Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.
Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу http://learn.javascript.ru.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Продвинутая работа с объектами

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Прототип: наследование и методы

В этой главе мы рассмотрим, как работает «прототипное наследование».

Его смысл — в том, что один объект можно сделать *прототипом* другого. При этом если свойство не найдено в объекте — оно берётся из прототипа. В JavaScript оно реализовано на уровне языка.

Наследование через ссылку `__proto__`

Наследование в JavaScript реализуется при помощи специального свойства `__proto__`.

Оно явно доступно в браузерах Chrome/Firefox и Safari, а в остальных — спрятано. Для наглядности вначале мы будем использовать его, а потом поговорим о других браузерах.

➡ Если один объект, например, `rabbit`, имеет специальную ссылку `__proto__` на другой объект `animal`, то все свойства, которые ищутся в `rabbit`, будут затем искаться в `animal`.

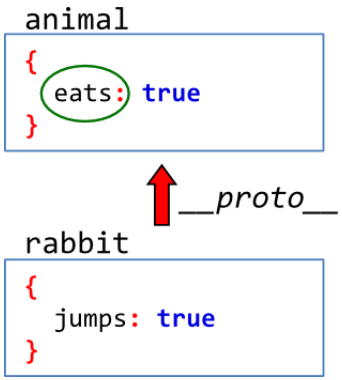
Пример кода (запускать в Firefox/Chrome/Safari):

```
1  var animal = { eats: true };
2  var rabbit = { jumps: true };
3
4  rabbit.__proto__ = animal;  // унаследовать
5
6  alert(rabbit.eats); // true
7  alert(rabbit.jumps); // true
```

Работает это так:

Когда запрашивается свойство `rabbit`, интерпретатор ищет его сначала в самом объекте `rabbit`, а если не находит — в объекте `rabbit.__proto__`, то есть, в данном случае, в `animal`.

Иллюстрация происходящего (поиск идет снизу вверх):



Объект, на который указывает `__proto__`, называется его *«прототипом»*.

Нет никаких ограничений на объект, который записывается в прототип. Например:

```
1  var rabbit = { };
2
3  rabbit.__proto__ = window;
4
5  rabbit.open('http://google.com'); // вызовет метод open из window
```

Ссылка __proto__ в спецификации

Если вы будете читать спецификацию EcmaScript — свойство `__proto__` обозначено в ней как `[[Prototype]]`.

Двойные квадратные скобки здесь важны, чтобы не перепутать его с совсем другим свойством, которое называется `prototype`, и которое мы рассмотрим позже.

Прототип используется только если свойство не найдено

Например, в следующем примере `eats` берется из самого объекта, до `animal` дело не доходит:

```
1 var animal = { eats: true };
2 var fedUpRabbit = { eats: false };
3
4 fedUpRabbit.__proto__ = animal;
5
6 alert(fedUpRabbit.eats); // false, свойство взято из fedUpRabbit
```

Прототип используется только при чтении

Любая запись значения, например, `rabbit.eats = value` или удаление `delete rabbit.eats` — работает напрямую с объектом.

Другими словами, прототип — это «резервное хранилище свойств и методов» объекта, автоматически используемое при поиске.

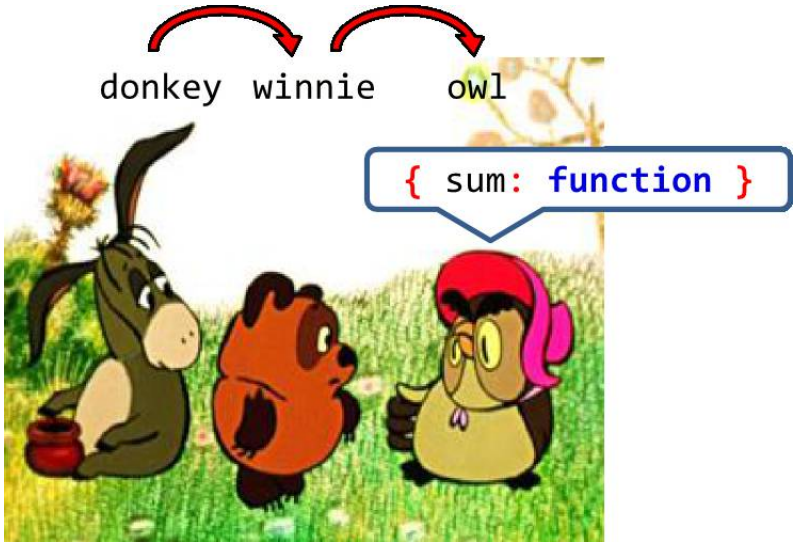
Цепочка прототипов

У объекта, который является `__proto__`, может быть свой `__proto__`, у того — свой, и так далее.

Например, цепочка наследования из трех объектов `donkey -> winnie -> owl`:

```
var owl = {
  sum: function(a, b) {
    return a+b;
  }
}

var winnie = { /* ... */ }
winnie.__proto__ = owl;

var donkey = { /* ... */ }
donkey.__proto__ = winnie;

alert( donkey.sum(2,2) );  // "4" ответит owl
```

Картина происходящего:



Создание объекта с данным прототипом

Как мы говорили ранее, свойство `__proto__` доступно в явном виде не во всех браузерах.

Поэтому используются другие способы для создания объектов с данным прототипом.

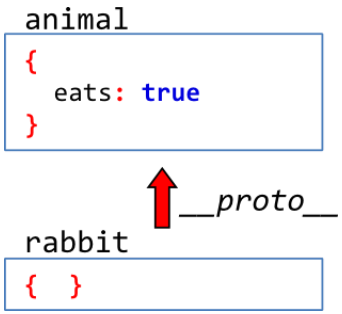
`Object.create(proto)` (кроме IE8-)

Вызов Object.create(proto) [1] создаёт пустой объект с данным прототипом `proto`.

Например:

```
var animal = { eats: true };

var rabbit = Object.create(animal);

alert(rabbit.eats); // true
```

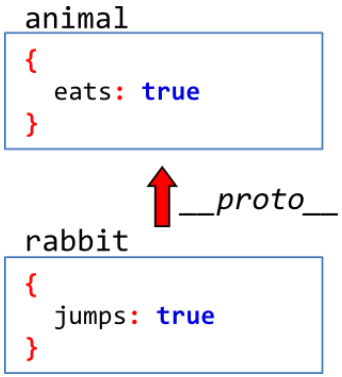
Этот код создал пустой объект `rabbit` с прототипом `animal`:



Мы можем добавить свойства в новый объект `rabbit`:

```
1  var animal = { eats: true };
2
3  var rabbit = Object.create(animal);
4
5  rabbit.jumps = true;
```

Станет:



Этот метод позволяет только создать новый пустой объект. Он не может изменить прототип существующего объекта.

У метода `Object.create` существует и второй необязательный аргумент (см. Дескриптор свойства, геттеры и сеттеры [2]), который позволяет задать свойства нового объекта. Но он никак не относится к наследованию, поэтому здесь мы его пропустим.

Свойство `F.prototype`

Созданием объектов часто занимается функция-конструктор. Чтобы таким объектам автоматически ставить прототип, существует свойство `prototype`.

Свойство `prototype` можно указать на любом объекте, но особый смысл оно имеет, если назначено функции.

При создании объекта через `new`, в его прототип `__proto__` записывается ссылка из `prototype` функции-конструктора.

Например:

```
var animal = { eats: true }

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

Rabbit.prototype = animal;

var rabbit = new Rabbit('John');

alert( rabbit.eats ); // true, т.к. rabbit.__proto__ == animal
```

Установка `Rabbit.prototype = animal` говорит интерпретатору следующее: *«запиши `__proto__ = animal` при создании объекта через `new Rabbit`».*

Значением `prototype` может быть только объект

Примитивное значение, такое как число или строка, будет проигнорировано.

Эмуляция `Object.create` для IE8-

Вызов `Object.create(proto)`, который создаёт пустой объект с данным прототипом, можно эмулировать, так что он будет работать во всех браузерах, включая IE6+.

Кросс-браузерный аналог — `inherit` состоит буквально из нескольких строк:

```
function inherit(proto) {
  function F() {}
  F.prototype = proto;
  var object = new F;
  return object;
}
```

Результат вызова `inherit(animal)` идентичен `Object.create(animal)`. Это будет новый пустой объект с прототипом `animal`.

Например:

```
var animal = { eats: true };

var rabbit = inherit(animal);

alert(rabbit.eats); // true
```

Посмотрите внимательно на функцию `inherit` и вы, наверняка, сами поймёте, как она работает...

Посмотрели? Подумали? Если да — раскройте ответ.

Посмотрим, как работает функция `inherit`, по шагам:

```
function inherit(proto) {
  function F() {}     // (1)
  F.prototype = proto // (2)
  var object = new F; // (3)
  return object;      // (4)
}
```

1. Создана новая функция `F`. Она ничего не делает с `this`, так что вызов `new F` вернёт пустой объект.
2. Свойство `F.prototype` устанавливается в будущий прототип `proto`
3. Результатом вызова `new F` будет пустой объект с `__proto__` равным значению `F.prototype`.
4. Готово! Мы получили пустой объект с заданным прототипом.

Эта функция широко используется в библиотеках и фреймворках.

Метод `Object.getPrototypeOf(obj)`

Вызов Object.getPrototypeOf(obj) [3] возвращает прототип `obj`.

Не поддерживается в IE8-.

Например:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);

alert( Object.getPrototypeOf(rabbit) === animal ); // true
```

Этот метод даёт возможность получить `__proto__`. но не изменить его.

Метод `obj.hasOwnProperty`

Метод obj.hasOwnProperty(prop) [4] есть у всех объектов. Он позволяет проверить, принадлежит ли свойство самому объекту, без учета его прототипа.

Например:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

alert( rabbit.hasOwnProperty('jumps') ); // true: jumps принадлежит rabbit

alert( rabbit.hasOwnProperty('eats') ); // false: eats не принадлежит
```

Перебор свойств объекта без прототипа

Цикл `for..in` перебирает все свойства в объекте и его прототипе.

Например:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

for (var key in rabbit) {
  alert (key + " = " + rabbit[key]); // выводит и "eats" и "jumps"
}
```

Иногда хочется посмотреть, что находится именно в самом объекте, а не в прототипе.

Это можно сделать, если профильтровать `key` через `hasOwnProperty`:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

for (var key in rabbit) {
  if ( !rabbit.hasOwnProperty(key) ) continue; // пропустить "не свои" свойства
  alert (key + " = " + rabbit[key]); // выводит только "jumps"
}
```

Итого

В этой главе мы рассмотрели основы наследования в JavaScript. Упорядочим эту информацию.

- Наследование реализуется через специальное свойство `__proto__` (в спецификации обозначено `[[Prototype]]`).

  Оно работает так: при попытке получить свойство из объекта, если его там нет, оно ищется в `__proto__` объекта.

- Замыкания и `this` с прототипами никак не связаны, они работают по своим правилам.

