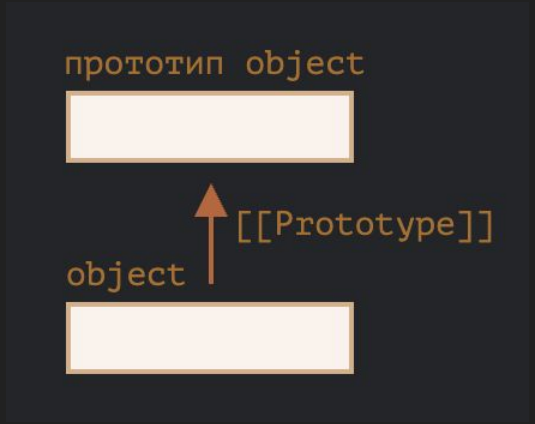
JavaScript
прототипное наследование

прототипное наследование;
F.prototype;
встроенные прототипы;
Методы прототипов, объекты без свойства __proto__;

Прототипное наследование

Прототипное наследование - не копировать/переопределять методы объекта, а просто создать новый объект на его основе другого.

- [[Prototype]]
- Операция записи не использует прототип
- Значение «this»
- Цикл for…in



[[Prototype]]

В JavaScript объекты имеют специальное скрытое свойство [[Prototype]] (так оно названо в спецификации), которое либо равно null, либо ссылается на другой объект. Этот объект называется «прототип».

Прототип даёт нам немного «магии». Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием».

Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование __proto__

__proto__

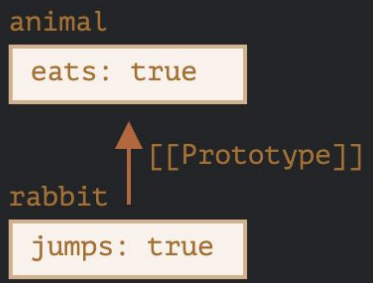
Есть только два ограничения:

- Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если мы попытаемся назначить __proto__ по кругу.
- Значение __proto__ может быть объектом или null. Другие типы игнорируются.

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// теперь мы можем найти оба свойства в rabbit:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```



__proto__ !== [[prototype]]

Свойство __proto__ — исторически обусловленный геттер/сеттер для [[Prototype]]

Обратите внимание, что __proto__ — не то же самое, что внутреннее свойство [[Prototype]]. Это геттер/сеттер для [[Prototype]]. Позже мы увидим ситуации, когда это имеет значение, а пока давайте просто будем иметь это в виду, поскольку мы строим наше понимание языка JavaScript.

Свойство __proto__ немного устарело, оно существует по историческим причинам. Современный JavaScript предполагает, что мы должны использовать функции Object.getPrototypeOf/Object.setPrototypeOf вместо того, чтобы получать/устанавливать прототип. Мы также рассмотрим эти функции позже.

По спецификации __proto__ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его. Так что мы вполне безопасно его используем.

Операция записи не использует прототип

Прототип используется только для чтения свойств.

Операции записи/удаления работают напрямую с объектом.

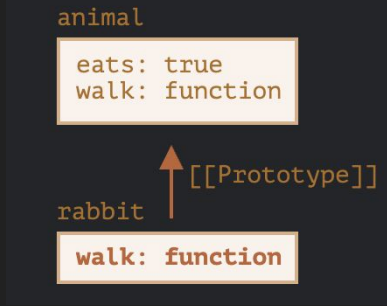
Теперь вызов rabbit.walk() находит метод непосредственно в объекте и выполняет его, не используя прототип.

```
let animal = {
  eats: true,
  walk() {
    /* этот метод не будет использоваться в rabbit */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```



Значение «this»

Неважно, где находится метод: в объекте или его прототипе. При вызове метода this — всегда объект перед точкой.

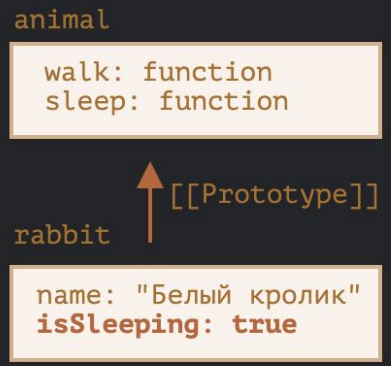
Это на самом деле очень важная деталь, потому что у нас может быть большой объект со множеством методов, от которого можно наследовать. Затем наследующие объекты могут вызывать его методы, но они будут изменять своё состояние, а не состояние объекта-родителя.

