Прототипы

Гоголев Сергей

Студент

```
var student = {
    name: 'Billy',
    type: 'human',
    getName: function () {
        return this.name;
    },

    sleep: function () {
        console.info('zzZZZ...');
    }
};
```

```
student.getName();
// Billy
```

Преподаватель

```
var lecturer = {
    name: 'Sergey',
    type: 'human',
    getName: function () {
        return this.name;
    }

    talk: function () {}
};
```

Объекты студента и предподавателя похожи

```
var student = {
    name: 'Billy',
    type: 'human',
    getName: function() {
        return this.name;
    },
    sleep: function() {}
};
```

```
var lecturer = {
    name: 'Sergey',
    type: 'human',
    getName: function() {
        return this.name;
    }
    talk: function() {}
};
```

Личность

```
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

И так — три несвязанных объекта

```
var student = {
    name: 'Billy',
    sleep: function () {}
};
```

```
var lecturer = {
    name: 'Sergey',
    talk: function () {}
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

Заимствование метода

```
var student = {
    name: 'Billy',
};
```

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

```
person.getName.call(student); // this ссылается на student
// Billy
```

А хотелось бы как раньше ...

```
student.getName();
// Billy
```

[[Prototype]]

Для создания такой связи необходимо в специальное внутреннее поле [[Prototype]] одного объекта записать ссылку на другой

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <ссылка на person>
};
```

```
student.getName();
// Billy
```

Обратиться напрямую нельзя

```
student['[[Prototype]]'] = person; // Так не работает
```

Но можно через специальный set/get __proto__

```
student.__proto__ = person;
```

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <ссылка на person>
};
```

Итак, связали два объекта

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

Объект, на который указывает ссылка в [[Prototype]], называется прототипом

«person послужил прототипом для student»

Мы решили нашу задачу

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
var lecturer = {
    name: 'Sergey',
    [[Prototype]]: <person>
};
```

```
var person = {
    getName: function () {}
};
```

Поиск метода в прототипе

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {
        return this.name;
    }
};
```

```
student.getName(); // Метод, которого нет у объекта
```

Но когда поиск остановится?

```
var student = {
    name: 'Billy',
    sleep: function () {},
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {},
    [[Prototype]]: // ?
};
```

Интепретатор будет идти по цепочке прототипов
в поиске поля или метода, пока не встретит null
в поле [[Prototype]]

Object.prototype

```
var person = {
    type: 'human',
    getName: function () {},
    [[Prototype]]: <Object.prototype>
}
```

«Глобальный прототип для всех объектов»

Содержит общие методы для всех объектов

Object.prototype.toString()

```
Object.prototype = {
    toString: function () {}
};
```

```
student.toString();
// [object Object]


console.info('Hello, ' + student);
// Hello, [object Object]
```

Когда поиск остановится?

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    [[Prototype]]: <Object.prototype>
};
```

```
Object.prototype = {
    toString: function () {},
    [[Prototype]]: null
};
```

Array.prototype

```
var fruits = ['Apple', 'Banana', 'Potato'];
```

```
Array.prototype = {
    concat: function () {},
    slice: function () {},
    splice: function () {},