Установка прототипа `__proto__`:

- Firefox, Chrome и Safari дают полный доступ к `obj.__proto__`. Эта нестандартная возможность бывает полезна в целях отладки.
- Функция-конструктор при создании объекта устанавливает его `__proto__` равным своему `prototype`.
- `Object.create(proto)` создаёт пустой объект с прототипом `proto`.

  В IE8- этого метода нет, но его можно эмулировать при помощи следующей функции `inherit`:

  ```
  function inherit(proto) {
    function F() {}
    F.prototype = proto;
    return new F;
  }
  ```

Можно даже присвоить `Object.create = inherit`, чтобы иметь унифицированный вызов, но при этом стоит иметь в виду, что современные браузеры поддерживают также дополнительный второй аргумент `Object.create`, позволяющий задать свойства объекта, а `inherit` — нет.

Кроме того, есть следующие методы для работы с прототипом:

- `Object.getPrototypeOf(obj)` — получить прототип объекта `obj`, кроме IE8-
- `obj.hasOwnProperty(prop)` — возвращает `true`, если `prop` является свойством объекта `obj`, без учёта прототипа.

Проверка `obj.hasOwnProperty` используется, в частности, в `for..in` для перебора свойств именно самого объекта, без прототипа.

"Классы" в JavaScript

В JavaScript есть встроенные объекты: `Date`, `Array`, `Object` и другие. Они используют прототипы и демонстрируют концепцию «псевдоклассов», которую мы вполне можем применить и для себя.

Откуда методы у {} ?

Начнём мы с того, что создадим пустой объект и выведем его.

```
1 var obj = { };
2 alert( obj ); // "[object Object]" ?
```

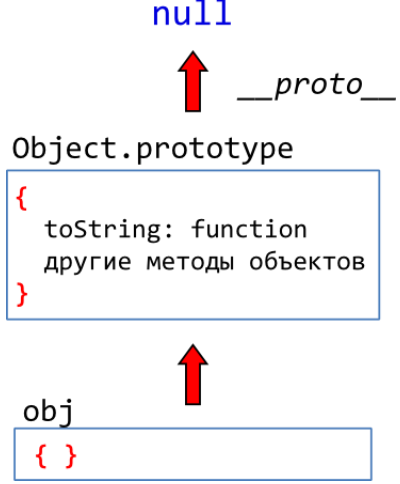
В объекте, очевидно, ничего нет... Но кто же тогда генерирует строковое представление для `alert(obj)`?

`Object.prototype`

...Конечно же, это сделал метод `toString`, который находится во встроенном прототипе `Object.prototype`. Этот прототип ставится всем объектам `Object`, и поэтому его методы доступны с момента создания.

В деталях, работает это так:

1. Запись `obj = {}` является краткой формой `obj = new Object`, где `Object` — встроенная функция-конструктор для объектов.
2. При выполнении `new Object`, создаваемому объекту ставится `__proto__` по `prototype` конструктора, в данном случае это будет `Object.prototype` — встроенный объект, хранящий свойства и методы, общие для объектов, в частности, есть `Object.prototype.toString`.
3. В дальнейшем при обращении к `obj.toString()` — функция будет взята из прототипа.

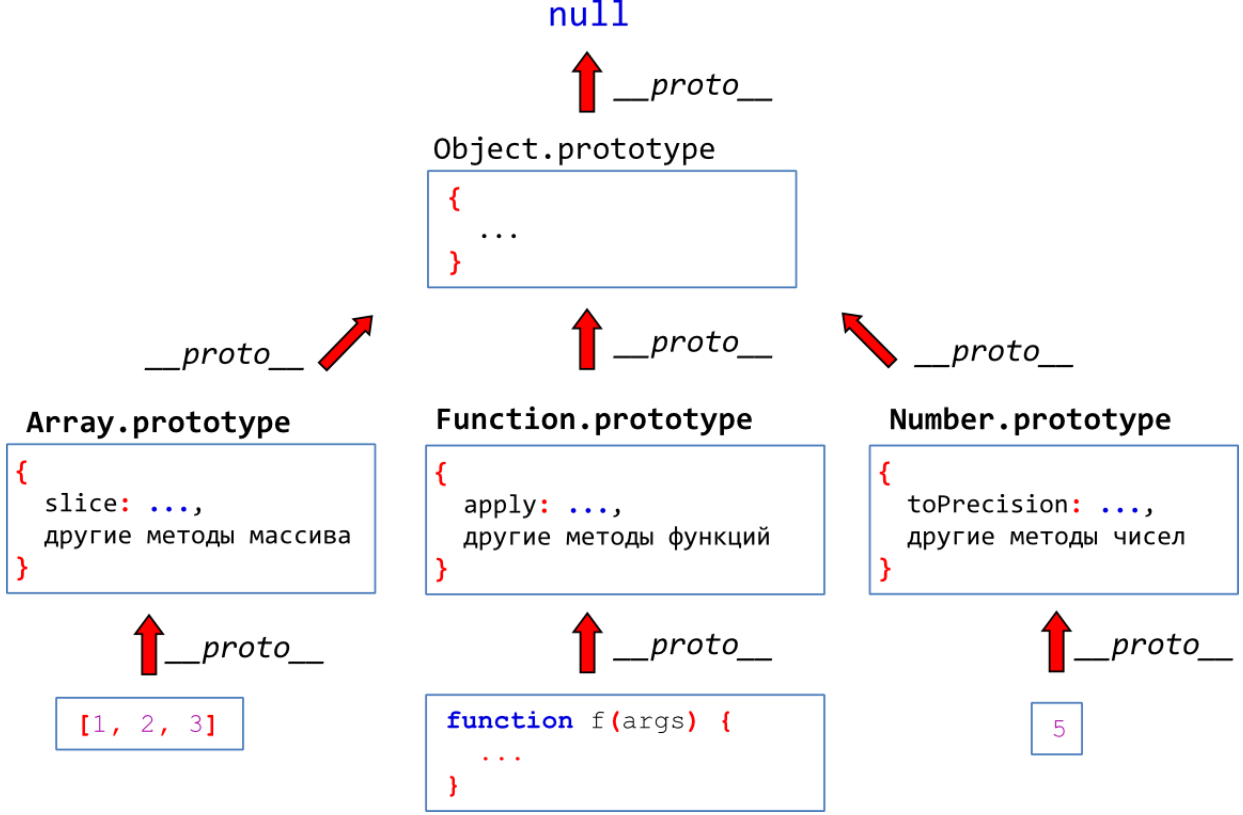


Это можно легко проверить:

```
var obj = { };

// метод берётся из прототипа?
alert(obj.toString == Object.prototype.toString); // true, да

// проверим протототип в Firefox, Chrome (где есть __proto__)
alert(obj.__proto__ == Object.prototype); // true
```

Встроенные «классы» в JavaScript

Точно такой же подход используется в массивах `Array`, функциях `Function` и других объектах. Встроенные методы для них находятся в `Array.prototype`, `Function.prototype` и т.п.



Как видно, получается иерархия наследования, которая всегда заканчивается на `Object.prototype`. Объект `Object.prototype` — единственный, у которого `__proto__` равно `null`.

Поэтому говорят, что «все объекты наследуют от `Object`». На самом деле ничего подобного. Это *все прототипы наследуют от `Object.prototype`*.

Некоторые методы при этом переопределяются. Например, у массива `Array` есть свой `toString`, который находится в `Array.prototype.toString`:

```
var arr = [1, 2, 3]
alert( arr ); // 1,2,3 <-- результат работы Array.prototype.toString
```

JavaScript ищет `toString`, сначала в `arr`, затем в `arr.__proto__ == Array.prototype`. Если бы там не нашёл — пошёл бы выше в `Array.prototype.__proto__`, который по стандарту (см. диаграмму выше) равен `Object.prototype`.

Методы `apply/call` у функций тоже берутся из встроенного прототипа `Function.prototype`.

```
function f() { }

alert( f.call == Function.prototype.call ); // true
```

Объявляем свои «классы»

Термины «псевдокласс», «класс» отсутствуют в спецификации ES5. Но их используют, потому что подход, о котором мы будем говорить, похож на «классы», используемые в других языках программирования, таких как C++, Java, PHP и т.п.

Классом называют функцию-конструктор вместе с её `prototype`.

Например: «класс `Object`», «класс `Date`» — это примеры встроенных классов. Мы можем использовать тот же подход для объявления своих.

Чтобы задать класс, нужно:

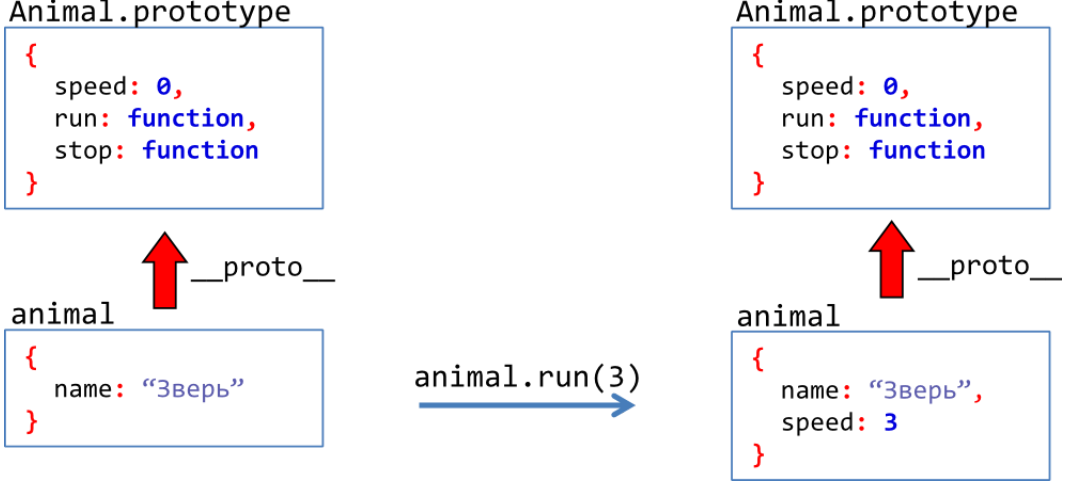
1. Объявить функцию-конструктор.
2. Записать методы и свойства, нужные всем объектам класса, в `prototype`.

Опишем класс `Animal`:

```
// конструктор
function Animal(name) {
  this.name = name;
}

// методы в прототипе
Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert(this.name + ' бежит, скорость ' + this.speed);
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert(this.name + ' стоит');
};

// свойство speed со значением "по умолчанию"
Animal.prototype.speed = 0;

var animal = new Animal('Зверь');

alert(animal.speed);                // 0, свойство взято из прототипа
animal.run(5);                      // Зверь бежит, скорость 5
animal.run(5);                      // Зверь бежит, скорость 10
animal.stop();                      // Зверь стоит
```

Здесь объекту `animal` принадлежит лишь свойство `name`, а остальное находится в прототипе.

Вызовы `animal.run()`, `animal.stop()` в примере выше изменяют `this.speed`.



При этом начальное значение `speed` берётся из прототипа, а новое — пишется уже в сам объект. И в дальнейшем используется. Это вполне нормально, но здесь есть важная тонкость.