```
// методы animal
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// модифицирует rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)
```



Цикл for…in

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

Если унаследованные свойства нам не нужны, то мы можем отфильтровать их при помощи встроенного метода obj.hasOwnProperty(key): он возвращает true, если у obj есть собственное, не унаследованное, свойство с именем key.

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

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys возвращает только собственные ключи
alert(Object.keys(rabbit)); // jumps

// for..in проходит и по своим, и по унаследованным ключам
for(let prop in rabbit) alert(prop); // jumps, затем eats
```

Откуда взялся метод rabbit.hasOwnProperty?

Заметим ещё одну деталь. Откуда взялся метод rabbit.hasOwnProperty? Мы его явно не определяли. Если посмотреть на цепочку прототипов, то видно, что он берётся из Object.prototype.hasOwnProperty. То есть он унаследован.

…Но почему hasOwnProperty не появляется в цикле for..in в отличие от eats и jumps? Он ведь перечисляет все унаследованные свойства.

Ответ простой: оно не перечислимо. То есть у него внутренний флаг enumerable стоит false, как и у других свойств Object.prototype. Поэтому оно и не появляется в цикле.

Почти все остальные методы, получающие ключи/значения, такие как Object.keys, Object.values и другие – игнорируют унаследованные свойства.

Они учитывают только свойства самого объекта, не его прототипа.

F.prototype

Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем.

Установка Rabbit.prototype = animal буквально говорит интерпретатору следующее: "При создании объекта через new Rabbit() запиши ему animal в [[Prototype]]".

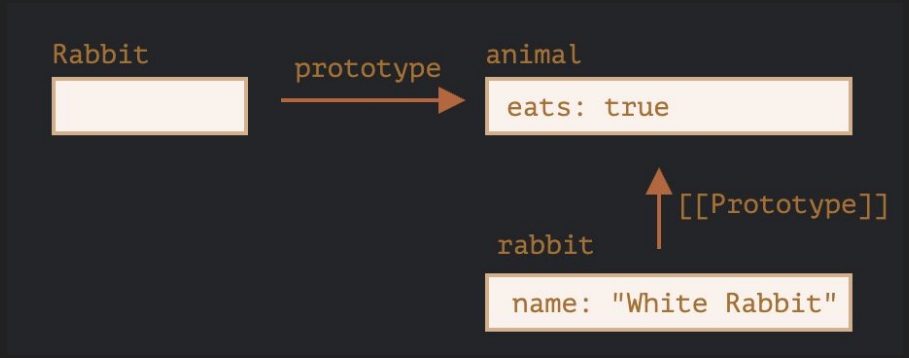
```
let animal = {
  eats: true
};

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

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal

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



F.prototype используется только в момент вызова new F

F.prototype используется только при вызове new F и присваивается в качестве свойства [[Prototype]] нового объекта.

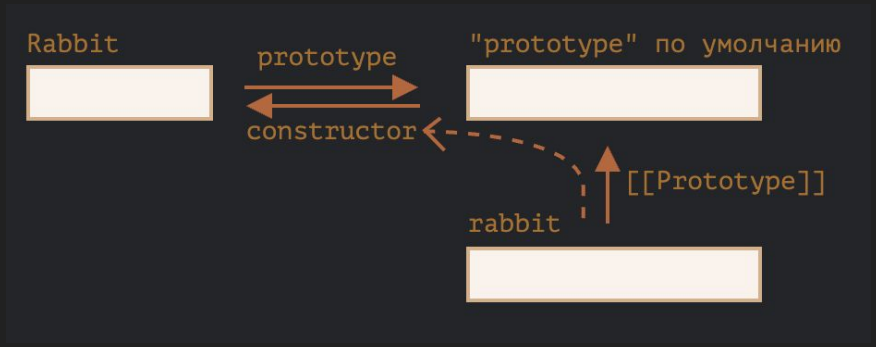
Если после создания свойство F.prototype изменится (F.prototype = <другой объект>), то новые объекты, созданные с помощью new F, будут иметь в качестве [[Prototype]] другой объект, а уже существующие объекты сохранят старый.

F.prototype по умолчанию, свойство constructor

У каждой функции (за исключением стрелочных) по умолчанию уже есть свойство "prototype".

По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор.

