как мы с вами говорили на предыдущих занятиях по JavaScript, указатель this является динамическим и определяется контекстом вызова метода. В данном случае контекст – это объект rect и именно на него ссылается this при вызове сеттера из базового объекта geom. В результате, состояние базового объекта не меняется, а в rect добавляется свойство name:



И в этом легко убедиться, если выполнить следующие строчки:

console.log(rect.nameGeom);

console.log(geom.nameGeom);

Как видите, для объекта geom отображается имя «фигура», тогда как для rect – имя «Прямоугольник». Или, эту проверку можно визуализировать еще лучше, если воспользоваться специальным методом:

obj.hasOwnProperty(key)

который возвращает true, если ключ (key) принадлежит объекту obj и false – в противном случае. В нашем случае можно записать такую конструкцию:

**for**(let prop **in** rect)

**if**( rect.hasOwnProperty(prop) )

                   console.log(prop + ": " + rect[prop]);

На выходе увидим:

draw: function () …  
name: Прямоугольник

То есть, свойство name действительно было создано в дочернем объекте rect.

Но здесь есть один тонкий момент. Если проделать ту же операцию с объектами, а не примитивными типами данных, например с sp и ep:

let geom = {

         name: "фигура",

         sp: {x: 0, y: 0},

         ep: {x: 100, y: 20},

**get** nameGeom() {**return** **this**.name; },

**set** nameGeom(name) {**this**.name = name; },

**get** coords() {

**return** [**this**.sp.x, **this**.sp.y, **this**.ep.x, **this**.ep.y];

         },

**set** coords(coords) {

**this**.sp.x = coords[0]; **this**.sp.y = coords[1];

**this**.ep.x = coords[2]; **this**.ep.y = coords[3];

         }

};

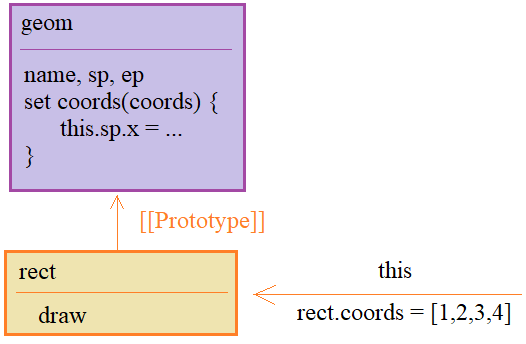
то операция:

rect.coords = [1,2,3,4];

изменит состояние базового объекта geom, оставив дочерний без изменений:

console.log(rect.coords);

console.log(geom.coords);



Дело в том, что мы здесь обращаемся сначала к свойству sp, которое определено в базовом объекте (и мы автоматически переходим к нему), а затем уже, меняем его свойства x и y. То есть, мы не перезаписываем объект целиком, а работаем с его отдельными свойствами. А если переписать, то оно добавится в дочернем объекте:

**set** coords(coords) {

**this**.sp = coords[0]; **this**.sp = coords[1];

**this**.ep = coords[2]; **this**.ep = coords[3];

         }

Вот на это следует обращать внимание.

# Свойство prototype

[Смотреть материал на видео](https://www.youtube.com/watch?v=tKGm1XLLUbo&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

<https://proproprogs.ru/javascript_oop/svoystvo-prototype>

На одном из предыдущих занятий по JavaScript мы с вами говорили о возможности создания объектов с помощью оператора new через специальную функцию-конструктор, например, так:

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

    console.log("Рисование фигуры: "+**this**.name);

  }

}

let r = **new** Rect();

r.draw();

Как здесь указывать базовый объект, который бы расширяла функция Rect? Для этого также можно воспользоваться свойством \_\_proto\_\_, о котором мы начали говорить на предыдущем занятии.

Допустим, выше имеется объявление объекта prop:

let prop = {

  sp: {x: 0, y: 0},

  ep: {x: 100, y: 20},

**get** coords() {

**return** [**this**.sp.x, **this**.sp.y, **this**.ep.x, **this**.ep.y];

  },

**set** coords(coords) {

**this**.sp.x = coords[0]; **this**.sp.y = coords[1];

**this**.ep.x = coords[2]; **this**.ep.y = coords[3];

  }

};

и, далее, в функции добавим строчку:

**//this**.\_\_proto\_\_ = prop;

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

    console.log("Рисование фигуры: "+**this**.name);

  }

**this**.\_\_proto\_\_ = prop;

}

let r = **new** Rect();

r.draw();

Все, мы установили в качестве базового объекта – объект prop. В этом легко убедиться, если перебрать свойства объекта r:

**for**(let p **in** r)

  console.log(p+': '+r[p]);

Как видите, в консоли отобразились атрибуты, связанные и с объектом prop. Однако, в ранних скриптах можно встретить и другую запись:

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

    console.log("Рисование фигуры: "+**this**.name);

  }

}

Rect.**prototype** = prop;

Причем, обращение к свойству prototype происходит вне функции. Это имеет принципиальное значение. Если поместить эту конструкцию внутрь:

**this**.**prototype** = prop;

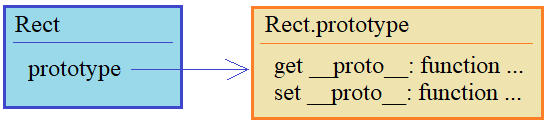
то будет попросту создано свойство с именем **prototype** и значением ссылки на объект prop. Получается, что prototype – это не совсем прототип объекта, подобно другому свойству \_\_proto\_\_, которое можно использовать и после создания объекта. Здесь же важно, чтобы мы определили свойство prototype до вызова оператора new (до создания объекта). Тогда в процессе создания объекта Rect объект, указанный в prototype автоматически станет базовым для Rect. Мало того, в таком качестве свойство prototype используется только в момент вызова оператора new. В последующем, его можно совершенно спокойно видоизменять и это никак не скажется на связке «базовый объект – дочерний объект». Например, после создания объекта r, изменим это свойство:

r.**prototype** = "hello"

и при выводе свойств для r видим в консоли все атрибуты базового объекта и еще свойство prototype со значением hello. То есть, механизм наследования никак не был затронут после создания объекта r и изменения prototype.