Значения по умолчанию не следует хранить в прототипе в том случае, если это объекты

Посмотрите внимательно задачу ниже на эту тему.

Детали: свойство "constructor"

Свойство `constructor`, по замыслу, должно быть в каждом объекте и указывать, какой функцией создан объект.

Однако на практике оно ведет себя не всегда адекватным образом. Мы рассмотрим, как оно работает и почему именно так.

Свойство `constructor`

По замыслу, свойство `constructor` объекта должно содержать ссылку на функцию, создавшую объект, вот так:

```
1 function Rabbit() { }
2
3 var rabbit = new Rabbit();
4
5 alert( rabbit.constructor == Rabbit ); // true
```

Как видим, всё работает. Мы получили из объекта функцию, которая его создала.

Но всё не так просто. Расширим наш пример установкой `Rabbit.prototype`:

```
function Rabbit() { }

Rabbit.prototype = { jumps: true } ;

var rabbit = new Rabbit();

alert( rabbit.constructor == Rabbit ); // false (упс, потеряли конструктор!)
```

…Сломалось! Чтобы детальнее понять происходящее — посмотрим, откуда берется свойство `constructor` и что с ним произошло.

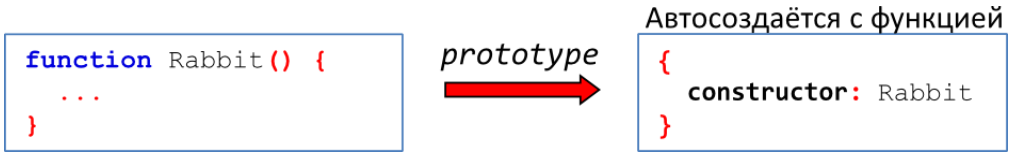
Значение prototype по умолчанию

Свойство `prototype` есть у каждой функции, даже если его не ставить.

Оно создается автоматически вместе с самой функцией, и по умолчанию является пустым объектом с единственным свойством `constructor`, которое ссылается обратно на функцию.

Вот такой вид имеет прототип по умолчанию:

```
Rabbit.prototype = {
  constructor: Rabbit
}
```



Так что при вызове `new Rabbit`, новый объект получает `rabbit.__proto__ = Rabbit.prototype`, и свойство `constructor` становится доступным из объекта, но на самом деле оно в прототипе.

Сделаем прямую проверку:

```
function Rabbit() { }

var rabbit = new Rabbit();

alert( rabbit.hasOwnProperty('constructor') ); // false, в объекте нет!

alert( Rabbit.prototype.hasOwnProperty('constructor') ); // true
```

Потеря constructor

При переопределении прототипа свойство `constructor` может из него пропасть. JavaScript никак не воспрепятствует этому.

Например:

```
function Rabbit() { }

Rabbit.prototype = {};  // (*)

var rabbit = new Rabbit();

alert( rabbit.constructor == Object ); // true
```

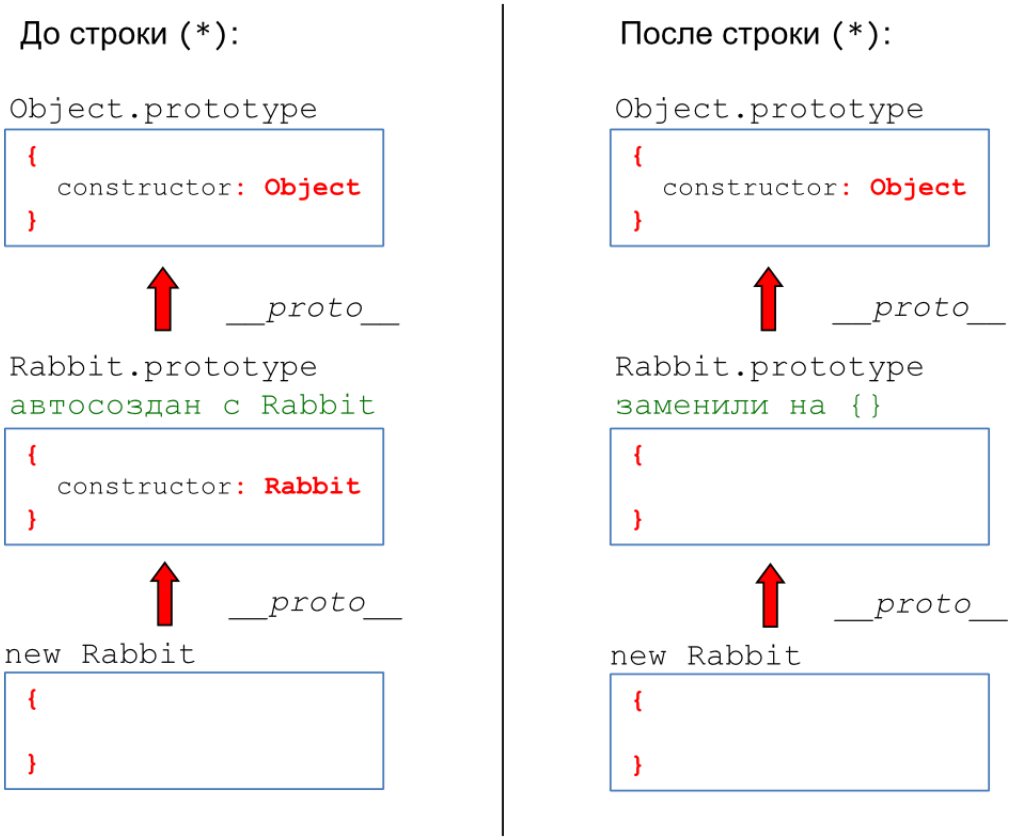
Ага! Но почему же конструктором является `Object`?

Это происходит потому что прототипом `rabbit` теперь является `new Object`.

При поиске `rabbit.constructor` JavaScript идёт по цепочке:
`rabbit -> Rabbit.prototype (new Object) -> Object.prototype`.

Здесь поиск заканчивается, так как существует встроенное свойство `Object.prototype.constructor`, равное `Object`.

Вот иллюстрация для лучшего понимания:



Итого

При создании функции, её `prototype` — это объект с единственным свойством `constructor`, которое указывает обратно на функцию.

Забавен тот факт, что больше нигде в спецификации JavaScript свойство `constructor` не упоминается. Интерпретатор не использует его никак и нигде.

В результате, в частности, `constructor` теряется при замене `prototype` на новый объект.

Таким образом, мы можем использовать это свойство, чтобы получить функцию-конструктор объекта, но при смене прототипа нужно смотреть, чтобы ненароком не перезаписать его.

Проверка прототипа: "instanceof"

Оператор `instanceof` позволяет проверить, создан ли объект данным конструктором, с учетом наследования.

Алгоритм работы `instanceof`

Рассмотрим пример:

```
function Rabbit() { }
var rabbit = new Rabbit;

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

Оператор `instanceof` никак не использует свойство `constructor`. Он опирается исключительно на цепочку `__proto__`.

Проверка `obj instanceof F` работает по следующей логике:

1. Получить `obj.__proto__`
2. Сравнить `obj.__proto__` с `F.prototype`
3. Если не совпадает, тогда заменить `obj` на `obj.__proto__` и повторить проверку на шаге два до тех пор, пока либо не найдется совпадение (результат `true`), либо цепочка не закончится (результат `false`).

В примере выше, совпадение найдено на первом же шаге, так как: `rabbit.__proto__ == Rabbit.prototype`.

Пример с проверкой на родителя более высокого уровня (`Object`):

```
function Rabbit() { };
var rabbit = new Rabbit;

alert(rabbit instanceof Object); // true, т.к. Rabbit наследует от Object
```

Здесь совпадение найдено на следующем шаге, т.к. `rabbit.__proto__.__proto__ == Object.prototype`.

Заметим, что при проверке `obj instanceof F` объект `obj` и сама функция `F` не участвуют в процессе проверки!

Оператор `instanceof` использует только цепочку прототипов для объекта и `F.prototype`.

Примитивы — не объекты, доказательство

Проверим, является ли число объектом `Number`:

```
alert(123 instanceof Number); // false, нет!
```

…С другой стороны, если создать встроенный объект `Number` (не делайте так):

```
alert( new Number(123) instanceof Number ); // true
```

Не друзья: `instanceof` и фреймы

Оператор `instanceof` не срабатывает, когда значение приходит из другого окна или фрейма.

Например, массив, который создан в ифрейме и передан родительскому окну — будет массивом *в том ифрейме*, но не в родительском окне. Проверка `instanceof Array` в родительском окне вернёт `false`.

Вообще, у каждого окна и фрейма — своя иерархия объектов и свой `window` .

Как правило, эта проблема возникает со встроенными объектами, в этом случае используется проверка внутреннего свойства `[[Class]]`. Более подробно это описано в главе Оператор typeof, [[Class]] и утиная типизация [5].

Итого

- Оператор `obj instanceof Func` проверяет тот факт, что `obj` является результатом вызова `new Func`. Он учитывает цепочку `__proto__`, поэтому наследование поддерживается.

  К примеру, массив является и объектом:

  ```
  var arr = [];
  alert(arr instanceof Array); // true
  alert(arr instanceof Object); // true
  ```

- Оператор `instanceof` не сможет проверить тип значения, если объект создан в одном окне/фрейме, а проверяется в другом. Это потому, что в каждом окне — своя иерархия объектов. Поэтому для проверки встроенных объектов используют свойство `[[Class]]`.

Наследование для классов в JavaScript

Классы — это не только конструктор с прототипом. Это ещё и дополнительные возможности по объектно-ориентированной разработке, в частности — наследование.

Примеры наследования мы встречаем и среди встроенных типов JavaScript. Например, массивы `Array` используют методы из своего прототипа `Array.prototype`, а если их там нет — то методы из родителя `Object.prototype`.

Мы можем использовать этот же подход для расширения собственных классов.

Наследование

Объявим класс `Rabbit`, который будет наследовать от `Animal`.

Вот так выглядят классы до наследования:

`Animal`:

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speed = 0;

Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert(this.name + ' бежит, скорость ' + this.speed);
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert(this.name + ' стоит');
};
```

`Rabbit`:

```
function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype.jump = function() {
  this.speed++;
  alert(this.name + ' прыгает');
};

var rabbit = new Rabbit('Кроль');
```

Наследование будет работать, если свойства будут искаться по цепочке `rabbit -> Rabbit.prototype -> Animal.prototype`.

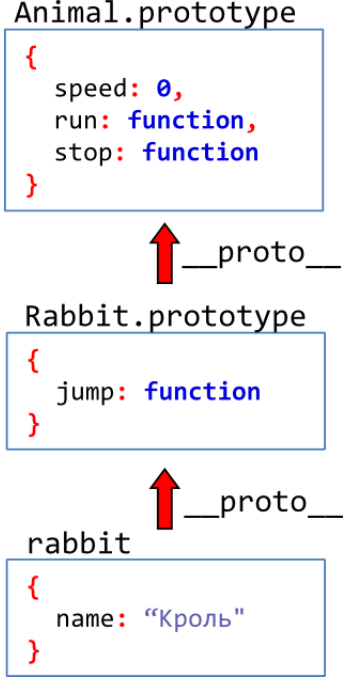
Для этого в `Rabbit.prototype` нужно поместить объект с прототипом `Animal.prototype`.

