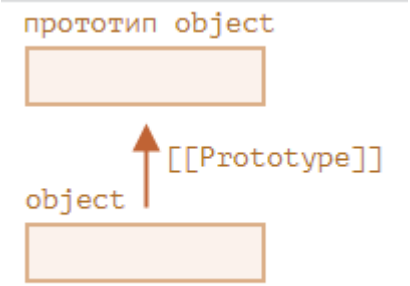
Прототипы

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»:



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__, например так:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal;
// теперь мы можем найти оба свойства в rabbit:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
// walk взят из прототипа
rabbit.walk(); // Animal walk
```

__proto__ — *не то же самое*, что [[Prototype]]. Это **геттер/сеттер** для него. В современном языке его заменяют функции Object.getPrototypeOf/Object.setPrototypeOf, которые также получают/устанавливают прототип.

Цепочка прототипов может быть длиннее. 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.

Это вполне очевидно, но всё же: может быть только один [[Prototype]]. Объект не может наследоваться от двух других объектов.

Прототип используется только для чтения свойств. Операции записи/удаления **работают напрямую с объектом**. Свойства-**аксессоры** – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции.

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

```
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};
let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};
```

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

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

```
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
```

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. Пример такой фильтрации:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true,
  __proto__: animal
};
for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);
  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

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

Как мы помним, новые объекты могут быть созданы с помощью функции-конструктора new F(). Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

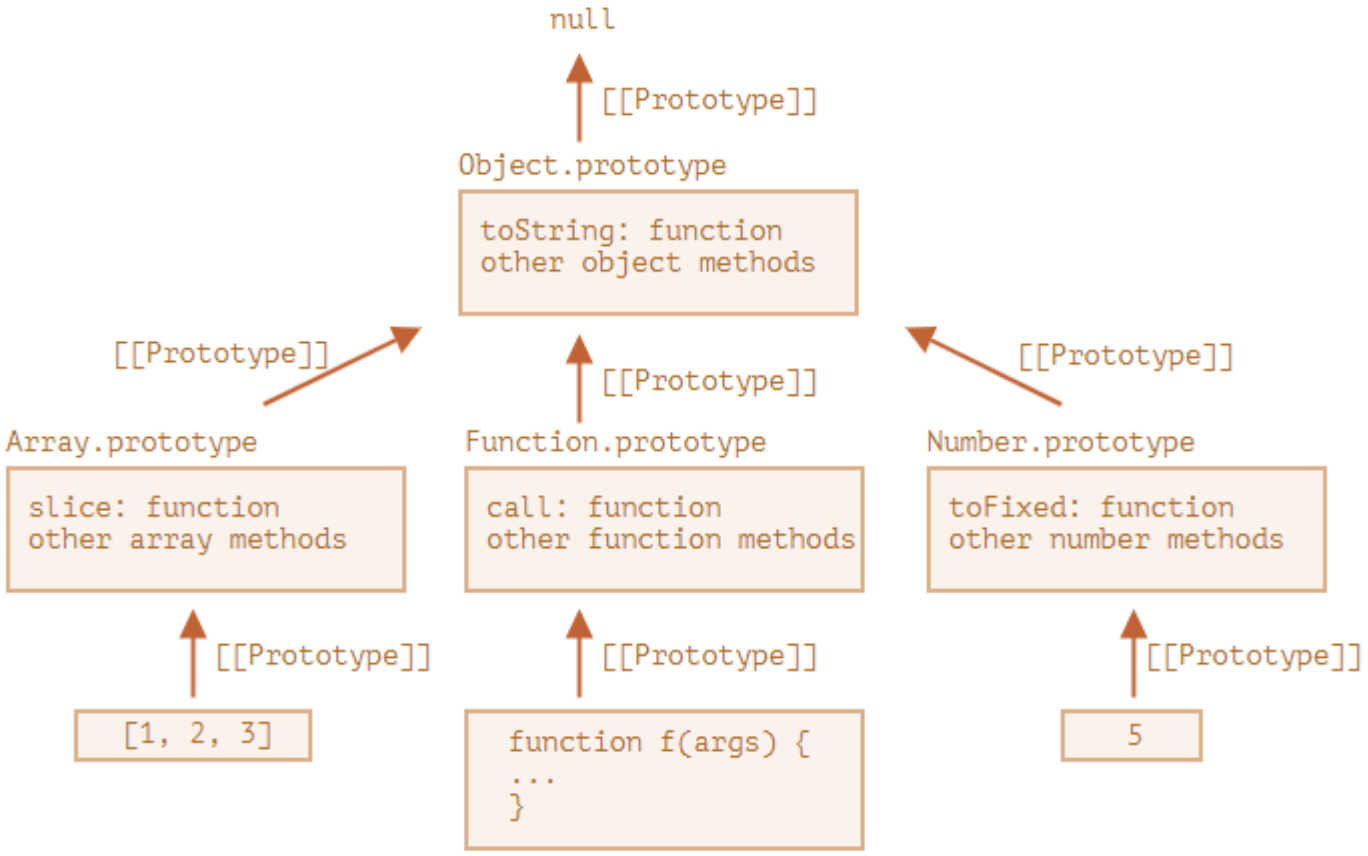
- Свойство F.prototype (не путать с [[Prototype]]) устанавливает[[Prototype]] для новых объектов при вызове new F().
- Значение F.prototype должно быть либо объектом, либо null. Другие значения не будут работать.
- Свойство "prototype" является особым, только когда оно назначено функции-конструктору, которая вызывается оператором new. В обычных объектах prototype не является чем-то особенным:

У каждой функции по умолчанию уже есть свойство `"prototype"`. По умолчанию `"prototype"` – объект с единственным свойством `constructor`, которое ссылается на функцию-конструктор.