## Связь между \_\_proto\_\_ и prototype

Поведение свойств \_\_proto\_\_ и prototype очень похоже, следовательно, между ними должна быть какая-то связь. И, действительно, \_\_proto\_\_ это аксессор объекта prototype:



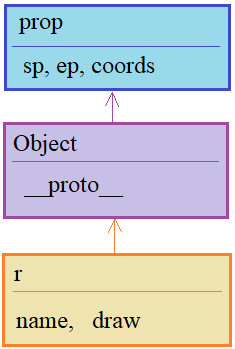
То есть, когда встречается конструкция:

**this**.\_\_proto\_\_ = prop;

то уже после создания объекта устанавливается связь с базовым объектом prop. Или же, перед созданием объекта Rect можно записать строчку:

Rect.**prototype**.\_\_proto\_\_ = prop;

и тогда в момент создания текущему объекту будет назначен базовый prop. Причем, автоматически будет создан некий базовый объект для r, в котором через свойство \_\_proto\_\_ устанавливается связь на объект prop:



## Добавление свойств через prototype

Итак, мы только что увидели, что поведение свойства prototype несколько более изощренное, чем можно было подумать на первый взгляд. Например, зададимся вопросом: что будет, если в этом объекте создать новые свойства:

Rect.**prototype**.\_\_proto\_\_ = prop;

Rect.**prototype**.getName = **function**() { **return** **this**.name; }

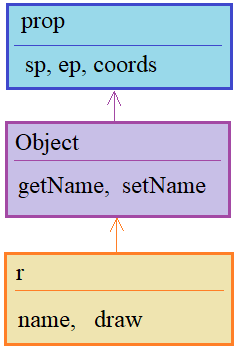
Rect.**prototype**.setName = **function**(name) { **this**.name = name; }

То есть, теперь prototype ссылается не на какой-то конкретный объект, а являясь сам по себе объектом, приобретет дополнительные свойства: prop, getName и setName. В этом случае, новый созданный объект Rect будет иметь доступ ко всем этими свойствам, расположенные в базовом объекте, а базовый объект, в свою очередь, будет ссылаться на объект prop:

let r = **new** Rect();

console.log( r.prop.coords );

console.log( r.getName() );



## Свойство constructor

Каждая функция в JavaScript уже хранит в объекте prototype некоторую служебную информацию. Например, если выполнить строчку:

console.log(Rect.**prototype**)

то мы увидим в этом объекте свойства:

* constructor – ссылка на функцию-конструктор для создания объекта;
* \_\_proto\_\_ – объект, определяющий объект-прототип.

Поэтому, изменяя его:

Rect.**prototype** = prop;

мы уничтожаем эту информацию. В чем легко убедиться, если попытаться вывести в консоль результат следующей проверки:

console.log(r.constructor == Rect)    *//false*

и после комментирования строки:

*// Rect.prototype = prop;*

Увидим совершенно разные результаты: false и true. Чтобы выполнить наследование объектов с использованием prototype и сохранить информацию о конструкторе, можно переопределить это свойство при создании объекта, то есть, внутри функции Rect записать строчку:

//Rect.**prototype**.constructor = Rect;

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

    console.log("Рисование фигуры: "+**this**.name);

  }

  Rect.**prototype**.constructor = Rect;

}

Либо так (как в видео)

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

    console.log("Рисование фигуры: "+**this**.name);

  }

}

Rect.**prototype**.\_\_proto\_\_ = prop;

Тем самым мы восстановим это значение. Но зачем оно вообще нужно? Теряется и пусть теряется. Дело в том, что в больших и сложных программах может потребоваться создать подобный объект. Здесь как раз хорошо подходит это свойство:

let r2 = **new** r.constructor();

В ряде случаев, такой подход очень удобен. И чтобы иметь эту функциональность, мы и сохраняем свойство constructor в prototype.

# Базовые свойства Object, методы create, getPrototypeOf и setPrototypeOf

[Смотреть материал на видео](https://www.youtube.com/watch?v=19Wp9R-QCHY&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

[https://proproprogs.ru/javascript\_oop/svoystva-object-metody-create-getprototypeof-setprototypeof](%20https:/proproprogs.ru/javascript_oop/svoystva-object-metody-create-getprototypeof-setprototypeof)

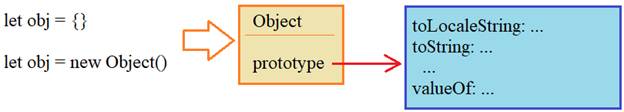
Продолжаем изучение наследования (расширения объектов) на JavaScript. Когда мы только начинали изучение объектов, то отмечали, что их можно создавать двумя способами, так:

let obj = {};

или так:

let obj = **new** Object();

В результате, любой объект JavaScript содержит множество предопределенных свойств и методов, которые описаны в объекте, на который ссылается свойство prototype функции-конструктора Object.



В этом легко убедиться, если вывести это свойство в консоль:

console.log(Object.**prototype**)

Что из этого следует? Как мы с вами отмечали на предыдущем занятии, все эти свойства базового объекта становятся доступными в каждом объекте JavaScript. Например, можно выполнить такую команду:

console.log(obj.toString())

которая не приведет к ошибке, т.к. метод toString объявлен в базовом объекте. А, учитывая, что в JavaScript объектами являются массивы и функции:

Array, Function

то все они могут оперировать методами базового объекта или, переопределять эти методы в своем классе, как это делает объект Array:

let obj = Array(1,2,3);

console.log( obj.toString() )

который выводит в консоль строку со списком значений элементов массива.

## Поведение примитивных типов данных

В JavaScript строки, числа и булевые переменные:

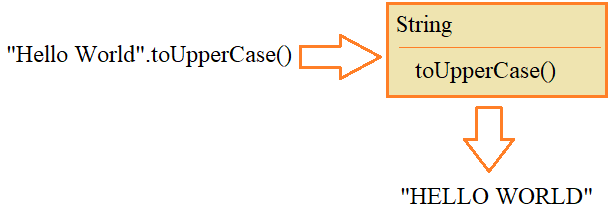
String, Number, Boolean

относятся к примитивным типам и, строго говоря, не являются объектами. Однако, для них также можно вызывать методы базового объекта, например:

console.log("Hello World".toString());

Конечно, эта запись не имеет особого смысла, т.к. она просто возвратит ту же самую строку, но она показывает, что строка, литерал, здесь превращается в объект. Как такое может быть? В действительности, все очень просто. Если виртуальная машина JavaScript «видит», что к литералу идет обращение как к объекту, то она временно создает объект соответствующего типа, для этого объекта вызывает указанный метод и, затем, временный объект уничтожается. В частности, строка обертывается в объект типа String и мы можем для литерала вызвать, например, метод toUpperCase:

console.log("Hello World".toUpperCase());



Раз это так, то для любого такого объекта, в нашем случае строки, можно менять и добавлять новые свойства и методы, используя объект prototype. Как мы отмечали на предыдущем занятии, если добавить в prototype свойство (или переопределить существующее), то новый созданный объект будет его содержать. Например, мы легко можем переопределить метод toUpperCase:

String.**prototype**.toUpperCase = **function**() {

**return** **this**;

}

В результате, он попросту вернет исходную строку. Или, можно добавить новый метод:

String.**prototype**.len = **function**() {

**return** **this**.length;

}

console.log("Hello World".len());

И так далее. Однако, этим функционалом следует пользоваться с большой осмотрительностью, т.к. если в разных программных модулях будут переопределяться одинаковые методы, но с разной логикой работы, то в основной программе работать будет тот, что импортируется в последнюю очередь. Это может привести к непредвиденным ошибкам. В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов.

*Полифил – это эмуляция метода, который существует в спецификации JavaScript, но ещё не поддерживается текущим движком JavaScript.*

Во всех остальных случаях лучше не прибегать к этому механизму.

## Методы create, getPrototypeOf и setPrototypeOf

На сегодняшний день свойство \_\_proto\_\_ считается устаревшим и формально поддерживается только в браузерной среде. Хотя, почти все остальные среды по-прежнему позволяют им пользоваться. Ему на смену пришли новые методы объекта Object:

* Object.create(proto, [descriptors]) – используется для создания нового объекта с указанием базового (proto) и необязательным набором дополнительных дескрипторов свойств – descriptors;
* Object.getPrototypeOf(obj) – возвращает ссылку на базовый объект, либо null, если его нет;
* Object.setPrototypeOf(obj, proto) – назначает базовый объект proto для уже существующего объекта obj.