    forEach: function () {},
    filter: function () {},
    map: function () {},
    [[Prototype]]: <Object.prototype>
}
```

Содержит общие методы для всех массивов

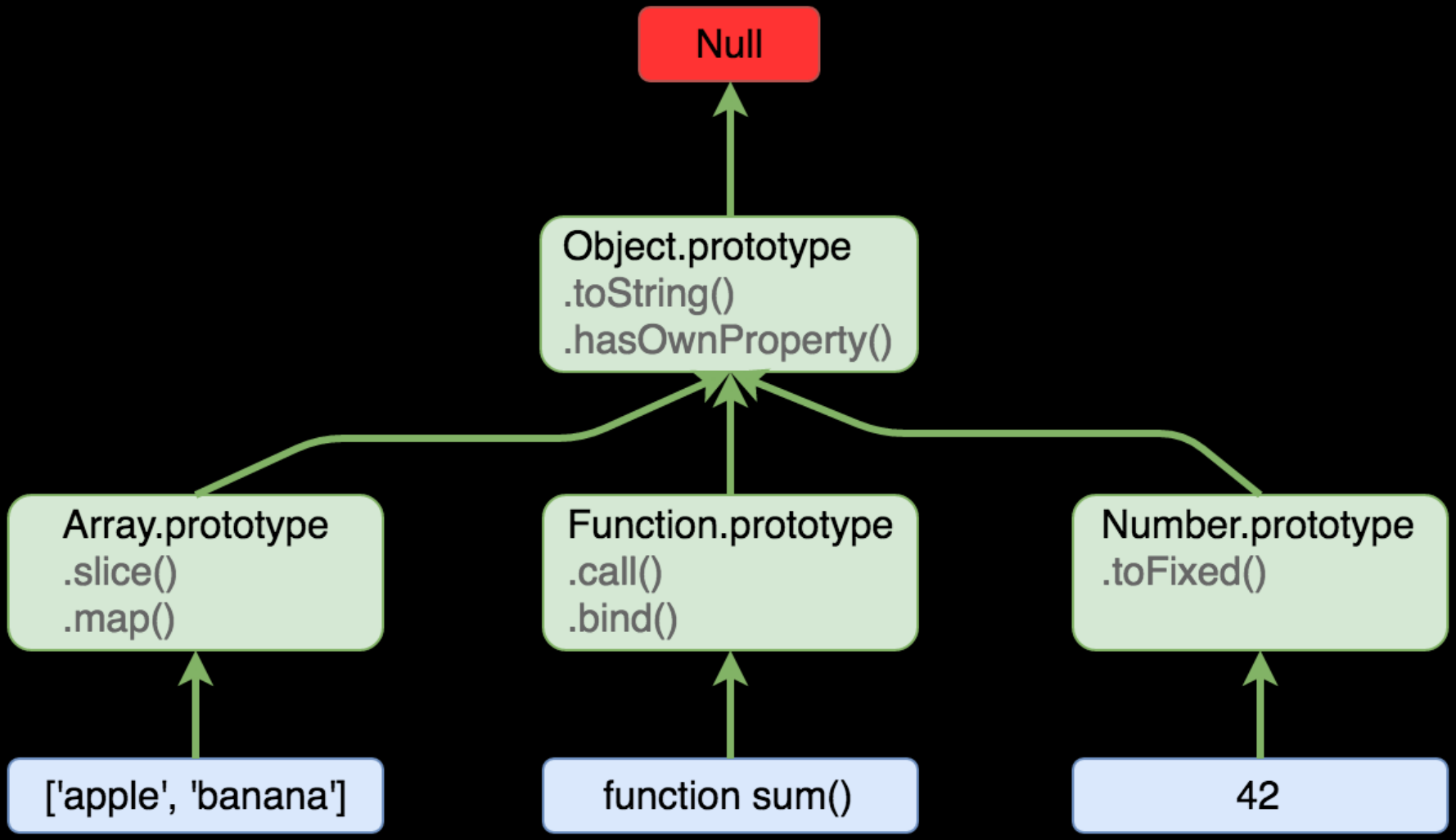
Function.prototype

```
function kawabanga () {
    console.info('Kawabanga!')
}
```

```
Function.prototype = {
    call: function () {},

    apply: function () {},
    bind: function () {},
    [[Prototype]]: <Object.prototype>
}
```

Содержит общие методы для всех функций


Null
Object.prototype
.toString()
.hasOwnProperty()
Array.prototype
.slice()
.map()
Function.prototype
.call()
.bind()
Number.prototype
.toFixed()
['apple', 'banana']
function sum()
42

Цикл в цепочке прототипов

```
var lecturer = { name: 'Sergey'}
var student = { name: 'Billy' }

lecturer.__proto__ = student;
student.__proto__ = lecturer;



console.info(lecturer.abrakadabra);
```

Uncaught TypeError: Cyclic __proto__ value

Ещё на строчке «student.__proto__ = lecturer»

Способы установки прототипа

Способы установки прототипа: __proto__

- Не является частью ECMAScript 5
- Поддерживается не всеми платформами
- Его появлению способствовали разработчики браузеров
- Появится в ECMAScript 6

Способы установки прототипа: Object.create()

```
var student = Object.create(person)
```

- Уже является частью ECMAScript 5
- Делает больше работы,
  чем простое присваивание ссылки
- Создаёт новые объекты и
  не может менять прототип существующих

Способы установки прототипа: Object.setPrototypeOf()

```
var student = {
    name: 'Billy',
    sleep: function () {}
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

```
Object.setPrototypeOf(student, person);

student.getName();
// Billy
```

Способы установки прототипа: Object.setPrototypeOf()

- Появился только в ECMAScript 6
- Близок к __proto__, но имеет особенность:

```
student.__proto__ = 42; // Неявно проигнорируется
```

```
Object.setPrototypeOf(student, 42);
```

TypeError: Object prototype may only be an Object or null

Object.getPrototypeOf()

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
```

```
var person = {
    type: 'human',
    getName: function () {}
}
```

```
Object.getPrototypeOf(student) === person;
// true
```

```
Object.getPrototypeOf(person) === Object.prototype;
// true
```

```
Object.getPrototypeOf(Object.prototype) === null;
// true
```

Особенности поведения полей

объекта с учётом прототипов

Установка своих полей объекта