Реализуем это при помощи функции `Object.create` (для IE8- inherit [6]), разобранной нами ранее.

Класс `Animal` остаётся без изменений, а в `Rabbit` добавляется строка:

```
function Rabbit(name) {
  this.name = name;
}

// задаём наследование
Rabbit.prototype = Object.create(Animal.prototype);

// и добавим свой метод
Rabbit.prototype.jump = function() { ... };
```

Выглядеть иерархия будет так:



Итоговый код с двумя классами `Animal` и `Rabbit`:

```
// 1. Конструктор Animal
function Animal(name) {
  this.name = name;
}

// 1.1. Методы и свойства по умолчанию -- в прототип
Animal.prototype.speed = 0;

Animal.prototype.stop = function() {
  this.speed = 0;
  alert(this.name + ' стоит');
}

Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert(this.name + ' бежит, скорость ' + this.speed);
};

// 2. Конструктор Rabbit
function Rabbit(name) {
  this.name = name;
}

// 2.1. Наследование
Rabbit.prototype = Object.create(Animal.prototype);

// 2.2. Методы Rabbit
Rabbit.prototype.jump = function() {
  this.speed++;
  alert(this.name + ' прыгает, скорость ' + this.speed);
}
```

Альтернативный вариант: `Rabbit.prototype = new Animal`

Можно унаследовать от `Animal` и по-другому:

```
Rabbit.prototype = new Animal();
```

В этом случае мы получаем в прототипе не пустой объект с прототипом `Animal.prototype`, а реальный `Animal`.

Можно даже сконфигурировать его:

```
Rabbit.prototype = new Animal("Зверь номер два");
```

В этом случае новые `Rabbit` будут создаваться на основе конкретного экземпляра `Animal`. Это интересная возможность.

Но как правило мы не хотим создавать `Animal`, а хотим только унаследовать его методы. Поэтому такой код, используется чрезвычайно редко, почти никогда.

Вызов конструктора родителя

`Rabbit` может не только наследовать методы `Animal`, но и вызвать его конструктор, вот так:

```
function Rabbit(name) {
  Animal.apply(this, arguments);
  // ..после вызова родителя можно добавить к объекту что-то своё
}
```

При этом `Animal` выполнится в контексте текущего объекта, со всеми аргументами. Можно было бы использовать и `Animal.call(this, name)`, но `apply` надёжнее, так как при добавлении любых аргументов в конструктор они автоматически будут передаваться.

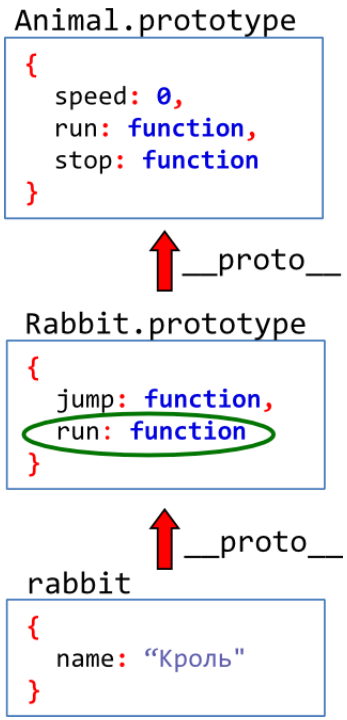
Переопределение метода

Итак, `Rabbit` наследует `Animal`. Теперь если какого-то метода нет в `Rabbit.prototype` — он будет взят из `Animal.prototype`.

В `Rabbit` может понадобиться заменить какие-то методы родителя на свои. Например, переопределим метод `run()`:

```
Rabbit.prototype.run = function(speed) {
  this.speed++;
  this.jump();
};
```

Вызов `rabbit.run()` теперь будет брать `run` из своего прототипа:



Кстати, можно назначить метод и на уровне одного объекта:

```
rabbit.run = function() {
  alert('Особый метод подпрыгивания для этого кролика');
};
```

...Но нужно это довольно редко.

Вызов метода родителя после переопределения

Естественно, кролик в своём методе `run` может обращаться к коду родителя. Получить этот метод можно напрямую из прототипа:

```
Rabbit.prototype.run = function() {
  Animal.prototype.run.apply(this, arguments);
  this.jump();
}
```

Обратите внимание на `apply` и явное указание контекста. Если вызвать просто `Animal.prototype.run()`, то в качестве `this` функция `run` получит `Animal.prototype`, а это неверно, нужен текущий объект.

Защищенные методы

При использовании наследования различают три основных ограничения доступа:

- **Приватное свойство (метод)** доступно только самому объекту.
- **Защищенное свойство (метод)** доступно самому объекту и его наследникам, но не доступно снаружи.
- **Публичное свойство (метод)** доступно отовсюду.

В JavaScript отсутствуют языковые средства для создания таких свойств. Потому прибегают к *соглашению о наименованиях*.

Свойства и методы, которые начинаются с подчеркивания "_", считаются защищенным. Порядочный программист должен обращаться к ним только из самого объекта и из его наследников.

Например:

```
Rabbit.prototype._age = 10; // свойство _age -- приватное

Rabbit.prototype._isOld = function() { // приватный метод
  return this._age > 10;
};

Rabbit.prototype.run = function() {
  if (this._isOld()) {
    alert('Слишком стар, чтобы бегать');
  }
};

// доступ снаружи осуществляется только к run
// _age и _isOld() - исключительно для внутренних нужд
```

Приватные методы

Приватные методы используются редко, так как защищённые уже обеспечивают отделение внутреннего интерфейса от внешнего.

Если уж очень хочется, то реализовать их можно, например обернув конструктор с прототипом в дополнительную функцию.

```
01 var Animal = (function() {
02   function Animal(name) { // Конструктор
03     this.name = name;
04   };
05
06   function privateMethod() { ... }    // Приватный метод
07
08   Animal.prototype.sit = function() {
09     // при вызове приватного метода нужно явно указать this, иначе
10     privateMethod.call(this); // он не будет знать об объекте
11     ...
12   };
13
14   return Animal;
15 }());
```

Этот код носит скорее теоретический характер, так как обычно защищённых методов хватает, и приватные не нужны.

Итого

- Для наследования нужно, чтобы «склад методов потомка» (`Child.prototype`) наследовал от «склада метода родителей» (`Parent.prototype`).

  Это можно сделать при помощи `Object.create` или её эмуляции inherit [7]` (IE8-):

  Код:

  ```
  Rabbit.prototype = Object.create(Animal.prototype);
  // Rabbit.prototype = inherit(Animal.prototype);
  ```

- Наследник может вызвать конструктор родителя в своём контексте, используя `apply(this, arguments)`, вот так:

  ```
  function Rabbit(...) {
    Animal.apply(this, arguments);
  }
  ```

- При переопределении метода родителя в потомке, к исходному методу можно обратиться, взяв его напрямую из прототипа:

  ```
  1 Rabbit.prototype.run = function() {
  2   var result = Animal.prototype.run.apply(this, ...);
  3   // result -- результат вызова метода родителя
  4 }
  ```

Структура наследования полностью:

```
// --------- Класс-Родитель ------------
// Конструктор родителя
function Animal(name) {
  this.name = name;
}

// Методы родителя
Animal.prototype.run= function() {
  alert(this + " бежит!")
}

Animal.prototype.toString = function() {
  return this.name;
}

// --------- Класс-потомок -----------
// Конструктор потомка
function Rabbit(name) {
  Animal.apply(this, arguments);
}

// Унаследовать
Rabbit.prototype = Object.create(Animal.prototype);

// Методы потомка
Rabbit.prototype.run = function() {
  Animal.prototype.run.apply(this); // метод родителя вызвать
  alert(this + " подпрыгивает!");
};

var rabbit = new Rabbit('Кроль');
rabbit.run();
```

Защищённые свойства начинают с подчёркивания `"_"`.

Технически, к ним можно обратиться отовсюду, но существует соглашение использовать их только в методах объекта.

Область применения наследования

До определённого момента JavaScript-разработчики действительно не сталкиваются с наследованием.

Поэтому иногда у начинающих специалистов возникает вопрос: нужно ли вообще наследование?

Наследование для расширения

Причина, по которой возникают такие вопросы, очень проста. На начальном этапе разработки наследование действительно не нужно. Ведь наследование — это способ расширения существующего функционала. А если интерфейс состоит из трёх кнопок и одного меню — там нечего расширять.

С другой стороны, пусть у нас есть меню `Menu`, с простыми методами `close()`, `open()` и событием `select`.

Проект растет. Через некоторое время может понадобиться создать расширенный вариант меню, например с анимацией при открытии. А

в другом месте проекта будет нужно, чтобы меню загружало своё содержимое с сервера при открытии… В третьем месте — еще что-то.

Конечно, можно учитывать возрастающие требования и добавлять возможности в функцию-конструктор `Menu`, но чем дальше, тем более громоздким объект будет становиться, тем сложнее будет продолжать работу с ним, просто потому что возможностей масса, и все они в одном месте.

- Если такое меню является частью библиотеки, то разработчик много раз подумает, прежде чем тащить в проект супер-универсальный комбайн с 50 методами, из которых ему нужны только 3.
- К тому же, работу с универсальным меню очень сложно осваивать, ведь много возможностей — это много документации.

Поэтому и было придумано *наследование*.

При помощи наследования, описываются объекты, *наследующие* от меню и модифицирующие его поведение. Там, где нужна модификация — используется наследник, там где нет — обычное меню.

Наследование для выноса общего функционала

Второй наиболее частый способ применения наследования — вынос базового функционала «за скобки», в общий родительский класс. Он применяется, в частности, для создания новых компонент.

Например, мы сделали в проекте меню `Menu`, дерево `Tree`, вкладки `Tabs`… И замечаем, что во всех них используются общие методы для работы с документом, элементами.

Наследование позволяет нам вынести эти методы в «базовый компонент» `Widget`, так что `Menu`, `Tree` и другие компоненты расширяют его и получают их сразу.

Итого

Эта статья — лишь введение в наследование, чтобы показать, где оно применимо. Конечно, тема наследования в ООП гораздо шире и раскрыта в литературе (см. книги [8]).

Создание интерфейсов имеет древнюю историю. В современном программировании наследование в них активно используется, включая написание на Java, C++, C# и других языках.

Поэтому довольно естественно, что оно может быть полезно и в JavaScript.

ООП: фреймворк Class.extend

В этой главе мы рассмотрим ООП-фреймворк:

- С удобным синтаксисом наследования.
- С вызовом родительских методов.
- С поддержкой статических методов и *примесей*.

Этот фреймворк представляет собой «синтаксический сахар» к наследованию на классах.

Идея и оригинальный код были предложены Джоном Ресигом: Simple JavaScript Inheritance [9], частично использовались ранее в Dojo Toolkit, в общем подход это не новый, но на мой взгляд один из лучших.