Например, используя объекты из предыдущего занятия:

let prop = {

         sp: {x: 0, y: 0},

         ep: {x: 100, y: 20},

**get** coords() {

**return** [**this**.sp.x, **this**.sp.y, **this**.ep.x, **this**.ep.y];

         },

**set** coords(coords) {

**this**.sp.x = coords[0]; **this**.sp.y = coords[1];

**this**.ep.x = coords[2]; **this**.ep.y = coords[3];

         }

};

**function** Rect() {

**this**.name = "прямоугольник";

**this**.draw = **function**() {

                   console.log("Рисование фигуры: "+**this**.name);

         }

**this**.\_\_proto\_\_ = prop;

}

Перепишем объявление объекта Rect с использованием метода create:

let rect = Object.create(prop, {

         name: {value: "прямоугольник", writable: **true**},

         draw: {value: **function**() {

                            console.log("Рисование фигуры: "+**this**.name);

                   }

         },

});

console.log( rect.coords );

rect.draw();

Чтобы получить ссылку на базовый класс, можно воспользоваться методом getPrototypeOf:

console.log( Object.getPrototypeOf(rect) === prop );

Наконец, для замены базового объекта на другой, выполним метод setPrototypeOf:

Object.setPrototypeOf( rect, {} );

Мы здесь указали пустой объект вместо прежнего prop и теперь свойство coords возвращает значение undefined.

## Способ клонирования объекта

Часто в скриптах описанные методы используют для клонирования объекта с сохранением ссылки на базовый объект:

let clone = Object.create(Object.getPrototypeOf(rect),

         Object.getOwnPropertyDescriptors(rect));

И если, например, поменять свойство name в клоне:

clone.name = "клон";

то это никак не отразится на объекте rect:

console.log( rect.name );

console.log( clone.name );

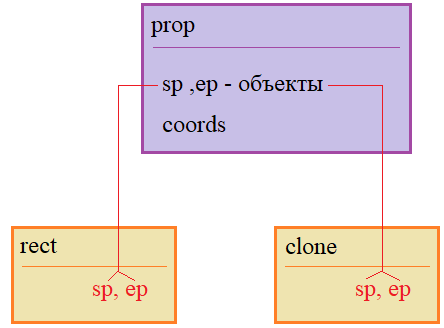
Однако, свойства-объекты ведут себя иначе и при изменении состояния базового объекта:

clone.coords = [1,2,3,4];

console.log( rect.coords );

console.log( clone.coords );

Увидим изменение и у второго объекта rect. И это, в общем-то логично. Как мы с вами отмечали на предыдущем занятии, свойства, которые ссылаются на объекты, а не примитивные типы данных, изменяются непосредственно в базовом классе. А, так как оба дочерних объекта наследуются от одного и того же базового, то и получаем такой результат.



# Классы, методы и свойства, Class Expression

[Смотреть материал на видео](https://www.youtube.com/watch?v=qKEC18S8vVg&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

[https://proproprogs.ru/javascript\_oop/klassy-metody-i-svoystva-class-expression](%20https:/proproprogs.ru/javascript_oop/klassy-metody-i-svoystva-class-expression)

На предыдущих занятиях мы с вами работали с объектами, расширяли их по распространенному синтаксису с использованием свойств:

\_\_proto\_\_ и prototype.

Однако, относительно недавно в JavaScript появилась возможность объявлять классы для создания объектов по общепринятому во многих языках программирования синтаксису – через ключевое слово class:

class <название> {  
   constructor() { … }  
   method1() { … }  
   …  
   methodN() { … }  
}

Например, в самом простом случае, класс для представления информации о книге можно записать так:

class Book {

         constructor() {

                   console.log("создание объекта book");

         }

}

И, затем, с помощью оператора new создать объект по этому шаблону:

let book1 = **new** Book();

В результате увидим строчку «создание объекта book», означающая, что автоматически был вызван метод constructor класса Book. Если кто не знает, это специальный метод, который вызывается при создании объекта. В нем можно выполнять начальную инициализацию каких-либо свойств. Например, мы хотим передавать объекту, при его создании, несколько параметров:

заголовок, автор и цена книги

Для этого в конструкторе определим эти параметры, затем, проинициализируем их, то есть, создадим соответствующие свойства в объекте:

class Book {

         constructor(title, author, price) {

**this**.title = title;

**this**.author = author;

**this**.price = price;

         }

}

И при создании объекта, передадим этим параметрам определенные значения:

let book1 = **new** Book("Муму", "Тургенев", 112);

Выведем в консоль этот объект:

console.log(book1);

и увидим все эти, созданные в конструкторе, свойства. И, разумеется, ссылка this будет указывать на новый созданный объект, а не на класс Book.

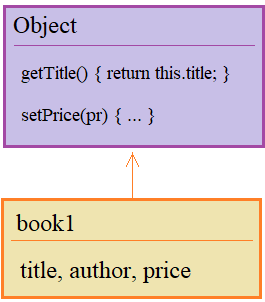
Затем, в этот класс можно поместить несколько методов, например, такие:

         getTitle() { **return** **this**.title; }

         setPrice(pr) { **this**.price = pr; }

Обратите внимание, мы здесь не пишем ключевое слово function, а объявляем методы, используя синтаксис объектов, но без запятых в конце.

И, теперь смотрите, при выводе в консоль объекта book1 мы не увидим в нем этим методов. Они будут находиться в базовом объекте, на который ссылается свойство \_\_proto\_\_:



## Отличия классов от функций

Здесь может показаться, что, в действительности, класс в JavaScript работает наподобие функции-конструктора:

**function** Book(title, author, price) {

**this**.title = title;

**this**.author = author;

**this**.price = price;

}

Book.**prototype**.getTitle = **function**() { **return** **this**.title; }

Book.**prototype**.setPrice = **function**(pr) { **this**.price = pr; }

И вы будете близки к истине. Действительно, если вернуться к классу и отобразить его тип:

console.log(**typeof**(Book));

то увидим значение «function». Но есть и отличия, например, класс создает функцию не в «чистом» виде. Это специальная функция, которая помечена как

[[FunctionKind]]:"classConstructor"

и без оператора new она не может быть вызвана. Например, так:

let book1 = Book("**Муму", "Тургенев**", 112);

Далее, методы класса автоматически помечаются как неперечисляемые и не выводятся с помощью цикла for in:

**for**(let p **in** book1)

         console.log(p+": "+book1[p]);

Наконец, весь код внутри классов автоматически помечается директивой

"use strict"

Так что отличия от обычной функции все же есть.

## Class Expression

Подобно функциям классы можно определять по синтаксису Class Expression в следующем виде:

let Book = class {

  constructor(title, author, price) {

**this**.title = title;

**this**.author = author;

**this**.price = price;

         };

}

Причем, использовать класс можно только после его объявления как бы он ни был объявлен. В отличие от функций, которые можно вызывать в скрипте до их определения, если они заданы по синтаксису Function Declaration.

