Яндекс

Конструкторы и прототипы (часть 2)

Евгений Марков, разработчик интерфейсов

На прошлой лекции

- Внутреннее свойство [[Prototype]]
- Внутреннее свойство [[HomeObject]] и ключевое слово super
- Цепочки прототипов и алгоритм поиска методов и свойств
- Object.create, Object.getPrototypeOf, Object.setPrototypeOf
- instance.hasOwnProperty и Object.getOwnPropertyNames

На прошлой лекции мы поговорили про

- Внутреннее поле [[Prototype]]
- Узнали что можно строить цепочки прототипов и поиск методов и свойств в объекте осуществляется последовательно от начала этой цепочки
- Зафиксировали для себя, что Object.prototype - прототип для всех создаваемых объектов. Можно это обойти и с помощью Object.create создать объект с null ссылкой на [[Prototype]].
- Узнали как создавать объекты с указанием ссылки на прототип и как затем получать/менять её с помощью getPrototypeOf и setPrototypeOf
- Познакомились с ключевым словом super и внутренним свойством [[HomeObject]]
- Вспомнили какие атрибуты есть у свойств объекта, как перебирать свойства и какие есть особенности связанные с прототипами и как с этими особенностями работать имея методы hasOwnProperty и getOwnPropertyNames

Сегодня

- Функции-фабрики
- Функции-конструкторы
- Классы

Сегодня поговорим про способы создания объектов. Рассмотрим самый простой исконно JavaScript'овый вариант – через фабрики. Потом изучим специальным образом оформленные функции – функции-конструкторы. А потом доберёмся до современных возможностей JS, а именно, до классов.

Функции-фабрики

```
function createUser(name) {
  return {
    name,
    isAdmin: false
  };
}

const user = createUser('Аркадий');
```

Добавление метода в объект

```
function createUser(name) {
  return {
    name,
    isAdmin: false
  };
}

const user = createUser('Аркадий');
```

+

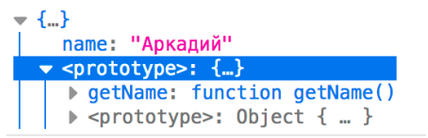
```
getName() {
  return `this.name${this.isAdmin ? ' (admin)' : ''}`;
}
```

Немного повспоминаем прошлую лекцию. Вот я хочу добавить метод getName в создаваемые фабрикой объекты. Как я могу это сделать?

- Просто объявить в возвращаемом объекте (много памяти)
- Создать прототип. Добавить метод в него.

Object.setPrototypeOf

```
const userProto = {
  getName() {
    return this.name;
  }
}

function createUser(name) {
  const user = { name };

  Object.setPrototypeOf(user, userProto);

  return user;
}
```



Итак, создаём объект userProto, в нём объявляем метод getName. В функции-фабрике используем Object.setPrototypeOf чтобы назначить ссылку на прототип для всех создаваемых пользователей. Таким образом мы не создаём каждым раз новый метод getName(), экономим ресурсы.

Кто-нибудь помнит, что на прошлой лекции Саша предупреждал об особенностях работы setPrototypeOf? JS движки выполняют различные внутренние оптимизации обращения к свойствам и методам и смена прототипа в цепочке приводит к деоптимизации кода. Поэтому лучше стараться избегать такого в коде.

Object.create (~7.5x faster)

```
const userProto = {
  getName() { ... }
}

function createUser(name) {
  return Object.create(userProto, {
   name: {
    value: name,
    writable: true,
    enumerable: true,
    configurable: true
   }
  });
}
```

Поэтому существует метод Object.create, Саша про него тоже говорил. Он позволяет создавать объекты с указанным прототипом. Такая реализация функции-фабрики приводит к такому же результату.

Функции-конструкторы

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

const user = new User('Аркадий');
```

- Пишутся с заглавной буквы
- Работают в паре с оператором new
- this ссылается на создаваемый объект

Другой вариант создания объектов – функции-конструкторы. Это обычные функции, для которых есть 2 соглашения:

1. Их названия пишут с большой буквы
2. Они не применяются без оператора new

Что происходит когда вызывается оператор new с функцией конструктором?

1. Создаётся новый пустой объект и он присваивается в this
2. Выполняется код функции-конструктора, обычно он модифицирует this и добавляет туда свойства
3. Возвращается объект this

Возвращать явно из конструктора ничего не нужно. Но если же вы это делаете, то будет работать так:

- При вызове return с объектом будет возвращён объект, а не this
- При вызове return с примитивом, примитив будет отброшен

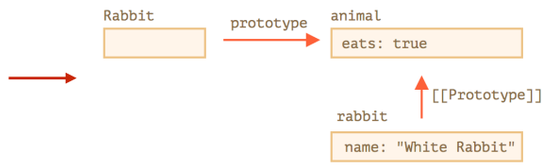
Методы в конструкторах

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

  this.sayHi = function() {
    console.log('Меня зовут: ' + this.name);
  }
}

const user = new User('Аркадий');

user.sayHi();
```