Полный код фреймворка: class-extend.js [10]. Он содержит много комментариев, чтобы было проще его понять, но смотреть его лучше после того, как ознакомитесь с возможностями.

Создание класса

Итак, начнём.

Фреймворк предоставляет всего один метод: `Class.extend`.

Чтобы представлять себе, как выглядит класс «на фреймворке», взглянем на рабочий пример:

```
// Объявление класса Animal
var Animal = Class.extend({

  init: function(name) {
    this.name = name;
  },

  run: function() {
    alert(this.name + " бежит!");
  }

});

// Создать (вызовется `init`)
var animal = new Animal("Зверь");

// Вызвать метод
animal.run(); // "Зверь бежит!"
```

Здесь были создан класс `Animal`, с использованием фреймворка.

Работает это так:

- Вызов `Class.extend(props)` объявляет и возвращает функцию-конструктор.
- В `prototype` конструктора он копирует свойства из `props`.
- Если среди них есть метод `init`, то он будет автоматически вызван при создании объекта.

В общем-то, на этом можно было бы и остановиться. Класс объявлен, все довольны.

Но есть и другие возможности.

Статические свойства

У метода `Class.extend` есть и второй, необязательный аргумент: объект `staticProps`.

Если он есть, то его свойства копируются в функцию-конструктор.

Например:

```
// Объявить класс Animal
var Animal = Class.extend({

  init: function(name){
    this.name = name;
  },

  toString: function(){
    return this.name;
  }

},
{ // статические свойства
  compare: function(a, b) {
    return a.name - b.name;
  }
});

var arr = [new Animal('Зорька'), new Animal('Бурёнка')]

arr.sort(Animal.compare);

alert(arr); // Бурёнка, Зорька
```

Наследование

Наследование с правильным фреймворком позволяет упростить синтаксис.

Метод `extend` копируется в создаваемые классы.

Поэтому его можно вызывать в любой момент, чтобы создать класс-наследник.

Например, унаследуем от `Animal`:

```
var Animal = Class.extend({
  init: function(name) {
    this.name = name;
  },
  run: function() {
    alert(this.name + ' бежит!');
  }
});

// Объявить класс Rabbit, наследующий от Animal
var Rabbit = Animal.extend({

  init: function(name) {
    this._super(name); // вызвать родительский init(name)
  },

  run: function() {
    this._super();     // вызвать родительский run
    alert('..и мощно прыгает за морковкой!');
  }

});

var rabbit = new Rabbit("Кроль");
rabbit.run(); // "Кроль бежит!", затем "..и мощно прыгает за морковкой!"
```

В коде выше появился ещё один замечательный метод: `this._super`.

Вызов `this._super(аргументы)` вызывает метод *родительского класса*, с указанными аргументами.

То есть, здесь он запустит родительский `init(name)`:

```
init: function(name) {
  this._super(name); // вызвать родительский init(name)
}
```

…А здесь — родительский `run`:

```
run: function() {
  this._super();     // вызвать родительский run
  alert('..и мощно прыгает за морковкой!');
}
```

Работает это, примерно, так: когда фреймворк копирует методы в прототип, он смотрит их код, и если видит там слово `_super`, то оборачивает метод в обёртку, которая ставит `this._super` в метод родителя, затем вызывает метод, а затем возвращает `this._super` как было ранее.

Это вызывает некоторые дополнительные расходы при объявлении, так как чтобы увидеть, обращается ли функция к `_super`, нужно получить её код через `toString` и обработать простым регулярным выражением. Как правило, эти расходы несущественны, но если они важны для вас — не составляет труда изъять эту возможность из фреймворка или оптимизировать так, чтобы более быстрое (через прототип) обращение к родителю генерировалась во время сжатия кода.

Кстати, как раз это и делает Google Closure Compiler, когда сжимает код, написанный на Google Closure Library. Сделать это легко, в том числе и для сжимателя UglifyJS.

Примеси

Согласно теории ООП, *примесь* (англ. mixin) — класс, реализующий какое-либо чётко выделенное поведение. Используется для *уточнения* поведения других классов, и не предназначен для порождения самостоятельно используемых объектов.

Такими поведениями может быть, например:

- Публикация событий и подписка на них.
- Работа с шаблонизатором.
- … любое поведение, дополняющее объект.

Иными словами, *примесь* позволяет легко добавить в существующий класс новые возможности.

Как правило, примесь реализуется в виде объекта, свойства которого копируются в прототип.

Есть шаблоны разработки, которые предполагают копирование нужных свойств и возможностей именно в объект, в частности — (Dependency Injection [11]), но они чаще используются в языках Java, C# и других, из-за некоторых их ограничений, которых нет в JavaScript.

В JavaScript их тоже можно применить, если речь идёт о больших приложениях. Этот паттерн идеально интегрируется с фреймворком, описанным в этой статье, но заслуживает отдельного обсуждения, поэтому мы здесь его не проходим.

Например, напишем примесь `EventMixin` для работы с событиями. Она будет содержать три метода — `on/off` (подписка) и `trigger` (генерация события):

```
var EventMixin = {
  _eventHandlers: {},

  on: function (eventName, handler) {
    if (!this._eventerHandlers[eventName]) {
      this._eventerHandlers[eventName] = [];
    }

    this._eventerHandlers[eventName].push(handler);
  },

  off: function(eventName, handler) {
    ...
  },

  trigger: function (eventName, args) {
    if (!this._eventerHandlers[eventName]) {
      return;
    }

    var handlers = this._eventerHandlers[eventName];
    for (var i = 0; i < handlers.length; i++) {
      handlers[i].apply(this, args);
    }
  }

};
```

Скопировав свойства из `EventMixin` в любой объект, мы дадим ему возможность генерировать события (`trigger`) и подписываться на них (`on/off`).

Чтобы было проще, во фреймворк добавлена возможность указания примесей при объявлении класса.

Для добавления примесей у метода `Class.extend` существует синтаксис с первым аргументом-массивом:

➡ **`Class.extend([mixin1, mixin2...], props, staticProps).`**

Если первый аргумент — массив, то его элементы `mixin1, mixin2..` записываются в прототип по очереди, перед `props`, примерно так:

```
for(var key in mixin1) prototype[key] = mixin1[key];
for(var key in mixin2) prototype[key] = mixin2[key];
...
for(var key in props) prototype[key] = props[key];
```

При этом, если названия методов совпадают, то последующий затрёт предыдущий, так как в объекте может быть только одно свойство с данным названием. Впрочем, обычно такого не происходит, т.к. примеси проектируются так, чтобы их методы были уникальными и ни с чем не конфликтовали.

Применение:

```
var Rabbit = Class.extend( [ EventMixin ], {

  /* свойства и методы для Rabbit */

});

var rabbit = new Rabbit();

rabbit.on("jump", function() { // повесить функцию на событие jump
  alert("jump &-@!");
});

rabbit.trigger('jump'); // alert сработает!
```

Примеси могут быть самыми разными. Например `TemplateMixin` для работы с шаблонами:

```
Rabbit = Class.extend([EventMixin, TemplateMixin], {

  /* Теперь Rabbit умеет использовать события и шаблоны */

});
```

Красиво, не правда ли? Всего лишь указали одну-другую примесь и объект уже всё умеет!

Примеси могут задавать и многое другое, например автоматически подписывать компонент на стандартные события, добавлять AJAX-функционал и т.п.

Итого

1. **Фреймворк имеет основной метод `Class.extend` с несколькими вариациями:**
   - `Class.extend(props)` — просто класс с прототипом `props`.
   - `Class.extend(props, staticProps)` — класс с прототипом `props` и статическими свойствами `staticProps`.
   - `Class.extend(mixins, props [, staticProps])` — если первый аргумент массив, то он интерпретируется как примеси. Их свойства копируются в прототип перед `props`.
2. **У созданных этим методом классов также есть `extend` для продолжения наследования.**
3. **Методы родителя можно вызвать при помощи `this._super(...)`.**

Плюсы и минусы:

- Такой фреймворк удобен потому, что класс можно задать одним вызовом `Class.extend`, с читаемым синтаксисом, удобным наследованием и вызовом родительских методов.
- Редакторы и IDE, как правило, не понимают такой синтаксис, а значит, не предоставляют автодополнение. При этом они обычно понимают объявление методов через явную запись в объект или в прототип.
- Есть некоторые дополнительные расходы, связанные с реализацией `_super`. Если они критичны, то их можно избежать.

То, как работает фреймворк, подробно описано в комментариях: class-extend.js [12].

ООП: функциональная реализация классов

При функциональном подходе все методы и свойства объекта объявляются в конструкторе, без прототипов.

У него есть ряд достоинств и недостатков, которые обычно делают его неподходящим, но иногда — весьма удобным.

Этот паттерн мы многократно использовали ранее для объявления объектов. В этой главе мы посмотрим, как реализовать наследование.

Объявление

Для примера рассмотрим конструктор `Menu`.

```
function Menu(options) {
  var self = this;

  var elem = options.elem;

  // ---------- приватные методы ----------

  function onTitleClick() {
    self.toggle();
  }

  // ---------- публичные методы ----------

  this.toggle = function() { ... };

  this.open = function() { ... };

  this.close = function() { ... };
}
```

Использование: `new Menu({ elem: элемент })`. Конструктор создаст объект меню и назначит обработчики.

Наследование

Для наследования конструктор наследника вызывает конструктор родителя в своём контексте.

Например, создадим функцию-конструктор `SlidingMenu`, которая будет *наследовать* `Menu` и переопределять метод `open`.

```
function SlidingMenu(menuId) {
  Menu.apply(this, arguments); // (1)

  var parentOpen = this.open;  // (2)

  this.open = function() {     // (3)
    parentOpen();              // (3.1)
    ...                        // (3.2)
  };

}
```

Рассмотрим подробно каждую строку этого примера.

1. Вызов `Menu.apply(this, arguments)` вызывает функцию-конструктор меню, передавая ей текущий объект `this` и аргументы.

   Она успешно срабатывает, инициализует меню и добавит в `this` публичные свойства и методы, например `this.open`.

2. Мы хотим заменить метод `this.open()` на свой, расширенный. Это мы сделаем в следующей строке `3.1`, а пока — скопируем *родительский метод open* в переменную.
3. Создадим наш собственный `this.open()`. Он перезапишет тот, который был создан `Menu`, в строке `(1)`. Чтобы сохранить доступ к старому методу, мы скопировали его в `3.1`.
   1. Запустить родительский метод.
   2. Затем наш собственный код...

`self` вместо `this`

При вызове `parentOpen()` не будет передан текущий `this`.

```
function SlidingMenu(options) {
  ...

  var parentOpen = this.open;
  this.open = function() {
    parentOpen(); // <-- текущий this не попадёт в parentOpen!
    ...
  }
}
```

Поэтому в коде родительского `open` необходимо использовать (заранее скопированное) `self` вместо `this`.

Также есть и другие способы фиксации `this`, например привязать метод к объекту через bind [13].

Защищенные свойства

Приватные свойства, такие как `elem`, реализованы через замыкание. Поэтому они недоступны в `SlidingMenu`.