Также, подобно функциям, классам можно назначать имя при следующем определении:

let Book = class BookClass {

  constructor(title, author, price) {

**this**.title = title;

**this**.author = author;

**this**.price = price;

  };

}

Причем, это второе имя будет видно только внутри этого класса. За его пределами можно обращаться только через переменную Book.

Интересной особенностью является возможность создавать классы динамически с помощью функции, например, так:

**function** createFruit(name, weight) {

**return** class {

    constructor() {

**this**.name = name;

**this**.weight = weight;

  };

  showInfo() { console.log(**this**.name + " " + **this**.weight) };

  }

}

Здесь описана функция, которая формирует новый класс со значениями параметров name и weight, которые будут записаны в объект при его создании. Далее, мы можем создать класс для определенного фрукта:

let f1 = createFruit("Яблоко", 100);

Причем, здесь f1 – это ссылка на класс, а не на созданный объект, то есть, далее, нужно вызвать оператор new и создать объект:

let apple = **new** f1();

apple.showInfo();

Если вызвать функцию еще раз с другими параметрами:

let f2 = createFruit("Вишня", 40);

то будет создан еще один класс, никак не связанный с предыдущим и, далее, также можно выполнить следующие конструкции:

let v = **new** f2();

v.showInfo();

То есть, все классы работают совершенно независимо.

## Геттеры и сеттеры классов

Часто в классах прописывают специальные методы:

* геттеры – для получения свойств объекта;
* сеттеры – для изменения свойств объекта.

В рамках JavaScript такие методы можно объявлять с помощью специальных ключевых слов:

get и set

следующим образом:

**get** titleBook() {**return** **this**.title; }

**set** titleBook(value) { **this**.title = value; }

И, далее, использовать, просто обращаясь к ним по имени:

let book1 = **new** Book("Муму", "Тургенев", 112);

console.log( book1.titleBook );

book1.titleBook = "Мы болеем за Муму";

console.log( book1.titleBook );

Это бывает очень удобно.

## Свойства классов

Совсем недавно была добавлена возможность описания свойств непосредственно внутри класса, например, так:

class Book {

  pages = 123;

  constructor(title, author, price) {

**this**.title = title;

**this**.author = author;

**this**.price = price;

  };

}

Причем, они автоматически становятся частью самого объекта book1, а не базового класса. Мы в этом можем легко убедиться, если выведем объект в консоль:

console.log(book1);

Вот что из себя представляют классы в базовом представлении.

# Наследование классов, функция super

[Смотреть материал на видео](https://www.youtube.com/watch?v=ZHMQ2NJIUW0&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

<https://proproprogs.ru/javascript_oop/nasledovanie-klassov-funkciya-super>

На предыдущем занятии мы с вами познакомились с классами JavaScript. А раз есть классы, то, конечно же, должно быть и наследование классов. Эта фундаментальная концепция ООП не обошла язык JavaScript стороной и сейчас мы посмотрим как происходит расширение функционала базового класса при помощи механизма наследования.

Для тех, кто только к нам присоединился и задумывается: а что такое вообще наследование и зачем оно нужно? В двух словах поясню. Сейчас мы будем рассматривать пример, в котором создадим некий начальный (базовый) класс для описания свойств геометрических фигур. И назовем его prop:

class Prop {

         constructor(width, color) {

**this**.width = width;

**this**.color = color;

         }

}

Затем, создадим связь между этим классом и новым (дочерним) для представления линии на плоскости. Пусть класс называется Line и без связки выглядит так:

class Line {

  constructor(sp, ep, width, color) {

**this**.sp = sp;

**this**.ep = ep;

**this**.width = width;

**this**.color = color;

  }

  draw() {

    console.log("Линия: "+**this**.sp.x+", "+**this**.sp.y+

    ", "+**this**.ep.x+", "+**this**.ep.y);

  }

}

И вот здесь мы бы хотели, чтобы свойства width и color формировались через базовый класс Prop и, вообще, весь функционал с ними связанный, находился бы в этом классе. Для этого, как раз, и используется механизм наследования, который позволяет расширять функционал базового класса за счет дочернего. Синтаксически, базовый класс указывается с помощью ключевого слова

extends <имя базового класса>

например, так:

class Line extends Prop { … }

и в конструкторе уберем формирование свойств width и color:

         constructor(sp, ep, width, color) {

**this**.sp = sp;

**this**.ep = ep;

         }

Если сейчас попробовать сформировать объект Line:

let l1 = **new** Line({x: 0, y: 0}, {x: 10, y: 20}, 1, 'red');

то произойдет ошибка. Ошибка, связанная с особенностью работы конструктора в наследуемом классе Line. Дело в том, что JavaScript формирует указатель this только после выполнения конструктора базового класса. Но мы здесь нигде его не вызываем и автоматически он тоже не вызывается. Из-за этого и происходит данная ошибка. Поправим это. Для вызова конструктора базового класса существует специальная функция:

super([параметры])

которой мы и воспользуемся:

class Line extends Prop {

         constructor(sp, ep, width, color) {

                   super(width, color);

**this**.sp = sp;

**this**.ep = ep;

         }

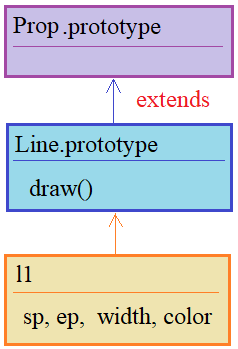
…

Причем, обратите внимание, функция super в конструкторе должна вызываться до использования объекта this, так как он формируется только после ее вызова.

Если теперь обновить страницу браузера, то никаких ошибок не возникнет. В результате, будет создан объект l1 с указанными свойствами и методом draw:

console.log( l1 );

Но, какая структура наследования у нас в итоге получилась? Как мы помним, в JavaScript используется прототипное наследование, а классы – это практически синтаксический сахар над всем этим механизмом. И, действительно, результат может показаться, на первый взгляд, несколько необычным:



Смотрите, сначала формируется объект l1 с базовым Line.prototype, а затем, уже идет объект Prop.prototype. Кроме того, все свойства формируются в дочернем объекте l1, так как именно на него указывает this. Поэтому, когда в базовом классе Prop выполняется конструктор:

         constructor(width, color) {

**this**.width = width;

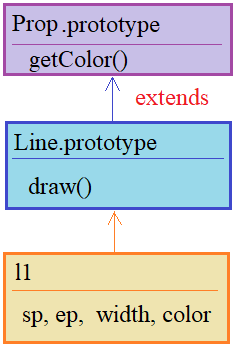
**this**.color = color;

         }

то width и color записываются в дочерний объект l1. Если добавить в базовый класс метод, например:

         getColor() { **return** **this**.color; }

то он будет храниться именно в нем:



## Переопределение методов

При необходимости, методы базовых классов можно переопределять в дочерних. Например, переопределим метод getColor в классе Line:

         getColor() {

**return** '['+**this**.color+']';

         }

И, если теперь его вызвать:

console.log( l1.getColor() );

то в консоли увидим строчку:

[red]

Если же закомментировать этот метод в дочернем классе и повторить вызов, то увидим просто слово red. То есть, вот так, довольно просто можно переопределять методы базового класса. Однако, часто нужно сохранить функциональность метода из базового класса и расширить ее в дочернем. Для этого можно вызвать переопределяемый метод из базового класса, используя специальный объект super:

         getColor() {

                   let color = super.getColor();

**return** '['+color+']';

         }

Смотрите, мы здесь сначала берем значение цвета через геттер базового класса, а, затем, добавляем квадратные скобки. Это более гибкий подход, т.к. цвет может возвращаться и в виде строки и в виде числа или в еще каком-либо другом виде. И мы, благодаря обращению к методу базового класса, сохраняем эту функциональность.