```
1  function Rabbit() {}
2
3  /* прототип по умолчанию
4  Rabbit.prototype = { constructor: Rabbit };
5  */
```




```
function Rabbit() {}
// по умолчанию:
// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true
```

…JavaScript сам по себе не гарантирует правильное значение свойства "constructor".

Да, оно является свойством по умолчанию в "prototype" у функций, но что случится с ним позже – зависит только от нас.

В частности, если мы заменим прототип по умолчанию на другой объект, то свойства "constructor" в нём не будет.

Таким образом, чтобы сохранить верное свойство "constructor", мы должны добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком.

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

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

```
function Rabbit() {}

// Не перезаписываем Rabbit.prototype полностью,
// а добавляем к нему свойство
Rabbit.prototype.jumps = true;
// Прототип по умолчанию сохраняется,
// и мы всё ещё имеем доступ к Rabbit.prototype.constructor
```

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// теперь свойство constructor снова корректное, так как мы добавили его
```

Встроенные прототипы

Свойство "prototype" широко используется внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

- Object.prototype
- Другие встроенные прототипы
- Примитивы
- Изменение встроенных прототипов
- Заимствование у прототипов

Object.prototype

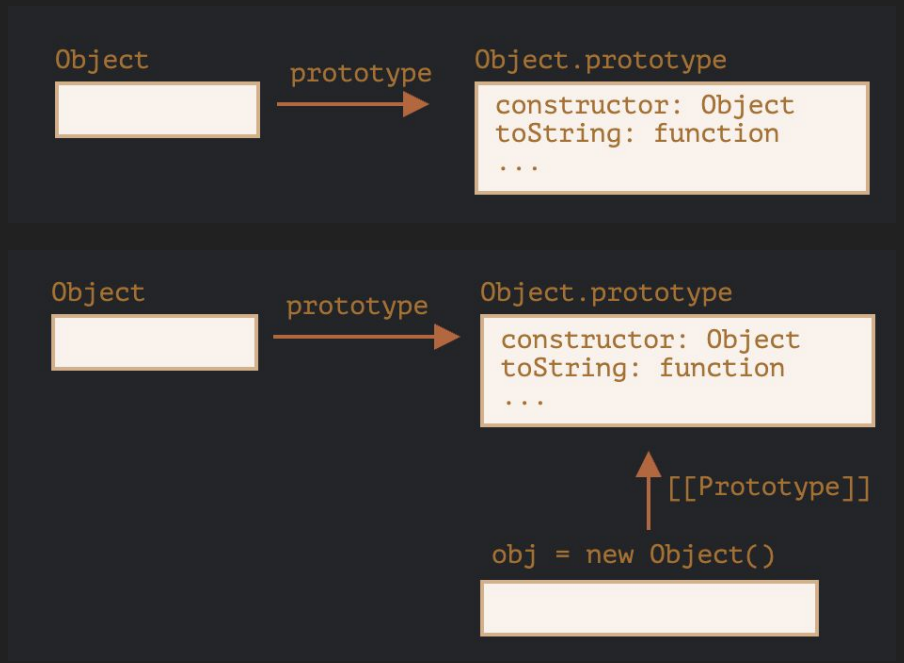
```
let obj = {};
alert( obj ); // "[object Object]" ?
```

Где код, который генерирует строку "[object Object]"? Это встроенный метод toString, но где он? obj ведь пуст!

…Но краткая нотация obj = {} – это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, которое ссылается на огромный объект с методом toString и другими.

