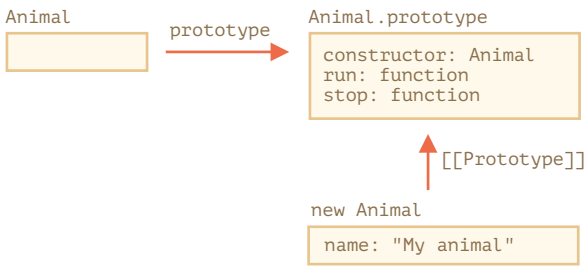
Наследование классов

Допустим, у нас есть два класса.

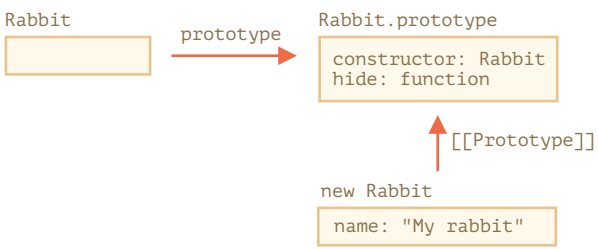
`Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
  }
}

let animal = new Animal("Мой питомец");
```



…И `Rabbit`:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbit = new Rabbit("Мой кролик");
```



Сейчас они полностью независимы.

Но мы хотим, чтобы `Rabbit` расширял `Animal`. Другими словами, кролики должны происходить от животных, т.е. иметь доступ к методам `Animal` и расширять функциональность `Animal` своими методами.

Редактировать на GitHub

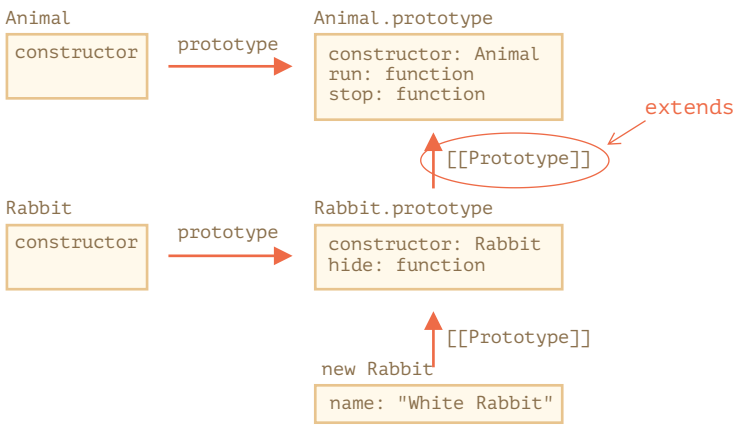
Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово `"extends"` и указать название родительского класса перед `{..}`.

Ниже `Rabbit` наследует от `Animal`:

```
1  class Animal {
2    constructor(name) {
3      this.speed = 0;
4      this.name = name;
5    }
6    run(speed) {
7      this.speed = speed;
8      alert(`${this.name} бежит со скоростью ${this.speed
9    }
10   stop() {
11     this.speed = 0;
12     alert(`${this.name} стоит.`);
13   }
14 }
15
16 // Наследуем от Animal указывая "extends Animal"
17 class Rabbit extends Animal {
18   hide() {
19     alert(`${this.name} прячется!`);
20   }
21 }
22
23 let rabbit = new Rabbit("Белый кролик");
24
25 rabbit.run(5); // Белый кролик бежит со скоростью 5.
26 rabbit.hide(); // Белый кролик прячется!
```

Теперь код `Rabbit` стал короче, так как используется конструктор класса `Animal` по умолчанию и кролик может использовать метод `run` как и все животные.

Ключевое слово `extends` работает, используя прототипы. Оно устанавливает `Rabbit.prototype.[[Prototype]]` в `Animal.prototype`. Так что если метод не найден в `Rabbit.prototype`, JavaScript берёт его из `Animal.prototype`.



Как мы помним из главы Встроенные прототипы, в JavaScript используется наследование на прототипах для встроенных объектов. Например `Date.prototype.[[Prototype]]` это `Object.prototype`, поэтому у дат есть универсальные методы объекта.

После `extends` разрешены любые выражения

Синтаксис создания класса допускает указывать после `extends` не только класс, но любое выражение.