```
var student = {
    name: 'Billy',
    sleep: function () {}
};
```

```
student.name = 'Willy';
```

```
console.info(student.name);
// Willy
```

Установка не своих полей объекта

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
```

```
var person = {
    type: 'human',
    getName: function () {}
}
```

```
console.info(student.type); // human
```

```
student.type = 'robot';
```

```
console.info(student.type); // robot
```

```
console.info(person.type); // ???
```

Установка не своих полей объекта

```
console.info(person.type); // 'human'
```

```
var student = {
    name: 'Billy',
    type: 'robot',
    [[Prototype]]: <person>
}
```

```
var person = {
    type: 'human',
    getName: function () {}
}
```

Такой эффект называется

затенением свойств

Object.prototype.toString()

```
Object.prototype = {
    toString: function () {}
};
```

```
student.toString();
// [object Object]


console.info('Hello, ' + student);
// Hello, [object Object]
```

Object.prototype.toString()

```
var student = {
    name: 'Billy'
    toString: function () {
        return this.name;
    }
};
```

```
student.toString();
// Billy

console.info('Hello, ' + student);
// Hello, Billy
```

Установка полей со свойствами

writable — помечает поле как изменяемое

Read-only поля

```
var student = { name: 'Billy' };
```

```
Object.defineProperty(student, 'gender', {
    writable: false,
    value: 'male',
});
```

```
console.info(student.gender); // male
```

```
student.gender = 'robot';

console.info(student.gender); // male
```

Неявное поведение!

Read-only поля use strict;

```
'use strict';

var student = { name: 'Billy' };

Object.defineProperty(student, 'gender', {
    writable: false,

    value: 'male'
});
```

```
student.gender = 'robot';
```

TypeError: Cannot assign to read only property 'gender' of object

Read-only поля в прототипах use strict;

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

```
Object.defineProperty(person, 'planet', {
    writable: false,
    value: 'Earth'
});
```

```
console.info(student.planet); // Earth
```

```
student.planet = 'Mars';
```

TypeError: Cannot assign to read only property 'planet' of object

Свойства полей

set/get — переопределяет установку/чтение

Set/get поля

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
Object.defineProperty(student, 'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});
```

```
student.age = '20 лет';
```

```
console.info(student.age); // 20;
```

Set/get поля в прототипах

```
var student = {
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human'
};
```

```
Object.defineProperty(person, 'age', {
    set: function(age) { this._age = parseInt(age); },
    get: function() { return this._age; }
});

student.age = '20 лет';
console.info(student.age); // 20;
```

```
student.hasOwnProperty(age); // false;
```

Затенение не работает

Свойства полей

enumerable — помечает как перечисляемое

Перечисляемые поля

```
var student = {
    name: 'Billy',
    age: 20
};
```

```
for (var key in student) console.info(key);
```

```
// name, age
```

Перечисляемые поля в прототипах

```
var student = {
    name: 'Billy',
    age: 20,
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

```
for (var key in student) console.info(key);
```

```
// 'age', 'name', 'type', 'getName'
```

*Оператор in проверяет наличие свойства не только у объекта,
но и в цепочке прототипов

Object.prototype.hasOwnProperty()

```
var student = {
    name: 'Billy',
    age: 20,
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human',
    getName: function () {}
};
```

```
for (var key in student)
    if (student.hasOwnProperty(key)) console.info(key);
```

```
// 'age', 'name'
```

Object.keys()

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
}
```

```
var person = {
    type: 'human',
    getName: function () {}
}
```

```
var keys = Object.keys(student); // Получаем массив ключей

console.info(keys);
```

```
// ['name']
```

Неперечисляемые поля

```
var student = { name: 'Billy' };
```

```
Object.defineProperty(student, 'age', {
  enumerable: false,
  value: '20'
});
```

```
for (var key in student) console.info(key);
// 'name'

Object.keys(student);
// ['name']
```

Неперечисляемые поля в прототипах

```
var student = {
    name: 'Billy',
    [[Prototype]]: <person>
};
```

```
var person = {
    type: 'human'
};
```

```
Object.defineProperty(person, 'age', {
    enumerable: false
});
```

```
for (var key in student) console.info(key);
```

```
// 'name', 'type'
```

Неперечисляемые поля по умолчанию

```
Object.prototype = {
    toString: function () {},
    [[Prototype]]: null
};

var person = {

    type: 'human',
    [[Prototype]]: <Object.prototype>
};
```

```
for (var key in person) console.info(key);
```

```
// 'type'
```