```
function Rabbit() {}
// по умолчанию: Rabbit.prototype = { constructor: Rabbit }
let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}
alert(rabbit.constructor == Rabbit); // true (свойство получено из прототипа)
```

Встроенные прототипы

```
let arr = [1, 2, 3];
alert( arr.__proto__ === Array.prototype ); // true
alert( arr.__proto__.__proto__ === Object.prototype ); // true
alert( arr.__proto__.__proto__.__proto__ ); // null
```



Примитивы

Самое сложное происходит со строками, числами и булевыми значениями. 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

Изменение встроенных прототипов

Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – это добавление нового метода из стандарта, который ещё не поддерживается движком JavaScript (**полифил**).

Заимствование у прототипов

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. Пример:

```
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 прототопов 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.

Эти методы нужно использовать вместо __proto__. Например:

```
let animal = {
  eats: true
};
// создаём новый объект с прототипом animal
let rabbit = Object.create(animal);
alert(rabbit.eats); // true
alert(Object.getPrototypeOf(rabbit) === animal); // получаем прототип объекта rabbit
Object.setPrototypeOf(rabbit, {}); // заменяем прототип объекта rabbit на {}
```

Мы также можем использовать Object.create для «продвинутого» клонирования объекта, более мощного, чем копирование свойств в цикле for..in:

```
// клон obj с тем же прототипом (с поверхностным копированием свойств)
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Ещё методы:

- Object.keys(obj) / Object.values(obj) / Object.entries(obj) – возвращают массив всех перечисляемых собственных строковых ключей/значений/пар ключ-значение.
- Object.getOwnPropertySymbols(obj) – возвращает массив всех собственных символьных ключей.
- Object.getOwnPropertyNames(obj) – возвращает массив всех собственных строковых ключей.
- Reflect.ownKeys(obj) – возвращает массив всех собственных ключей.
- obj.hasOwnProperty(key): возвращает true, если у obj есть собственное (не унаследованное) свойство с именем key.