Пример вызова функции, которая генерирует родительский класс:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Привет") {}

new User().sayHi(); // Привет
```

Здесь `class User` наследует от результата вызова `f("Привет")`.

Это может быть полезно для продвинутых приёмов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

Переопределение методов

Давайте пойдём дальше и переопределим метод. Сейчас `Rabbit` наследует от `Animal` метод `stop`, который устанавливает `this.speed = 0`.

Если мы определим свой метод `stop` в классе `Rabbit`, то он будет использоваться взамен родительского:

```
class Rabbit extends Animal {
  stop() {
    // ...будет использован для rabbit.stop()
  }
}
```

...Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово `"super"` для таких случаев.

- `super.method(...)` вызывает родительский метод.
- `super(...)` вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

```
class Animal {

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

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

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

```

```
}

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

  stop() {
    super.stop(); // вызываем родительский метод stop
    this.hide(); // и затем hide
  }
}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.stop(); // Белый кролик стоит. Белый кролик пряч
```

Теперь у класса `Rabbit` есть метод `stop`, который вызывает родительский `super.stop()` в процессе выполнения.

У стрелочных функций нет `super`

Как упоминалось в главе Повторяем стрелочные функции, стрелочные функции не имеют `super`.

При обращении к `super` стрелочной функции он берётся из внешней функции:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // вызыв
  }
}
```

В примере `super` в стрелочной функции тот же самый, что и в `stop()`, поэтому метод отрабатывает как и ожидается. Если бы мы указали здесь «обычную» функцию, была бы ошибка:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Переопределение конструктора

С конструкторами немного сложнее.

До сих пор у `Rabbit` не было своего конструктора.

Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор:

```
class Rabbit extends Animal {
  // генерируется для классов-потомков, у которых нет с
  constructor(...args) {
    super(...args);
  }
}
```

Как мы видим, он просто вызывает конструктор родительского класса. Так будет происходить, пока мы не создадим собственный конструктор.

Давайте добавим конструктор для `Rabbit`. Он будет устанавливать `earLength` в дополнение к `name`:

```
class Animal {
```

Поделиться

Редактировать на GitHub

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

class Rabbit extends Animal {

  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }

  // ...
}

// Не работает!
let rabbit = new Rabbit("Белый кролик", 10); // Error:
```

Упс! При создании кролика – ошибка! Что не так?

Если коротко, то в классах-потомках конструктор обязан вызывать `super(...)`, и (!) делать это перед использованием `this`.

...Но почему? Что происходит? Это требование кажется довольно странным.

Конечно, всему есть объяснение. Давайте углубимся в детали, чтобы вы действительно поняли, что происходит.

В JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством `[[ConstructorKind]]:"derived"`.

Разница в следующем:

- Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его `this`.
- Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если мы создаём собственный конструктор, мы должны вызвать `super`, в противном случае объект для `this` не будет создан, и мы получим ошибку.

Чтобы конструктор `Rabbit` работал, он должен вызвать `super()` до того, как использовать `this`, чтобы не было ошибки:

```
class Animal {

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

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

```

```
22
23  // теперь работает
24  let rabbit = new Rabbit("Белый кролик", 10);
    alert(rabbit.name); // Белый кролик
    alert(rabbit.earLength); // 10
```

Устройство super, [[HomeObject]]

> **Продвинутая информация**
>
> Если вы читаете учебник первый раз – эту секцию можно пропустить.
>
> Она рассказывает о внутреннем устройстве наследования и вызов `super`.

Давайте заглянем «под капот» `super`. Здесь есть некоторые интересные моменты.

Вообще, исходя из наших знаний до этого момента, `super` вообще не может работать!

Ну правда, давайте спросим себя – как он должен работать, чисто технически? Когда метод объекта выполняется, он получает текущий объект как `this`. Если мы вызываем `super.method()`, то движку необходимо получить `method` из прототипа текущего объекта. И как ему это сделать?

Задача может показаться простой, но это не так. Движок знает текущий `this` и мог бы попытаться получить родительский метод как `this.__proto__.method`. Однако, увы, такой «наивный» путь не работает.

Продемонстрируем проблему. Без классов, используя простые объекты для наглядности.

Вы можете пропустить эту часть и перейти ниже к подсекции `[[HomeObject]]`, если не хотите знать детали. Вреда не будет. Или читайте далее, если хотите разобраться.

В примере ниже `rabbit.__proto__ = animal`. Попробуем в `rabbit.eat()` вызвать `animal.eat()`, используя `this.__proto__`:

```
1  let animal = {
2    name: "Animal",
3    eat() {
4      alert(`${this.name} ест.`);
5    }
6  };
7
8  let rabbit = {
9    __proto__: animal,
10   name: "Кролик",
11   eat() {
12     // вот как предположительно может работать super.ea
13     this.__proto__.eat.call(this); // (*)
14   }
15 };
16
17 rabbit.eat(); // Кролик ест.
```

В строке `(*)` мы берём `eat` из прототипа (`animal`) и вызываем его в контексте текущего объекта. Обратите внимание, что `.call(this)` здесь неспроста: простой вызов `this.__proto__.eat()` будет выполнять родительский `eat` в контексте прототипа, а не текущего объекта.

Приведённый выше код работает так, как задумано: выполняется нужный `alert`.

Теперь давайте добавим ещё один объект в цепочку наследования и увидим, как все сломается:

```
1  let animal = {
```

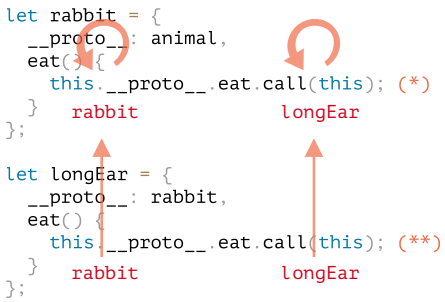
```
  name: "Животное",
  eat() {
    alert(`${this.name} ест.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...делаем что-то специфичное для кролика и вызыв
    this.__proto__.eat.call(this); // (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    // ...делаем что-то, связанное с длинными ушами и в
    this.__proto__.eat.call(this); // (**)
  }
};

longEar.eat(); // Error: Maximum call stack size exceed
```

Теперь код не работает! Ошибка возникает при попытке вызова `longEar.eat()`.

На первый взгляд все не так очевидно, но если мы проследим вызов `longEar.eat()`, то сможем понять причину ошибки. В обеих строках `(*)` и `(**)` значение `this` – это текущий объект (`longEar`). Это важно: для всех методов объекта `this` указывает на текущий объект, а не на прототип или что-то ещё.

Итак, в обеих линиях `(*)` и `(**)` значение `this.__proto__` одно и то же: `rabbit`. В обоих случаях метод `rabbit.eat` вызывается в бесконечном цикле не поднимаясь по цепочке вызовов.

Картина того, что происходит:



1. Внутри `longEar.eat()` строка `(**)` вызывает `rabbit.eat` со значением `this=longEar`.

```
// внутри longEar.eat() у нас this = longEar
this.__proto__.eat.call(this) // (**)
// становится
longEar.__proto__.eat.call(this)
// то же что и
rabbit.eat.call(this);
```

2. В строке `(*)` в `rabbit.eat` мы хотим передать вызов выше по цепочке, но `this=longEar`, поэтому `this.__proto__.eat` снова равен `rabbit.eat`!

```
// внутри rabbit.eat() у нас также this = longEar
this.__proto__.eat.call(this) // (*)
// становится
longEar.__proto__.eat.call(this)
```

```
// или (снова)
rabbit.eat.call(this);
```


3. … `rabbit.eat` вызывает себя в бесконечном цикле, потому что не может подняться дальше по цепочке.

Проблема не может быть решена с помощью одного только `this`.

[[HomeObject]]

Для решения этой проблемы в JavaScript было добавлено специальное внутреннее свойство для функций: `[[HomeObject]]`.

Когда функция объявлена как метод внутри класса или объекта, её свойство `[[HomeObject]]` становится равно этому объекту.

Затем `super` использует его, чтобы получить прототип родителя и его методы.

Давайте посмотрим, как это работает – опять же, используя простые объекты:

```
let animal = {
  name: "Животное",
  eat() {         // animal.eat.[[HomeObject]] == anima
    alert(`${this.name} ест.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Кролик",
  eat() {         // rabbit.eat.[[HomeObject]] == rabbi
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Длинноух",
  eat() {         // longEar.eat.[[HomeObject]] == long
    super.eat();
  }
};