В частности, переопределённый метод `open` не видит `elem`.

Чтобы разрешить доступ к `elem` из наследника, сделаем его защищённым.

Для этого в функциональном подходе, как и в классах, используется именование вида `this._elem` (свойство начинается с `"_"`).

Было (приватный доступ к `elem`):

```
function Menu(options) {
  var self = this;

  var elem = options.elem;

  this.open = function() {
    раскрыть elem
  }
  ...

}
```

Станет (защищённый доступ):

```
function Menu(options) {
  var self = this;

  this._elem = options.elem;

  this.open = function() {
    раскрыть this._elem
  }
  ...
}
```

Итого

Механизм наследования при функциональном объявлении класса выглядит так.

- Родительский конструктор записывает в объект свойства и методы.
- Используем `self` вместо `this`.
- Защищённые свойства записываются, начиная с подчёркивания `'_'`.

```
function Menu(options) {
  var self = this;

  this._elem = ...; // защищенное свойство

  this.open = function(arg) {
    ... /* используем self вместо this */ ...
  }

}
```

Для наследования конструктор-потомок:

1. Вызывает конструктор родителя в своём контексте.
2. Добавляет свои методы, переопределяя защищённые и публичные, полученные от родителя.

```
function SlidingMenu(options) {
  var self = this;

  // запустить конструктор родителя в контексте текущего объекта
  Menu.apply(this, arguments);

  // переопределяем метод: копируем старый в переменную
  var parentOpen = this.open; // (для будущего вызова, если будет нужен)

  this.open = function(arg) {  // создаём новый open
    parentOpen(arg); // вызвать родителя (если нужно)
    ...
  }
}
```

В следующей статье мы посмотрим область применения этого подхода и сравним его с классами на «прототипах».

ООП: почему наследование "на прототипах" - лучше

У функционального паттерна есть две серьёзные проблемы, которые ограничивают его применимость в реальных проектах.

Вызов публичного метода при инициализации

Код инициализации мы любим писать сверху конструктора:

```
function Menu(options) {
  var self = this;

  var elem = options.elem;

  // инициализовать меню
  privateInit();

  // а уже потом -- методы
  function privateInit() {
    ...
  }

  this.public = function() {
    ...
  }

}
```

Приватные методы при инициализации можно вызвать легко, так как они объявлены как `Function Declaration`, и значит — доступны до объявления.

```
function Menu() {

  privateInit();  // вызов будет работать
  ...

  function privateInit() { }
}
```

Но что, если мы захотим вызвать публичный или защищённый метод? Работать не будет!

В самом деле, такие методы присваиваются в `this` ниже по коду. В момент инициализации они не доступны.

Например:

```
function Menu() {

  // код инициализации
  ...
  this.public(); // ошибка, нельзя вызвать!

  // ..так как объявление метода идёт ниже.
  this.public = function() { ... }

  this._protected = function() {
    /* та же ситуация, защищённый метод вызвать выше нельзя */
  }

}
```

Что делать? Наше желание вполне обычно: при инициализации нужно уметь вызывать любые методы.

Есть решение.

Можно объявить публичный метод как локальную функцию, а затем присвоить его в `this`:

```
function Menu() {

  function public() { ... }

  ...
  this.public = public;
}
```

..Получится, что `public` является локальной функцией и можно вызывать его как изнутри виджета, так и снаружи.

Можно пойти дальше и объявлять вообще все методы как приватные, а потом копировать их во внешний интерфейс:

```
function Menu() {
  // код инициализации
  ...

  // методы
  function private() {  ... }
  function public() { ... }
  function protected() { ... }

  this.public = public;
  this._protected = protected;

}
```

Выделенную секцию можно заменить на более красивое перечисление методов, или как-то ещё незначительно улучшить код.

Теперь все методы можно вызывать из любого места класса.

Ограничения наследования

Итак, одну потенциальную проблему мы решили. Но функциональный паттерн обладает другим важным архитектурным ограничением.

Конструктор-наследник получает контроль лишь после полной инициализации родителя.

Например, конструктор `Menu` при инициализации создаёт меню, вызывает метод `render`, который создаёт нужные элементы на странице.

Потомок, естественно, хотел бы иметь возможность переопределить этот метод.

«Конечно» — скажем мы и сделаем его защищённым:

```
function Menu() {

  render();

  // ------ методы--------------

  function render() {
    ...Создать и украсить элемент...
  };

  // ------ защищённый метод ---
  this._render = render;
}
```

..Но действительно ли такой метод можно переопределить в потомке? Оказывается — нет!

Попробуем:

```
function SlidingMenu() {
  Menu.apply(this, arguments);

  // переопределить _render
  this._render = function() { ... }

  // ... но уже поздно!
  // конструктор Menu уже выполнился!
  // ... со своим, старым render!
}
```

Методы для инициализации, использованные родителем, переопределить нельзя.

Точнее, можно попытаться, но это будет слишком поздно, они уже вызвались. Это следствие того, что инициализация объединена с объявлением методов.

Недостаток весьма серьёзный. Фактически, он ограничивает возможности построения архитектуры. Красиво обойти его, увы, нельзя.

Заметим, что при использовании прототипов такой проблемы не возникает. Потому что сначала полностью задаётся иерархия, а *потом* создаются объекты. Мухи — отдельно, котлеты — отдельно.

Проверка `instanceof` работает частично

Есть и ещё одна проблема функционального подхода.

Проверка `new SlidingMenu() instanceof Menu` вернёт `false`. Иначе говоря, `instanceof` для потомков неполноценен.

Это вполне естественно, ведь `SlidingMenu` не является `Menu`.

Единственная связь между ними — конструктор `SlidingMenu` вызвал функцию `Menu` в своём контексте. Оператор `instanceof` работает через проверку цепочки прототипов, но здесь её нет.

На копирование методов требуется память и время

Все эти объявления `this.method = ...` — съедают время под обработку и память под конструкции.

Интерпретатор старается оптимизировать эти операции, но дополнительные расходы всё равно есть.

А при работе с прототипами — напротив, создаётся только сам объект.

Сравнение скорости

По идее, скорость доступа к методам из прототипа может быть медленнее, за счёт того, что браузеру их нужно ещё найти по цепочке `__proto__`.

Но в реальности, браузер закеширует место, где нашёл свойство и будет обращаться прямо туда.

В современных браузерах скорость доступа к методам в прототипе и в объекте одинакова.

Итого

Функциональный паттерн в сочетании с наследованием обладает рядом серьёзных проблем.

- При наследовании приватные свойства нужно заменять на защищённые — при этом пропадает основное преимущество функционального паттерна: красивые локальные переменные и функции без `this`.
- Инициализация объединена с созданием объекта. Поэтому потомок не может переопределить метод, используемый родителем при инициализации. Иначе говоря, «крылья ООП местами подрезаны».

Функциональный применим там, где проблем, описанных выше, точно не будет. Например, для описания объектов, которые создаются в единственном экземпляре:

```
var DragAndDropManager = new function() {
  var draggingObjects;

  function onMouseDown() {
    ...
  }

  this.public = ...
}
```

Если в проекте нужен единообразный стиль ООП, то лучше использовать прототипный подход. По крайней мере он гарантированно быстрее всего и не создаст в будущем проблем.

Расширение встроенных прототипов

Встроенные в JavaScript объекты можно расширять и изменять. Что интересно, изменение некоторых из них повлияет и на примитивы. Можно добавить методы стандартным числам, строкам, и не только.

Конструкторы `String`, `Number`, `Boolean`

Строки, числа, булевы значения в JavaScript являются примитивами. Но есть также и встроенные функции-конструкторы `String`, `Number`, `Boolean`.

Они существуют по историческим причинам. Можно вызвать `new Number(1)`, но это не рекомендуется. При этом создаются не примитивы «строка, число..», а объектные значения:

```
alert(typeof 1); // "number"

alert(typeof new Number(1)); // "object" ?!?
```

Можно ли, всё же, создать и использовать `new String/Number/Boolean`?

…Да, можно, но в ряде случаев получится откровенно бредовое поведение. Например:

```
var zero = new Number(0);

if (zero) {
  alert("число ноль -- true?!?");
}
```

Такая ситуация возникла потому, что `zero` является объектом, а объекты всегда `true`.

Единственное допустимое использование этих конструкторов — запуск в режиме обычной функции для преобразования типа. Например, `Number("12")` преобразует в число, так же как `+"12"`.

Автопреобразование примитивов

Несмотря на то, что в явном виде объекты String, Number, Boolean не создаются, их прототипы всё же используются. Они хранят методы строк, чисел, булевых значений.

Например, метод `slice` для строк хранится как `String.prototype.slice`.

При вызове метода на примитиве, например, `"строка".slice(1)`, происходит следующее:

1. Примитивное значение неявно преобразуется во временный объект `String`.
2. Затем ищется и вызывается метод прототипа: `String.prototype.slice`.
3. Результатом вызова `slice` является снова примитив, а временный объект уничтожается.

Посмотрим на интересное следствие такого поведения. Попытаемся добавить свойство строке:

```
var hello = "Привет мир!";

hello.test = 5; // запись свойства сработала, ошибки нет...

alert(hello.test); // ...читаем свойство -- выдаёт undefined!
```

Будет выведено `undefined`, так как присвоение произошло во временный объект, созданный для обработки обращения к свойству примитива. Этот временный объект тут же уничтожился вместе со свойством.

Конечно же, браузеры при таком преобразовании применяют оптимизации и, возможно, дополнительные объекты не создаются, но логика поведения — именно такая как описана.

А значит, методы для строк, чисел, булевых значений можно изменять и добавлять свои, в прототип…

Изменение встроенных прототипов

Встроенные прототипы можно изменять. В том числе — добавлять свои методы.

Мы можем написать метод для многократного повторения строки, и он тут же станет доступным для всех строк:

```
String.prototype.repeat = function(times) {
  return new Array(times+1).join(this);
};

alert( "ля".repeat(3) ) // ляляля
```

Аналогично мы могли бы создать метод `Object.prototype.each(func)`, который будет применять `func` к каждому свойству:

```
Object.prototype.each = function(f) {
  for (var prop in this) {
    var value = this[prop];
    f.call(value, prop, value); // вызовет f(prop, value), this=value
  }
}

// Попробуем! (внимание, пока что это работает неверно!)
var obj = { name: 'Вася', age: 25 };

obj.each(function(prop, val) {
  alert(prop); // name -> age -> (!) each
});
```

Обратите внимание — пример выше работает неправильно. Он выводит лишнее свойство `each`, т.к. цикл `for..in` перебирает свойства в прототипе. Встроенные методы при этом пропускаются, а наш метод — вылез.