Однако, обратите внимание, контекст super на базовый класс может легко потеряться, если мы обратимся к нему, например, из анонимной функции:

         showColor() {

                   setTimeout(**function**() {

                            console.log( super.getColor() );

                   }, 0);

         }

Произойдет ошибка. А вот так, все будет работать:

         showColor() {

                   console.log( super.getColor() );

         }

или, так:

         showColor() {

                   setTimeout(() => {

                            console.log( super.getColor() );

                   }, 0);

         }

Мы здесь используем стрелочную функцию, а как известно, они «прозрачные» и не образуют собственного контекста.

## Конструктор по умолчанию

В объявляемых классах не обязательно прописывать конструктор. Например, если их убрать, то ошибок при создании объекта не будет:

class Prop {

         getColor() { **return** **this**.color; }

}

class Line extends Prop {

         draw() {

                   console.log("Линия: "+**this**.sp.x+", "+**this**.sp.y+

                                     ", "+**this**.ep.x+", "+**this**.ep.y);

         }

}

let l1 = **new** Line({x: 0, y: 0}, {x: 10, y: 20}, 1, 'red');

При этом будет создана полная иерархия объектов. Правда сам объект l1 будет пустым:

console.log( l1 );

Но как это все работает без конструкторов? В действительности, в дочерних классах они всегда есть и по умолчанию, если мы не определяем своих, принимают вид:

constructor(...args) {  
    super(...args);  
}

То есть, дочерние классы по умолчанию вызывают конструкторы базовых, передавая им все принятые аргументы args. В этом легко убедиться, если в базовом классе Prop вернуть конструктор для инициализации двух параметров:

class Prop {

         constructor(width, color) {

**this**.width = width;

**this**.color = color;

         }

         getColor() { **return** **this**.color; }

}

Выполняя программу, в консоли увидим объект с этими двумя свойствами. Это как раз произошло, благодаря вызову конструктора по умолчанию в дочернем классе Line.

## Особенности работы extends

Последнее, о чем мы поговорим на этом занятии, это возможность динамической генерации базового класса. Дело в том, что после ключевого слова extends можно указывать любой исполняемое выражение:

extends <исполняемое выражение>

например, функцию, которая бы возвращала класс. Давайте сначала объявим такую функцию:

**function** getExtends(type) {

**return** class {

    constructor(width, color) {

**this**.type = type;

**this**.width = width;

**this**.color = color;

    }

  };

}

И, затем, вызовем ее после extends:

class Line extends getExtends("2D") {

  constructor(sp, ep, width, color) {

    super(width, color);

**this**.sp = sp;

**this**.ep = ep;

  }

  draw() {

    console.log("Линия: "+**this**.sp.x+", "+**this**.sp.y+

    ", "+**this**.ep.x+", "+**this**.ep.y);

  }

  getType() { **return** **this**.type; }

}

Возможно, это несколько необычно выглядит. Зато, бывает весьма эффективно. В частности, здесь, мы указываем тип фигуры «2D» - двумерная, который записывается в свойстве type создаваемого объекта. Причем, далее, по программе нам нет необходимости определять это свойство. Оно автоматически будет создаваться с этим значением, т.к. мы сгенерировали именно такой класс: значение type: 2D в него, как бы, «вшито».

# Статические методы и свойства классов

[Смотреть материал на видео](https://www.youtube.com/watch?v=OR0Xug9StOg&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

<https://proproprogs.ru/javascript_oop/staticheskie-metody-i-svoystva-klassov>

На этом занятии мы с вами поговорим о статических методах и свойствах. Что это такое? Давайте предположим, что имеется класс для описания пользователя. Он будет очень простым: хранить имя, возраст и иметь метод, возвращающий имя человека:

class Users {

         constructor(name, old) {

**this**.name = name;

**this**.old = old;

         }

         getName() { **return** **this**.name; }

}

Далее, можно создавать объекты этого класса, следующим образом:

let u1 = **new** Users("Михаил", 19);

let u2 = **new** Users("Федор", 19);

Теперь, приходит заказчик и говорит, что надо бы добавить функционал для сравнения пользователей, например, по возрасту. И мы начинаем думать: как это лучше сделать? Из того, что нам известно на этот момент, было бы логично добавить метод для такого сравнения, например, так:

         compareOld(user2) {

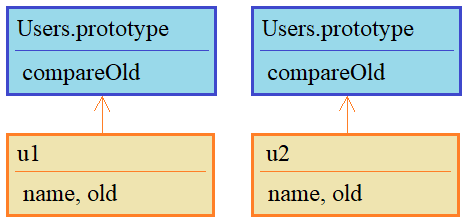
**return** **this**.old == user2.old;

         }

И, затем, использовать его:

console.log( u1.compareOld(u2) );

Отлично, все работает. Мы потираем руки и радуемся легкости исполнения задания. Но, тут приходит опытный программист, смотрит на весь этот код, и начинает изрыгать нелитературные выражения в наш адрес! Что ему не понравилось? Дело вот в чем. Правильнее было бы реализовать функционал для сравнения пользователей на более обобщенном уровне, не привязывая его к объектам каждого пользователя. Потому что эта привязка приведет к добавлению метода compareOld в каждый объект Users, что приведет к неэффективному использованию памяти:



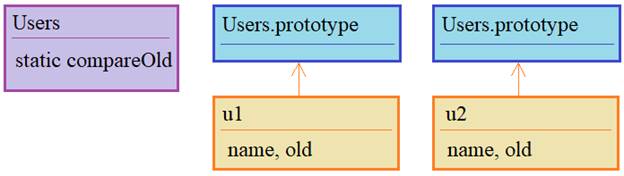
Чтобы это поправить, как раз и можно воспользоваться статическими методами. Если записать метод compareOld как статический (перед его именем ставится ключевое слово static):

**static** compareOld(user1, user2) {

**return** user1.old == user2.old;

         }

то он будет принадлежать только классу, но не объектам:



Соответственно, из объектов его вызвать уже нельзя и вот такая строчка приведет к ошибке:

console.log( u1.compareOld(u1, u2) );

Вызвать его можно только непосредственно из класса Users:

console.log( Users.compareOld(u1, u2) );

Это и означает, что статические методы принадлежат только классам, но не объектам и доступ к таким методам возможен только из класса. В результате, мы получаем некую общую реализацию для сравнения возрастов двух произвольных пользователей. И в контексте нашей задачи, такое решение будет лучшим.