В функциях конструкторах тоже к объектам можно добавлять методы и всё та же проблема с производительностью и памятью сохранится.

prototype

```
let animal = {
  eats: true
}

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

Rabbit.prototype = animal;

const rabbit = new Rabbit('White Rabbit');

console.log(rabbit.eats);
```



Попробуем это починить. В функциях-фабриках мы это уже умеем делать.
Есть специальное свойство "prototype" у функции.
Оно имеет смысл только тогда, когда функция вызывается с оператором new.
Если в prototype содержится объект, то оператор new устанавливает его
в качестве прототипа для всех создаваемых функцией объектов.

prototype по умолчанию

```
function Rabbit() {}

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

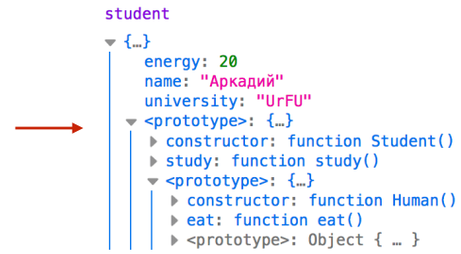
Лучше не перетирать prototype, а дополнять

```
function Rabbit() {}

Rabbit.prototype.jumps = true;
```

По умолчанию для любой функции уже выставлено свойство prototype и оно равно объекту со свойством constructor, которое указывает на саму функцию-конструктор. А что мы сделали в прошлом примере? Да, мы стёрли это свойство. Если мы разрабатываем библиотеку, то есть небольшая вероятность, что кто-то уже завязался в коде на эту особенность и мы всё сломали.

Наследование на прототипах

```
function Human(name, energy) {
  this.name = name;
  this.energy = energy;
}

Human.prototype.eat = function(amount) {
  this.energy += amount;
}

function Student(name, energy, university) {
  Human.call(this, name, energy);

  this.university = university;
}

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;
Student.prototype.study = function (hours) {
  this.energy -= hours;
}
```



Рассмотрим простую иерархию "классов". В ней есть Human и Student. В прототип Human мы складываем полезный метод "кушать", который прибавляет нам энергии. Студенту в конструкторе добавляем свойство university, чтобы можно было определить из какого он ВУЗа, а ещё ему нужно учиться, но учёба отнимает жизненные силы. Это запишем в метод study.
Теперь дело за малым, студенты тоже люди. Так об этом и скажем JavaScript'у. В свойство prototype мы записали новый объект, у которого внутреннее свойство Prototype указывает на прототип Human, восстановили ссылку на конструктор, а затем дополнили функцией study. Таким образом теперь мы можем у всех объектов созданных через Student вызывать методы study, eat, пользоваться свойствами name, energy, university.

Задачка

Написать собственную реализацию
метода bind для Function

Не повторять в production*

У меня есть для вас небольшая задачка. Вот мы сейчас узнали про свойство prototype. Теперь мы знаем как добавлять функциональность. Мы можем добавить в любую уже известную нам стандартную функцию-конструктор (Function, Array, Object, Map, Set, ...) новый метод или свойство. Попробуйте сами после лекции реализовать самостоятельно метод bind, вспомните как работать с контекстом в JavaScript. Присылайте в личку если напишете)

Конструкторы vs фабрики

Конструкторы

+ Привычность
- Неявное поведение
- Не работают асинхронно

Фабрики

+ Более гибкие
+ Явное поведение
+ this ведёт себя как обычно

Конструкторы и оператор new – самый часто применяемый подход во многих языках программирования и привычность этой конструкции не стоит недооценивать. Да, в JS есть неявное поведение, а именно речь про this, возвращаемое значение из конструктора, свойство prototype. Мы не можем вызвать new и подождать выполнения асинхронного кода. Так можно только с фабриками. Но про это расскажем на следующей неделе. Фабрики лишены перечисленных проблем, но не воспринимайте этот слайд как призыв писать только фабрики для конструирования объектов. По умолчанию используйте функции-конструкторы, а если нужна бОльшая гибкость и вариативность или асинхронное создание – используйте функции-фабрики.

Классы