В данном случае это легко поправить добавлением проверки `hasOwnProperty`:

```
Object.prototype.each = function(f) {

  for (var prop in this) {

    if (!this.hasOwnProperty(prop)) continue;

    var value = this[prop];
    f.call(value, prop, value);

  }

};

// Теперь все будет в порядке
var obj = { name: 'Вася', age: 25 };

obj.each(function(prop, val) {
  alert(prop); // name -> age
});
```

Здесь это сработало, теперь код работает верно. Но мы же не хотим добавлять `hasOwnProperty` в цикл по любому объекту! Поэтому…

Не добавляйте свойства в `Object.prototype`

Свойства, добавленные в `Object.prototype`, появятся во всех `for..in` циклах. Они в них будут лишними.

Современный стандарт и `for..in`

Встроенные свойства и методы не перебираются в `for..in`, так как у них есть специальный внутренний флаг `[[Enumerable]]`, установленный в `false`.

Современные браузеры (включая IE с версии 9) позволяют устанавливать этот флаг для любых свойств, используя специальные вызовы, описанные в главе Дескриптор свойства, геттеры и сеттеры [14]. При таком добавлении предупреждение станет неактуальным, так как они тоже не будут видны в `for..in`.

Есть объекты, которые не участвуют в циклах `for..in`, например строки, функции... С ними уж точно нет такой проблемы, и в их прототипы, пожалуй, можно добавлять свои методы.

Но здесь есть свои «за» и «против»:

- Методы в прототипе позволяют писать более короткий и ясный код.
- Новые свойства, добавленные в прототип из разных мест, могут конфликтовать между собой. Представьте, что вы подключили две библиотеки, которые добавили одно и то же свойство в прототип, но определили его по-разному. Конфликт неизбежен.
- Изменение встроенного прототипа влияет глобально, на весь код, и менять их не очень хорошо с архитектурной точки зрения.

Допустимо изменение прототипа встроенных объектов, которое добавляет поддержку метода из современных стандартов в те браузеры, где её пока нет.

Например, добавим `Object.create(proto)` в старые браузеры:

```
if (!Object.create) {

  Object.create = function(proto) {
    function F() {}
    F.prototype = proto;
    return new F;
  };

}
```

Существует даже библиотека es5-shim [15], которая предоставляет многие функции современного JavaScript для старых браузеров. Они добавляются во встроенные объекты и их прототипы.

Итого

- Методы встроенных объектов хранятся в их прототипах.
- Встроенные прототипы можно расширить или поменять.
- Добавление методов в `Object.prototype` ломает циклы `for..in`. Другие прототипы изменять не настолько опасно, но все же не рекомендуется во избежание конфликтов.

Отдельно стоит изменение с целью добавления современных методов в старые браузеры, таких как Object.create [16] , Object.keys [17] , Function.prototype.bind [18] и т.п. Это допустимо.

Дескриптор свойства, геттеры и сеттеры

В этой главе мы рассмотрим возможности, которые поддерживаются всеми современными браузерами, исключая IE<9, Opera<12, Safari<5.1.4.

Данный материал является ознакомительным. Те методы, которые невозможно эмулировать, в частности, любая работа с дескрипторами, не используются в других главах учебника. В целях обеспечения совместимости.

Дескрипторы

Кроме обычного присвоения свойства: `obj.prop = value`, можно объявить свойство вызовом Object.defineProperty [19] .

При этом появляется возможность поставить ряд важных внутренних параметров.

Синтаксис:

```
Object.defineProperty(obj, prop, descriptor)
```

Аргументы:

`obj`
: Объект, в котором объявляется свойство.

`prop`
: Имя свойства, которое нужно объявить или модифицировать.

`descriptor`
: Дескриптор — объект, который описывает поведение свойства. В нём могут быть следующие поля:

  `value`
  : Значение свойства, по умолчанию `undefined`

  `writable`
  : Значение свойства можно менять, если `true`. По умолчанию `false`.

  `configurable`
  : Если `true`, то свойство можно удалять, а также менять его в дальнейшем при помощи `defineProperty`. По умолчанию `false`.

  `enumerable`
  : Если `true`, то свойство будет участвовать в переборе `for..in`. По умолчанию `false`.

  `get`
  : Функция, которая возвращает значение свойства. По умолчанию `undefined`.

  `set`
  : Функция, которая записывает значение свойства. По умолчанию `undefined`.

Чтобы избежать конфликта, запрещено одновременно указывать значение `value` и функции `get/set`. Либо значение, либо функции для его чтения-записи, одно из двух. Также запрещено и не имеет смысла указывать `writable` при наличии `get/set`-функций.

Пример: обычное свойство

Создадим неизменяемое и неперечисляемое обычное свойство `name`:

```
var user = {};

Object.defineProperty(user, "name", {
  value: "Вася",
  writable: false,    // присвоение "user.name=" вызовет ошибку
  enumerable: false,  // свойства не будет в `for(key in user)`
  configurable: false // удаление "delete user.name" вызовет ошибку
});
```

Заметим, что ошибки произойдут только в строгом режиме соответствия стандарту, если браузер его поддерживает:

```
"use strict";

var user = {};

Object.defineProperty(user, "name", {
  value: "Вася",
  writable: false
});

// в strict mode присвоение "user.name=" вызовет ошибку
user.name = "Петя"
```

Без `use strict` соответствующая операция просто не сработает. Например:

```
var user = {};

Object.defineProperty(user, "name", {
  value: "Вася",
  writable: true, // можно писать
  configurable: false // но нельзя удалять
});

user.name = "Петя"; // всё в порядке, свойство writable
delete user.name;  // удаление не сработает, но ошибки без strict нет

alert(user.name); // Петя
```

Пример: геттер

Допустим, у нас есть имя `firstName` и фамилия `surname`. Можно создать свойство, которое возвращает полное имя `fullName`:

```
var user = {
  firstName: "Вася",
  surname: "Петров"
}

Object.defineProperty(user, "fullName", {
  get: function() {
    return this.firstName + ' ' + this.surname;
  }
});

alert(user.fullName); // Вася Петров
```

Пример: геттер и сеттер

Можно добавить и возможность указывать полное имя.

```
var user = {
  firstName: "Вася",
  surname: "Петров"
}

Object.defineProperty(user, "fullName", {

  get: function() {
    return this.firstName + ' ' + this.surname;
  },

  set: function(value) {
    var split = value.split(' ');
    this.firstName = split[0];
    this.surname = split[1];
  }
});

user.fullName = "Петя Иванов";
alert(user.firstName); // Петя
alert(user.surname); // Иванов
```

Пример: изменение существующего свойства

Свойство `toString`, объявленное обычным способом, будет участвовать в цикле `for..in`:

```
var user = {
  firstName: "Вася",
  surname: "Петров",

  toString: function() {
    return this.firstName + this.surname;
  }
}

for(key in user) {
  alert(key); // firstName, surname, toString
}
```

`Object.defineProperty` может помочь исключить `toString` из списка итерации. Достаточно поставить ему флаг `enumerable: false`:

```
var user = {
  firstName: "Вася",
  surname: "Петров",

  toString: function() {
    return this.firstName + this.surname;
  }
}

Object.defineProperty(user, "toString", { enumerable: false });

for(key in user) {
  alert(key); // firstName, surname, toString нет
}
```

Геттеры и сеттеры в литералах

Также можно задать геттеры/сеттеры для свойства в литеральном определении объекта. Для этого используется особый синтаксис: `get свойство` или `set свойство`. Например:

```
var user = {
  birthday: new Date(1987, 6, 1),

  get age() {  // геттер для свойства age
    var todayYear = new Date().getFullYear();
    return todayYear - this.birthday.getFullYear();
  }
}

alert(user.birthday);  // июль 1987
alert(user.age);       // текущий возраст
```

Геттеры и сеттеры для совместимости

Главная польза от геттеров и сеттеров — возможность получить контроль над свойством в любой момент.

Например, разрабатываем мы админку. И есть объект `User` с именем `name` и возрастом `age`:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

var pete = new User("Петя", 25);

alert(pete.age); // 25
```

Уже написано много кода, который использует эти свойства.

..Но вдруг — данные на сервере поменялись и теперь вместо возраста хранится дата рождения:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

var pete = new User("Петя", new Date(1987, 6, 1));
```

Но что делать со старым кодом, который выводит свойство `age`? Можно, конечно, найти все места и поправить их, но это долго, а иногда и невозможно (если вы делаете библиотеку для сторонних проектов, код в которых — чужой).

Геттеры позволяют обойти проблему легко и непринуждённо. Просто добавляем геттер `age`:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  Object.defineProperty(this, "age", {
    get: function() {
      var todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

var pete = new User("Петя", new Date(1987, 6, 1));

alert(pete.age); // получает возраст из даты рождения
```

Таким образом, геттеры и сеттеры позволяют нам свободно использовать свойства и заменять их на функции, когда нужно. Сохраняя совместимость внешнего интерфейса.

Другие методы работы со свойствами

Object.defineProperties(obj, descriptors) [20]

Позволяет объявить несколько свойств сразу:

```
var user = {}

Object.defineProperties(user, {
  firstName: {
    value: "Петя"
  },

  surname: {
    value: "Иванов"
  },

  fullName: {
    get: function() {
      return this.firstName + ' ' + this.surname;
    }
  }
});

alert( user.fullName ); // Петя Иванов
```

Object.getOwnPropertyDescriptor(prop) [21]

Возвращает дескриптор для свойства с `prop`.

Object.keys(obj) [22] , **Object.getOwnPropertyNames(obj) [23]**

Возвращает массив — список свойств объекта.

При этом `Object.keys` возвращает только `enumerable`-свойства, а `Object.getOwnPropertyNames` — все:

```
01 var obj = {
02   a: 1,
03   b: 2,
04   internal: 3
05 };
06
07 Object.defineProperty(obj, "internal", {enumerable: false});
08
09 alert( Object.keys(obj) ); // a,b
10 alert( Object.getOwnPropertyNames(obj) ); // a, internal, b
```

Object.preventExtensions(obj) [24]

Запрещает добавление свойств в объект.

Object.seal(obj) [25]

Запрещает добавление свойств, все текущие свойства делает `configurable: false`.

Object.freeze(obj) [26]

Запрещает добавление свойств, все текущие свойства делает `configurable: false, writable: false`.

Object.isExtensible(obj) [27] , **Object.isSealed(obj) [28]** , **Object.isFrozen(obj) [29]**

Возвращают `true`, если на объекте были вызваны методы `Object.preventExtensions/seal/freeze`.

Решения задач

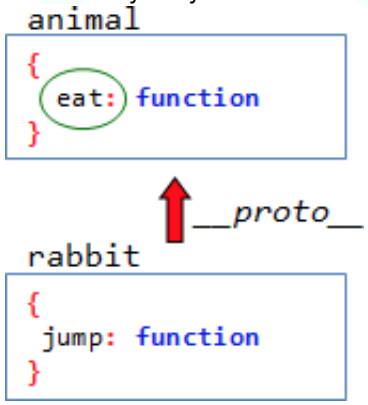
Решение задачи: Прототип и this

Ответ: свойство будет записано в `rabbit`.

Если коротко — то потому что `this` будет указывать на `rabbit`, а прототип при записи не используется.

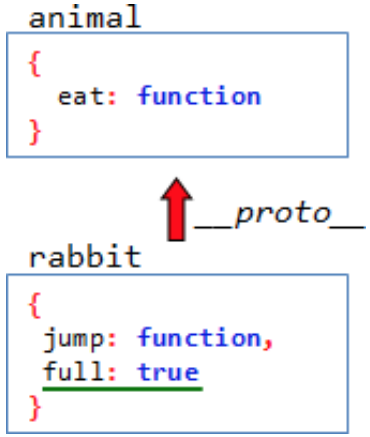
Если в деталях — посмотрим как выполняется `rabbit.eat()`:

1. Интерпретатор ищет `rabbit.eat`, чтобы его вызвать. Но свойство `eat` отсутствует в объекте `rabbit`, поэтому он идет по ссылке `rabbit.__proto__` и находит это свойство там.



2. Функция `eat` запускается. Контекст ставится равным объекту перед точкой [30], т.е. `this = rabbit`.

Итак — получается, что команда `this.full = true` устанавливает свойство `full` в самом объекте `rabbit`. Итог:



Решение задачи: Чему равно свойство после delete?

1. `true`, свойство взято из `rabbit`.
2. `null`, свойство взято из `animal`.
3. `undefined`, свойства больше нет.

Решение задачи: Алгоритм для поиска

1. Расставим `__proto__`:

```
var head = {
  glasses: 1
};

var table = {
  pen: 3
};
table.__proto__ = head;

var bed = {
  sheet: 1,
  pillow: 2
};
bed.__proto__ = table;

var pockets = {
  money: 2000
};
pockets.__proto__ = bed;

alert( pockets.pen ); // 3
alert( bed.glasses ); // 1
alert( table.money ); // undefined
```

2. **В современных браузерах, с точки зрения производительности, нет разницы, брать свойство из объекта или прототипа.** Они запоминают, где было найдено свойство и в следующий раз при запросе, к примеру, `pockets.glasses` начнут искать сразу в прототипе (`head`).

Решение задачи: Прототип после создания

1. Результат: `true`. Свойство `prototype` всего лишь задаёт `__proto__` у новых объектов. Так что его изменение не повлияет на `rabbit.__proto__`. Свойство `eats` будет получено из прототипа.
2. Результат: `false`. Свойство `Rabbit.prototype` и `rabbit.__proto__` указывают на один и тот же объект. В данном случае изменения вносятся в сам объект.
3. Результат: `undefined`. Удаление осуществляется из самого прототипа, поэтому свойство `rabbit.eats` больше взять неоткуда.
4. Результат был бы `true`, так как `delete rabbit.eats` попыталось бы удалить `eats` из `rabbit`, где его и так нет. А чтение в `alert` прошло бы из прототипа.

Решение задачи: Аргументы по умолчанию

Можно унаследовать от `options` и добавлять/менять опции в потомке:

```
function inherit(proto) {
  function F() {}
  F.prototype = proto;
  return new F;
}

function Menu(options) {
  var opts = inherit(options);
  opts.width = opts.width || 300;

  alert(opts.width);  // возьмёт width из opts
  alert(opts.height); // возьмёт height из options
  ...
}
```

Все изменения будут происходить в наследнике, а исходный объект останется незатронутым.

P.S. При этом нельзя *удалять* параметры. Вызов `delete opts.height` никак не повлияет на возможность получить `opts.height`, если это свойство находится в исходном объекте.

Решение задачи: Есть ли разница между вызовами?

- Первый вызов ставит `this == rabbit`, остальные ставят `this` равным прототипу, следуя правилу «`this` — объект перед точкой».
- При этом второй вызов не поддерживается в IE, т.к. свойство `__proto__` — нестандартное. А третий и четвёртый — идентичны.
- В Chrome идентичны три последних вызова.

Решение задачи: Хомяки с __proto__

Почему возникает проблема

Давайте подробнее разберем происходящее при вызове `speedy.found("яблоко")`:

1. Интерпретатор ищет свойство `found` в `speedy`. Но `speedy` — пустой объект, т.к. `new Hamster` ничего не делает с `this`.
2. Интерпретатор идёт по ссылке `speedy.__proto__` (`==Hamster.prototype`) и находят там метод `found`, запускает его.
3. Значение `this` устанавливается в объект перед точкой, т.е. в `speedy`.
4. Для выполнения `this.food.push()` нужно найти свойство `this.food`. Оно отсутствует в `speedy`, но есть в `speedy.__proto__`.
5. Значение `"яблоко"` добавляется в `speedy.__proto__.food`.

У всех хомяков общий живот! Или, в терминах JavaScript, свойство `food` изменяется в прототипе, который является общим для всех объектов-хомяков.

Заметим, что этой проблемы не было бы при простом присваивании:

```
this.food = something;
```

В этом случае значение записалось бы в сам объект, без поиска `found` в прототипе.

Проблема возникает только со свойствами-объектами в прототипе.

Исправьте её?

Исправление

Для исправления проблемы нужно дать каждому хомяку свой живот. Это можно сделать, присвоив его в конструкторе.

```
01 function Hamster() {
02   this.food = [];
03 }
04
05 Hamster.prototype.found = function(something) {
06   this.food.push(something);
07 };
08
09 speedy = new Hamster();
10 lazy = new Hamster();
11
12 speedy.found("яблоко");
13 speedy.found("орех");
14
15 alert(speedy.food.length) // 2
16 alert(lazy.food.length) // 0(!)
```

Теперь всё в порядке. У каждого хомяка — свой живот.

Решение задачи: Перепишите в виде класса

Решение: http://learn.javascript.ru/play/tutorial/prototype/menu-sketch.html

Решение задачи: Создать объект тем же конструктором

Обычно — можем. Например:

```
function User(name) {
  this.name = name;
}

var obj = new User('Вася');
var obj2 = new obj.constructor('Петя');

alert(obj2.name); // Петя (сработало)
```

.. Но это работает только потому, что `User.prototype.constructor == User`. Если кто-то перезапишет `User.prototype` (или заменит `constructor`), то код не сработает:

```
function User(name) {
  this.name = name;
}

User.prototype = {};

var obj = new User('Вася');
var obj2 = new obj.constructor('Петя');

alert(obj2.name); // undefined
```

В результате изменения конструктора в `obj2` создался пустой объект `new Object`.

Решение задачи: Странное поведение instanceof

Да, это выглядит достаточно странно, поскольку объект `a` не создавался функцией `B`.

Но методу `instanceof` на самом деле вообще не важна функция. Он смотрит на её `prototype` и сверяет его с цепочкой `__proto__` объекта.

В данном случае `a.__proto__ == B.prototype`, поэтому `instanceof` возвращает `true`.

По логике `instanceof` именно прототип задаёт «тип объекта», поэтому `instanceof` работает именно так.

Решение задачи: Что выведет instanceof?

Да, распознает.

Он проверяет наследование с учётом цепочки прототипов.

```
function Animal() { }

function Rabbit() { }
Rabbit.prototype = Object.create(Animal.prototype);

var rabbit = new Rabbit();

alert( rabbit instanceof Rabbit );
alert( rabbit instanceof Animal );
alert( rabbit instanceof Object );
```

Решение задачи: Что содержит constructor?

Нет, не распознает.

Свойство `constructor` содержится в `prototype` функции по умолчанию, интерпретатор не поддерживает его корректность.

Посмотрим, чему оно равно и откуда оно будет взято в данном случае.

Порядок поиска свойства `rabbit.constructor`, по цепочке прототипов:

1. `rabbit` — это пустой объект, в нём нет.
2. `Rabbit.prototype` — в него при помощи `Object.create` записан пустой объект, наследующий от `Animal.prototype`. Поэтому `constructor`'а в нём также нет.
3. `Animal.prototype` — у функции `Animal` свойство `prototype` никто не менял. Поэтому оно содержит `Animal.prototype.constructor == Animal`.

```
function Animal() { }

function Rabbit() { }
Rabbit.prototype = Object.create(Animal.prototype);

var rabbit = new Rabbit();

alert( rabbit.constructor == Rabbit ); // false
alert( rabbit.constructor ); // Animal
```

Вывод: если мы хотим опираться на свойство `constructor`, то придётся самим позаботиться о его наличии. В данном случае — присвоить:

```
Rabbit.prototype.constructor = Rabbit;
```

Решение задачи: Класс "часы"

Решение: http://learn.javascript.ru/play/tutorial/prototype/clock2prototype/index.html.

Решение задачи: Класс "расширенные часы"

Решение: http://learn.javascript.ru/play/tutorial/prototype/extendedclock/index.html.

Решение задачи: Меню с таймером для анимации

Решение: http://learn.javascript.ru/play/tutorial/prototype/menu-animating/index.html

- Константы состояний перенесены в прототип, чтобы `AnimatingMenu` их тоже унаследовал.

Решение задачи: Добавить функциям defer

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
}

function f() {
  alert("привет");
}

f.defer(1000); // выведет "привет" через 1 секунду
```

Решение задачи: Добавить функциям defer с аргументами

```
01 Function.prototype.defer = function(ms) {
02   var f = this;
03   return function() {
04     var args = arguments, context = this;
05     setTimeout(function() {
06       f.apply(context, args);
07     }, ms);
08   }
09 }
10
11 // проверка
12 function f(a, b) {
13   alert(a + b);
14 }
15
16 f.defer(1000)(1, 2); // выведет 3 через 1 секунду.
```

Ссылки

1. Object.create(proto) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/create
2. Дескриптор свойства, геттеры и сеттеры http://learn.javascript.ru/descriptors-getters-setters
3. Object.getPrototypeOf(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getPrototypeOf
4. Obj.hasOwnProperty(prop) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/HasOwnProperty
5. Оператор typeof, [[Class]] и утиная типизация http://learn.javascript.ru/type-detection
6. Inherit http://learn.javascript.ru/prototype#inherit
7. Inherit http://learn.javascript.ru/prototype#inherit
8. Книги http://learn.javascript.ru/books
9. Simple JavaScript Inheritance http://ejohn.org/blog/simple-javascript-inheritance/
10. Class-extend.js http://learn.javascript.ru/files/tutorial/js/class-extend.js
11. Dependency Injection http://ru.wikipedia.org/wiki/Внедрение_зависимости
12. Class-extend.js http://learn.javascript.ru/files/tutorial/js/class-extend.js
13. Bind http://learn.javascript.ru/bind
14. Дескриптор свойства, геттеры и сеттеры http://learn.javascript.ru/descriptors-getters-setters
15. Es5-shim https://github.com/kriskowal/es5-shim
16. Object.create https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/create
17. Object.keys https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/keys
18. Function.prototype.bind https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind
19. Object.defineProperty https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty
20. Object.defineProperties(obj, descriptors) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperties
21. Object.getOwnPropertyDescriptor(prop) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor
22. Object.keys(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/keys
23. Object.getOwnPropertyNames(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames
24. Object.preventExtensions(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/seal
25. Object.seal(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/seal
26. Object.freeze(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/freeze
27. Object.isExtensible(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isExtensible
28. Object.isSealed(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isSealed

29. Object.isFrozen(obj) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isFrozen
30. Объекту перед точкой http://learn.javascript.ru/this#four-scents-of-this