Фактически, слово static «указывает» создавать метод по синтаксису:

Users.compareOld = **function**(user1, user2) {

**return** user1.old == user2.old;

}

Тогда как обычные методы объявляются как свойства объекта prototype:

Users.**prototype**.getName = **function**() { **return** **this**.name; }

## Наследование статических методов

Но как будут вести себя статические методы при наследовании классов? Предположим, что мы хотим добавить еще одного специализированного пользователя Admin:

class Admin extends Users {

         constructor(name, old, login, psw) {

                   super(name, old);

**this**.login = login;

**this**.psw = psw;

         }

}

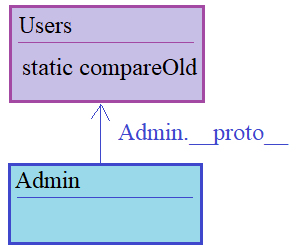
Мы расширяем базовый класс Users и добавляем еще два свойства: login и psw. Будет ли статический метод compareOld доступен в дочернем классе Admin? Да, будет и, далее, мы можем создать такого пользователя:

let u2 = **new** Admin("Федор", 19, "aaa", "0123");

и сравнить их, вызывая статический метод через класс Admin:

console.log( Admin.compareOld(u1, u2) );

То есть, метод compareOld можно вызывать и через класс Users и через класс Admin. Разницы никакой не будет. Это происходит по той причине, что свойство \_\_proto\_\_ класса Admin ссылается на класс Users:



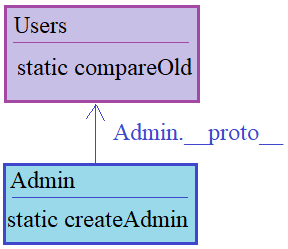
Если же добавить еще один статический метод, но уже в класс Admin:

**static** createAdmin(name, old) {

**return** **new** **this**(name, old, "admin", "root");

         }

то картина будет такой:



В методе createAdmin мы создаем нового пользователя Admin с использованием только двух параметров: name, old. Остальные два задаются по умолчанию как: "admin", "root". Причем, ключевое слово this здесь будет ссылаться на класс, указанный перед точкой, при вызове данного метода. Например:

let u3 = Admin.createAdmin("Сергей", 33);

Здесь this ссылается на Admin, поэтому будет создан новый объект класса Admin. А вот если мы в методе compareOld добавим вывод:

console.log(**this** == Admin);

и вызовем его двумя способами:

Users.compareOld(u1, u2);  *// false*

Admin.compareOld(u1, u2);  *// true*

то в первом случае this будет ссылаться на Users, а во втором – на Admin.

## Статические свойства

Недавно в JavaScript была добавлена возможность объявления статических свойств внутри класса, например, так:

class Users {

**static** countUsers = 0;

…

}

Это свойство будет принадлежать только классу Users и не копироваться в его объекты. Например, так можно сделать подсчет числа созданных пользователей в программе. Добавим в конструктор строчку:

Users.countUsers++;

И, затем, выведем это свойство в консоль:

console.log( Users.countUsers );

Увидим число созданных пользователей.

# Приватные методы и свойства, оператор instanceof

[Смотреть материал на видео](https://www.youtube.com/watch?v=FHGyblLVbaE&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

[https://proproprogs.ru/javascript\_oop/privatnye-metody-i-svoystva-operator-instanceof](%20https:/proproprogs.ru/javascript_oop/privatnye-metody-i-svoystva-operator-instanceof)

Согласно концепции инкапсуляции ООП в классах можно создавать как открытые (публичные) методы и свойства, так и закрытые, доступные только внутри класса или его объектов. До сих пор все методы и свойства у нас были публичные. Но, что если мы хотим некоторые из них указать как приватные, закрытые? Например, в классе Users такими сделать свойства name и old:

class Users {

         constructor(name, old) {

**this**.name = name;

**this**.old = old;

         }

         getName() { **return** **this**.name; }

}

Для этого программисты между собой давно договорились: если свойство или метод начинаются с символа подчеркивания, то это приватное свойство и к нему не следует обращаться из вне:

* \_name, \_old – приватные свойства;
* \_setProperty(key, value) – приватный метод.

И это именно договоренность. На уровне языка JavaScript никакого отличия между такими и другими свойствами и методами нет. Например, отмечая свойства name и old как приватные:

         constructor(name, old) {

**this**.\_name = name;

**this**.\_old = old;

         }

Мы к ним, тем не менее, можем совершенно спокойно обращаться из вне:

console.log( u1.\_name );

Но это уже остается на совести программиста и, вообще, считается порочной практикой, т.к. такие свойства в будущем (в другой версии модуля) могут измениться или совсем исчезнуть и тогда придется переделывать и саму программу, что не очень приятно. Поэтому, отмеченные таким образом атрибуты, следует использовать исключительно внутри класса. Таково общепринятое соглашение.

Разумеется, как обычные свойства, условно-приватные также наследуются и доступны в дочерних классах. Например, объявим метод getName как приватный:

\_getName() { **return** **this**.\_name; }

и создадим дочерний класс Admin:

class Admin extends Users {

         constructor(name, old, login, psw) {

                   super(name, old);

**this**.login = login;

**this**.psw = psw;

         }

}

Тогда после создания объекта:

let u2 = **new** Admin("Кирилл", 22, "admin", "1111");

мы совершенно спокойно можем вызвать метод базового класса:

console.log( u2.\_getName() );

даже, если он по соглашению приватный. Это еще раз показывает, что закрытые методы – это лишь соглашение между программистами, языком JavaScript они воспринимаются как обычные методы и свойства.

## Настоящие приватные свойства

Совсем недавно в JavaScript появилась возможность объявлять «настоящие» приватные свойства, которые становятся защищенными на уровне языка, а не просто соглашения между программистами. Для их обозначения перед именами ставится символ шарп:

#<имя свойства> - защищенное свойство.

Например, в классе Users для объявления таких свойств мы должны, во-первых, указать их непосредственно в классе:

class Users {

         #name;

         #old;

         constructor(name, old) {

**this**.#name = name;

**this**.#old = old;

         }

         getName() { **return** **this**.#name; }

}

А, затем, уже можем использовать в экземплярах классов:

let u1 = **new** Users("Михаил", 19);

let u2 = **new** Admin("Кирилл", 22, "admin", "1111");

console.log( u1.getName() ); *// Михаил*

console.log( u2.getName() ); *// Кирилл*

Как видите, в консоли выводятся разные имена, следовательно, для каждого объекта приватное свойство является уникальным и хранится непосредственно в нем.

Если же мы попытаемся обратиться к такому свойству из вне:

console.log( u1.#name );

то возникнет ошибка. Это как раз и показывает, что такие свойства по-настоящему приватные и могут изменяться только внутри класса или его объектах.

У таких свойств есть один нюанс: обращаться к ним по синтаксису:

this[<ключ>]

нельзя. Например, если в методе getName записать:

getName() { **return** **this**['#name']; }

то получим значение undefined. Тогда, как с обычными свойствами все работало бы без проблем.

Вот так в JavaScript можно объявлять и работать с приватными свойствами и методами.

## Оператор instanceof

В JavaScript есть довольно полезный и часто используемый оператор

obj instanceof Class

который проверяет принадлежность объекта obj классу Class, то есть, возвращает true, если obj действительно является экземпляром класса Class (или, любому базовому классу) и false – в противном случае.