Обратите внимание, что по цепочке прототипов выше Object.prototype больше нет свойства [[Prototype]]:

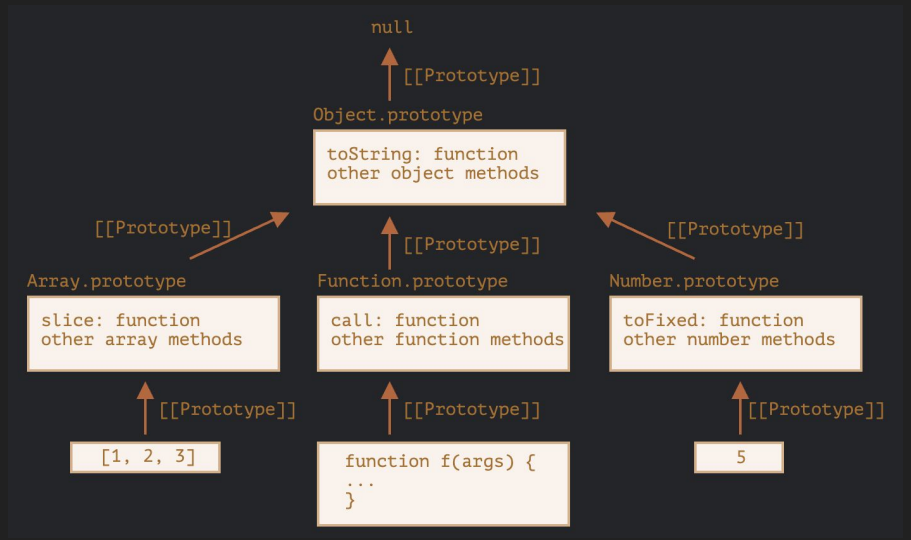
alert(Object.prototype.__proto__); // null



Другие встроенные прототипы

Другие встроенные объекты, такие как Array, Date, Function и другие, также хранят свои методы в прототипах.

Согласно спецификации, наверху иерархии встроенных прототипов находится Object.prototype. Поэтому иногда говорят, что «всё наследует от объектов».



Примитивы

Как мы помним, они не объекты. Но если мы попытаемся получить доступ к их свойствам, то тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number и Boolean, который предоставит методы и после этого исчезнет.

Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Значения null и undefined не имеют объектов-обёрток

Специальные значения null и undefined стоят особняком. У них нет объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. Например, если добавить метод к String.prototype, метод становится доступен для всех строк.

Важно:

Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

Так что, в общем, изменение встроенных прототипов считается плохой идеей.

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

Полифил

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов.

Полифил – это термин, который означает эмуляцию метода, который существует в спецификации JavaScript, но ещё не поддерживается текущим движком JavaScript.

```
if (!String.prototype.repeat) { // Если такого метода нет
  // добавляем его в прототип

  String.prototype.repeat = function(n) {
    // повторить строку n раз

    // на самом деле код должен быть немного более сложным
    // (полный алгоритм можно найти в спецификации)
    // но даже неполный полифил зачастую достаточно хорош для использования
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Заимствование у прототипов

Декораторы и переадресация вызова, call/apply - заимствование методов.

Заимствование - это когда мы берём метод из одного объекта и копируем его в другой.

Некоторые методы встроенных прототипов часто одалживают.

Например, если мы создаём объект, похожий на массив (псевдомассив), мы можем скопировать некоторые методы из Array в этот объект.

```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!
```

Методы прототипов, объекты без свойства __proto__

Свойство __proto__ считается устаревшим, и по стандарту оно должно поддерживаться только браузерами.

Современные же методы это:

- **Object.create(proto[, descriptors])** – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.
- **Object.getPrototypeOf(obj)** – возвращает свойство [[Prototype]] объекта obj.
- **Object.setPrototypeOf(obj, proto)** – устанавливает свойство [[Prototype]] объекта obj как proto.

"Простейший" объект

Как мы знаем, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение.

…Но если мы попробуем хранить созданные пользователями ключи (например, словари с пользовательским вводом), мы можем заметить интересный сбой: все ключи работают как ожидается, за исключением "__proto__".

Если пользователь введёт __proto__, присвоение проигнорируется!

Во-первых, мы можем переключиться на использование коллекции Map, и тогда всё будет в порядке.

Object.create(null) создаёт пустой объект без прототипа ([[Prototype]] будет null):

Таким образом не будет унаследованного геттера/сеттера для __proto__. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно.

Мы можем назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще, чем обычные объекты {...}.

Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как toString:

```
let obj = {};

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // [object Object], не "some value"!
```

```
let obj = Object.create(null);

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```

Ресурсы

- прототипное наследование - ТЫК
- F.prototype - ТЫК
- встроенные прототипы - ТЫК
- Методы прототипов, объекты без свойства __proto__ - ТЫК