```
class User {
  constructor(name) {
    this.name = name;
    this.isAdmin = false;
  }
}

const user = new User('Аркадий');
```

А теперь перейдём к современному JS. Мы только что насмотрелись на странные вещи – на функции-конструкторы. Так деды писали, но в современной промышленной разработке их применяют уже крайне редко. Теперь им на смену пришли классы.

```
» typeof User
← "function"
```

class, на самом деле, не является новой языковой сущностью. Если мы применим оператор typeof к классу, то получим function.
Что на самом деле делает конструкция class User { ... } ?
Создаёт функцию с названием User, код функции берётся из метода constructor, сохраняет все методы в User.prototype.

Отличия от функций-конструкторов

- Методы класса неперечислимы
- Конструктор класса не может быть вызван без new
- Код внутри класса всегда в строгом режиме ('use strict')
- У класса есть свойство [[FunctionKind]] со значением classConstructor

Генерация классов

```
function makeClass(phrase) {
  return class {
    sayHi() {
      console.log(phrase)
    }
  }
}

const User = makeClass('Привет')

new User().sayHi()
```

Поскольку классы – это функции, то мы автоматически приобретаем все их плюшки и, значит, можем генерировать классы на лету без существенного оверхэда.
Пригодится вам это не часто. Мы пару раз использовали в своём проекте чтобы написать классовые компоненты-декораторы на React, но практика не особо зашла.

Геттеры и сеттеры

```
class User {
  constructor(name) {
    this.name = name // Обращение к сеттеру
  }

  get name() {
    return this._name
  }

  set name(value) {
    if (value.length < 4) {
      console.error('Имя слишком короткое')

      return
    }

    this._name = value
  }
}

const user1 = new User('Иван')
console.log(user1.name)

const user2 = new User('Ким') // Имя слишком короткое
```



```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
})
```

Как и в других языках, в классах есть геттеры и сеттеры. Они нужны для того чтобы выполнять предобработку/постобработку данных или делать вычислимые поля. Под капотом геттер и сеттер для name объявляется примерно так как справа. Мы определяем свойство name в прототипе User.prototype, и у name вместо атрибутов enumerable, writable, configurable добавляем методы get() и set().

Свойства классов

```
class Rectangle {
  width;
  height = 0;

  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
}
```



```
function Rectangle(width, height) {
  _classCallCheck(this, Rectangle);
  _defineProperty(this, "width", undefined);
  _defineProperty(this, "height", 0);

  this.width = width;
  this.height = height;
}
```

На прошлых слайдах мы добавляли новые свойства в инстансы через модификацию this в конструкторе. В последних стандартах языка появился новый способ – через свойства класса.

Они, по правилам хорошего тона, описываются над конструктором и могут иметь дефолтное значение.

Описанный слева класс эквивалентен коду справа, где у нас функция-конструктор, которая проверяет, что её вызвали через new, затем добавляет свойства width и height с дефолтными значениями, а затем исполняет код конструктора.

«Защищённые» свойства и методы

```
class CoffeeMachine {
  _waterAmount = 0;

  setWaterAmount(value) {
    _checkWaterAmountValue(value);
    this._waterAmount = value;
  }

  getWaterAmount() {
    return this._waterAmount;
  }

  _checkWaterAmountValue(value) {
    if (value < 0) {
      throw new Error('Отрицательное количество воды');
    }
  }
}
```

Во всех книжках в которых пишут про то как надо писать хороший код единогласно пишут – нужно скрывать детали реализации, разделять внутренний и внешний интерфейсы. Как это делают в JS? Есть ли какая-то защита? До недавнего времени не было никакой. Было только соглашение между JS разработчиками, что они будут писать поля, к которым не надо обращаться извне, с подчёркиванием в начале. В самом языке никаких дополнительных проверок не было.

Приватные свойства и методы