Например, в нашей программе можно реализовать следующую проверку:

console.log( u1 **instanceof** Admin);   *// false*

console.log( u2 **instanceof** Admin );   *// true*

А, вот если вместо Admin написать Users:

console.log( u1 **instanceof** Users);

console.log( u2 **instanceof** Users);

то в обоих случаях получим true. Это, как раз, и показывает, что оператор instanceof выполняет проверку принадлежности с учетом наследования классов. Это отличает данный оператор от простого сравнения:

console.log( u1.constructor == Users);

console.log( u2.constructor == Users);

Здесь свойство constructor (если оно есть и не было искусственно изменено) хранит ссылку на функцию-конструктор, с помощью которой и был создан текущий объект. А, так как класс – это и есть функция-конструктор, то мы, таким образом, проверяем соответствие объекта тому или иному классу.

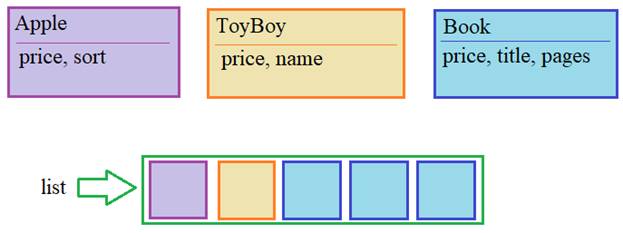
Вот так в JavaScript можно определять тип объекта с помощью оператор instanceof и простой проверкой на равенство.

# Примеси - mixins

[Смотреть материал на видео](https://www.youtube.com/watch?v=FvbzdR3KlJk&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

<https://proproprogs.ru/javascript_oop/primesi-mixins>

В ООП есть один распространенный и полезный прием, позволяющий добавлять в классы наборы методов по умолчанию. О чем здесь речь? Давайте представим, что реализуется магазин, в котором товары могут быть представлены разными классами, никак между собой не связанными:



Затем, создаются объекты на основе этих классов и помещаются в массив list. И, далее, разработчики между собой договариваются: пусть каждый класс содержит следующий предопределенный набор методов:

* getPrice – для получения цены товара;
* getWeight – для получения веса товара;
* getSize – для получения размера товара;
* getPages – для получения числа страниц в товаре (например, для книг).

Зачем это нужно? Ну, например, далее в модуле может быть выполнена сортировка товара по одному из этих признаков, допустим по цене:

list.sort( (a, b) => a.getPrice() - b.getPrice() );

или по весу:

list.sort( (a, b) => a.getWeight() - b.getWeight() );

или по числу страниц:

list.sort( (a, b) => a.getPages() - b.getPages() );

и так далее. То есть, все эти методы нужны для реализации последующего функционала. Причем, их число может в будущем меняться: один добавляться, другие убираться. Как в этих условиях лучше всего построить программу?

Конечно, первое что приходит на ум, это взять и прописать все эти методы непосредственно в этих классах, т.к. единого базового у них нет. В результате, получим что-то вроде такого:

class Apple {

         constructor(price, sort) {

**this**.price = price;

**this**.sort = sort;

         }

         getPrice() { **return** **this**.price; }

         getWeight() { **return** 0; },

         getSize() { **return** 0; },

         getPages() { **return** 0; }

}

class ToyBoy {

         constructor(price, name) {

**this**.price = price;

**this**.name = name;

         }

         getPrice() { **return** **this**.price; }

         getWeight() { **return** 0; },

         getSize() { **return** 0; },

         getPages() { **return** 0; }

}

class Book {

         constructor(price, title, pages) {

**this**.price = price;

**this**.title = title;

**this**.pages = pages;

         }

         getPrice() { **return** **this**.price; }

         getPages() { **return** **this**.pages; }

         getWeight() { **return** 0; },

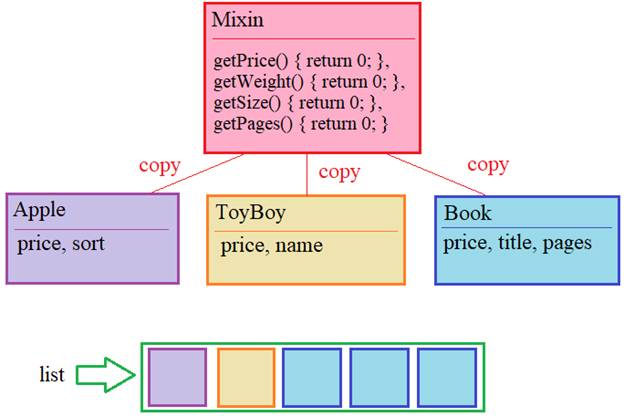
         getSize() { **return** 0; },

}

Как видите, мы вынуждены прописывать все эти методы в каждом классе, причем, некоторые из методов определены чисто формально, возвращающие 0. Теперь представьте, что таких классов десятка два и в них постоянно приходится вносить изменения по этим методам. Кажется, перспективы не очень радужные? Но как мы можем здесь облегчить свое положение? Как раз для этого очень хорошо подходит концепция примесей, или, по-английски

Mixins

в которой предлагается создать единый объект (или класс) с методами по умолчанию. Причем, этот объект не будет участвовать в наследовании, а просто скопирует свое содержимое в соответствующие классы:



А, затем, мы просто переопределим нужные из них. Объявление примеси можно определить следующим образом:

let ShopMixin = {

         getPrice() { **return** 0; },

         getWeight() { **return** 0; },

         getSize() { **return** 0; },

         getPages() { **return** 0; }

};

А классы определить так:

class Apple {

         constructor(price, sort) {

**this**.price = price;

**this**.sort = sort;

         }

}

Object.assign(Apple.**prototype**, ShopMixin);

Apple.**prototype**.getPrice = **function**() { **return** **this**.price; }

class ToyBoy {

         constructor(price, name) {

**this**.price = price;

**this**.name = name;

         }

}

Object.assign(ToyBoy.**prototype**, ShopMixin);

ToyBoy.**prototype**.getPrice = **function**() { **return** **this**.price; }

class Book {

         constructor(price, title, pages) {

**this**.price = price;

**this**.title = title;

**this**.pages = pages;

         }

}

Object.assign(Book.**prototype**, ShopMixin);

Book.**prototype**.getPrice = **function**() { **return** **this**.price; }

Book.**prototype**.getPages = **function**() { **return** **this**.pages; }

Здесь используется метод assign объекта Object, который и копирует свойства из ShopMixin в соответствующие объекты prototype. Таким образом, каждый создаваемый объект будет иметь доступ к методам по умолчанию.

Как видите, такая реализация выглядит более приемлемой и удобной, причем, нигде не используется механизм наследования, что в ряде случаев как раз и является большим преимуществом.

Далее, можно создать несколько объектов:

list.push(**new** Apple(100, "Белый налив"));

list.push(**new** ToyBoy(200, "Кукла"));

list.push(**new** Book(20, "Муму", 100));

list.push(**new** Book(40, "Евгений Онегин", 90));

list.push(**new** Book(30, "Мастер и Маргарита", 500));

И выполнять различные сортировки:

list.sort( (a, b) => a.getPrice() - b.getPrice() );

*// list.sort( (a, b) => a.getWeight() - b.getWeight() );*

*// list.sort( (a, b) => a.getPages() - b.getPages() );*

**for**(let s of list)

         console.log( s.price );

Объекты с не переопределенными методами по умолчанию встанут в начало списка, остальные будут отсортированы. Например, сортировка по числу страниц, формально, применима только к книгам, но у нас не будет возникать ошибок, если в списке присутствуют и другие типы объектов и все благодаря наличию методов по умолчанию. Вот общий принцип примесей и вот так их можно реализовать в JavaScript.