// работает верно
longEar.eat();  // Длинноух ест.
```

Это работает как задумано благодаря `[[HomeObject]]`. Метод, такой как `longEar.eat`, знает свой `[[HomeObject]]` и получает метод родителя из его прототипа. Вообще без использования `this`.

Методы не «свободны»

До этого мы неоднократно видели, что функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым `this`.

Но само существование `[[HomeObject]]` нарушает этот принцип, так как методы запоминают свои объекты. `[[HomeObject]]` нельзя изменить, эта связь – навсегда.

Единственное место в языке, где используется `[[HomeObject]]` – это `super`. Поэтому если метод не использует `super`, то мы все ещё можем считать его свободным и копировать между объектами. А вот если `super` в коде есть, то возможны побочные эффекты.

Вот пример неверного результата `super` после копирования:

```
let animal = {
```

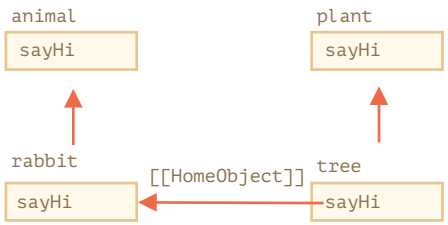
```
  sayHi() {
    console.log("Я животное");
  }
};

// rabbit наследует от animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    console.log("Я растение");
  }
};

// tree наследует от plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

tree.sayHi();  // Я животное (?!?)
```

Вызов `tree.sayHi()` показывает «Я животное». Определённо неверно.

Причина проста:

- В строке `(*)`, метод `tree.sayHi` скопирован из `rabbit`. Возможно, мы хотели избежать дублирования кода?
- Его `[[HomeObject]]` – это `rabbit`, ведь он был создан в `rabbit`. Свойство `[[HomeObject]]` никогда не меняется.
- В коде `tree.sayHi()` есть вызов `super.sayHi()`. Он идёт вверх от `rabbit` и берёт метод из `animal`.

Вот диаграмма происходящего:



Методы, а не свойства-функции

Свойство `[[HomeObject]]` определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как `method()`, а не `"method: function()"`.

Для нас различий нет, но они есть для JavaScript.

В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет `[[HomeObject]]`, и наследование не работает:

```
let animal = {
  eat: function() { // намеренно пишем так, а не eat()
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
```

```
10      super.eat();
11    }
12  };
13
14  rabbit.eat();  // Ошибка вызова super (потому что нет [
```

Итого

1. Чтобы унаследовать от класса: `class Child extends Parent`:
   - При этом `Child.prototype.__proto__` будет равен `Parent.prototype`, так что методы будут унаследованы.
2. При переопределении конструктора:
   - Обязателен вызов конструктора родителя `super()` в конструкторе `Child` до обращения к `this`.
3. При переопределении другого метода:
   - Мы можем вызвать `super.method()` в методе `Child` для обращения к методу родителя `Parent`.
4. Внутренние детали:
   - Методы запоминают свой объект во внутреннем свойстве `[[HomeObject]]`. Благодаря этому работает `super`, он в его прототипе ищет родительские методы.
   - Поэтому копировать метод, использующий `super`, между разными объектами небезопасно.

Также:

- У функций-стрелок нет своего `this` и `super`, поэтому они «прозрачно» встраиваются во внешний контекст.

Задачи

Ошибка создания экземпляра класса

важность: 5

В коде ниже класс `Rabbit` наследует `Animal`.

К сожалению, объект класса `Rabbit` не создаётся. Что не так? Исправьте ошибку.

```
1  class Animal {
2
3    constructor(name) {
4      this.name = name;
5    }
6
7  }
8
9  class Rabbit extends Animal {
10   constructor(name) {
11     this.name = name;
12     this.created = Date.now();
13   }
14 }
15
16 let rabbit = new Rabbit("Белый кролик"); // Error: this
17 alert(rabbit.name);
```

решение

Улучшенные часы

важность: 5

У нас есть класс `Clock`. Сейчас он выводит время каждую секунду

Поделиться

Редактировать на GitHub

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();

    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);

    console.log(output);
  }

  stop() {
    clearInterval(this.timer);
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(), 1000)
  }
}
```

Создайте новый класс `ExtendedClock`, который будет наследоваться от `Clock` и добавьте параметр `precision` – количество миллисекунд между «тиками». Установите значение в `1000` (1 секунда) по умолчанию.

- Сохраните ваш код в файл `extended-clock.js`
- Не изменяйте класс `clock.js`. Расширьте его.

Открыть песочницу для задачи.

решение

Проводим курсы по JavaScript и фреймворкам.

Комментарии

перед тем как писать...

© 2007—2020 Илья Кантор | о проекте | связаться с нами | пользовательское соглашение | политика конфи