```
class CoffeeMachine {
  #waterAmount = 0;

  setWaterAmount(value) {
    #checkWaterAmountValue(value);
    this.#waterAmount = value;
  }

  getWaterAmount() {
    return this.#waterAmount;
  }

  #checkWaterAmountValue(value) {
    if (value < 0) {
      throw new Error('Отрицательное количество воды');
    }
  }
}
```

Ещё нигде не работает*

Совсем недавно в язык добавили приватные методы. Вот так некрасиво с решёточкой их можно объявлять. На самом деле пока нельзя, это ещё нигде не работает. К приватным полям и методам можно обращаться только внутри класса, они не наследуются.

Статические свойства и методы

```
class Article {
  static counter = 0;

  constructor(title, date) {
    this.title = title;
    this.date = date;

    Article.counter++;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}
```

Мы можем присвоить свойство или метод самому классу, а не его prototype. Такие свойства и методы называются статическими. Они доступны для обращения по имени класса.

Фабрики на статических методах

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static createTodays() {
    return new this("Сегодняшний дайджест", new Date());
  }
}
```

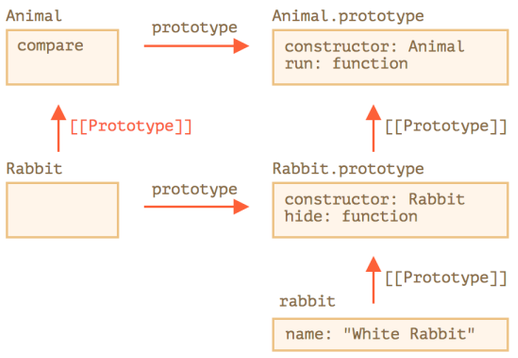
this === Article

В какой-то момент вам станет может станет мало конструктора и снова понадобится бОльшая гибкость при создании инстансов класса. Тогда вы можете прибегнуть к статическому методу-фабрике и сконструировать инстанс в ней.

Замечу, что this в статических методах указывает на сам класс.

Наследование

```
class Animal {
  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    console.log(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}

class Rabbit extends Animal {
  hide() {
    console.log(`${this.name} прячется!`);
  }
}

let rabbits = [
  new Rabbit('Белый кролик', 10),
  new Rabbit('Чёрный кролик', 5)
];

rabbits.sort(Rabbit.compare);

rabbits[0].run();
```



На наследование через функции-конструкторы мы уже посмотрели. Теперь давайте глянем как это будет выглядеть на классах. Рассмотрим на примере Animal и Rabbit.
Rabbit унаследуем от Animal с помощью конструкции Rabbit extends Animal. Что делает эта конструкция?

1. Прототипно наследует функцию Rabbit от функции Animal
2. Прототипно наследует Rabbit.prototype от Animal.prototype

В итоге наследование работает как для статических свойств и методов, так и для свойств и методов инстансов. Мы можем вызвать compare у Rabbit, хотя метод определён в Animal.

Вообще, синтаксис extends позволяет справа указать не только класс, но и вообще любое другое выражение. Редко используемая фича, но может пригодиться для продвинутых приёмов проектирования.

Переопределение методов

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed += speed;
    console.log(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    console.log(`${this.name} стоит.`);
  }
}

class Rabbit extends Animal {
  constructor(...args) {
    super(...args);
  }

  hide() {
    console.log(`${this.name} прячется!`);
  }

  stop() {
    super.stop();
    this.hide();
  }
}

const rabbit = new Rabbit('Белый кролик');

rabbit.run(5);
rabbit.stop();
```

Методы в классах наследниках можно переопределять. Допустим, мы здесь переопределяем метод stop(). На этом слайде хотелось бы заострить внимание на super. Это ключевое слово, которое позволяет обращаться к методам и свойствам родителя. Конструкторы тоже можно переопределять. Тут есть небольшая специфичная особенность. В конструкторе наследников обязательно нужно вызывать конструктор родительского класса. На скрине так называемый дефолтный конструктор, который просто проксирует все аргументы.

instanceof

```
class Rabbit {}

const rabbit = new Rabbit();

rabbit instanceof Rabbit // true
```

```
function Rabbit() {}

const rabbit = new Rabbit();

rabbit instanceof Rabbit;
```

Оператор instanceof позволяет проверить к какому классу принадлежит объект с учётом наследования. Это работает как для объект созданных с помощью классов, так и с помощью функций-конструкторов.

Алгоритм работы оператора довольно прост. Он пробегает вглубину по ссылкам на прототипы и сравнивает их, а затем возвращает true или false.

Яндекс

Спасибо

Евгений Марков

Разработчик интерфейсов