# Блоки try/catch/finally, оператор throw, проброс исключений

[Смотреть материал на видео](https://www.youtube.com/watch?v=gpwmjwlywrs&list=PLA0M1Bcd0w8xXjtQlT1vf8xgJKtSh8j4w" \t "_blank)

[https://proproprogs.ru/javascript\_oop/bloki-trycatchfinally-operator-throw-probros-isklyucheniy](%20https:/proproprogs.ru/javascript_oop/bloki-trycatchfinally-operator-throw-probros-isklyucheniy)

В практически любой язык программирования, в том числе и JavaScript, имеет специальную конструкцию:

try/catch/finally

позволяющую «ловить» ошибки во время выполнения программы и предпринимать определенные действия (обрабатывать эти ошибки). Общая идея работы этого блока заключается в том, что мы помещаем в блок try потенциально уязвимый скрипт:

let res = 0;

**try** {

         res = 5/d;

         console.log(res);

}

**catch**(error) {

         console.log(error);

}

И, если в нем что-то пошло не так, то выполнение программы прерывается и управление передается блоку catch, в котором можно реализовать определенные действия по обработке ошибки, например, вывести сообщение в консоль.

Причем, обратите внимание, если бы в нашем примере отсутствовал блок try, то скрипт полностью прекратил бы свою работу и его выполнение было бы остановлено браузером. Но, благодаря наличию этого блока, прерывается только та часть программы, которая находится в try и, далее, мы уже попадаем в блок catch, чтобы выполнить определенные действия при нештатной ситуации. И это значительно повышает надежность работы скрипта.

Если же в блоке try ошибок не возникло, то блок catch пропускается и не выполняется:

let res = 0;

**try** {

         let d = 2;

         res = 5/d;

         console.log(res);

}

**catch**(error) {

         console.log(error);

}

Но, обратите внимание, блок try выполняется в синхронном режиме, то есть, например, если в него поместить вот такой асинхронный вызов функции:

let res = 0;

**try** {

         setTimeout(**function**() {

                   res = 5/d;

                   console.log(res);

         }, 0);

}

**catch**(error) {

         console.log(error);

}

То ошибка обработана не будет, т.к. анонимная функция внутри setTimeout выполняется уже после выполнения блока try. То есть, блок try сначала полностью выполнится, а затем, будет запущена анонимная функция. В такой ситуации блок try/catch следует поместить внутрь анонимной функции:

let res = 0;

setTimeout(**function**() {

**try** {

    res = 5/d;

    console.log(res);

  }

**catch**(error) {

    console.log(error);

  }

}, 0);

## Свойства объекта ошибки

В языке JavaScript объект ошибки блока catch имеет три полезных атрибута:

* name – содержит название ошибки;
* message – содержит текст ошибки;
* stack – стек вызова до возникновения ошибки (нестандартное свойство).

Например, мы можем вывести эти атрибуты в консоль, следующим образом:

let res = 0;

**try** {

  d = 2;

  res = 5/d;

  console.log(res);

}

**catch**(error) {

  console.log(error.name);

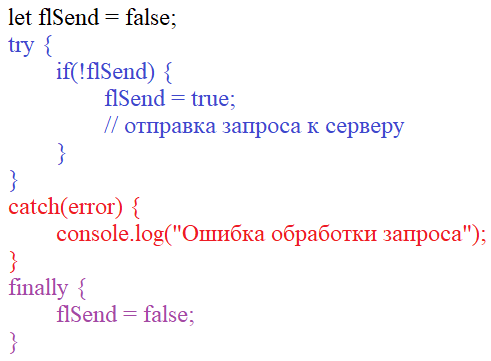
  console.log(error.message);

  console.log(error.stack);

}

## Блок finally

При обработке ошибок часто используется еще один необязательный блок finally, который выполняется в любом случае: возникли ошибки или нет, причем, в последнюю очередь. Зачем он может понадобиться? Например, чтобы выполнить некоторые обязательные действия после завершения соединения с сервером:



Мы здесь определяем некоторый флаг (flSend) для определения выполнения текущего запроса и если он установлен в true, то запрос второй раз отправляться не будет. Соответственно, потом, после успешной или ошибочной обработки, этот флаг обязательно нужно вернуть в значение false. Как раз это и делается в блоке finally.

## Генерация собственных исключений – throw

JavaScript позволяет генерировать свои собственные исключения с помощью оператора

throw <объект ошибки>

В качестве объекта ошибки может быть и число и строка или еще какой-либо примитив и объект, но лучше использовать стандартные классы ошибок, например:

* Error – для общих ошибок;
* SyntaxError – для синтаксических ошибок;
* TypeError – для ошибок типов данных;
* ReferenceError – ссылка на несуществующую переменную

и так далее. Чтобы воспользоваться этими классами, достаточно создать соответствующий объект:

let err1 = **new** Error("Ошибка выполнения");

let err2 = **new** SyntaxError("Ошибка синтаксиса");

let err3 = **new** TypeError("Ошибка типов данных");

и указать после оператора throw:

**throw** err1;

**throw** **new** SyntaxError("Ошибка синтаксиса");

Например, зададим функцию деления двух чисел, которая генерирует исключение, если знаменатель равен нулю:

**function** divide(a, b) {

**if**( b == 0 ) {

**throw** **new** Error("Деление на ноль");

         }

**return** a/b;

}

И, далее, вызываем ее в блоке try:

let res = 0;

**try** {

         res = divide(1, 0);

         console.log(res);

}

**catch**(error) {

         console.log(error.name);

         console.log(error.message);

}

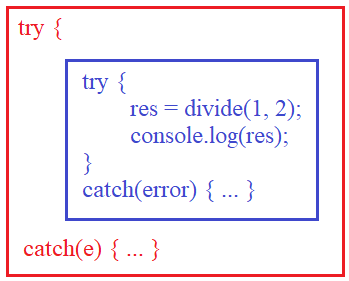
В консоли увидим строчки:

Error, Деление на ноль

То есть, свойство name ссылается на имя конструктора объекта ошибки, а message – на аргумент этого объекта.

## Проброс исключений

В нашей последней реализации блок catch отлавливает все исключения в блоке try. Но, что если мы хотим отслеживать лишь определенный их тип, например, Error, а остальные не трогать – пересылать дальше – блокам try более высокого уровня:



Это и называется пробросом исключения. Для его реализации нам достаточно проверить во внутреннем блоке catch тип ошибки и, если она не соответствует Error, то передать дальше:

*// let res = 0;*

**try** {

         res = divide(1, 2);

         console.log(res);

}

**catch**(error) {

**if**(error.name == "Error") {

                   console.log(error.name);

                   console.log(error.message);

         }

**else** {

**throw** error;

         }

}

Соответственно, внутри будет обрабатывать только исключения Error, а остальные пробрасывать блокам более высокого уровня.