Обзор на Прототипно-базираното ООП и сравнение с "традиционното" ООП. Примери в JS. Какво е указателят this в JS?

Екип 5 - Велислава, Веселин, Никола и Петър-Габриел

План

- "Традиционно" ООП
- Прототипно-базирано ООП
- Prototype chaining
- this

Какво е “традиционното” ООП?

- Абстракция
- Капсулация
- Наследяване
- Полиморфизъм

```cpp
#include <iostream>

// Пример за дефиниране на клас в C++
class Rectangle {
private:
    int width;
    int height;

public:
    Rectangle(int w, int h) : width(w), height(h) {}

    int calculateArea() {
        return width * height;
    }
};

// Пример за създаване на обект от класа Rectangle
int main() {
    Rectangle myRectangle(5, 10);
    std::cout << "Area of rectangle: " << myRectangle.calculateArea() << std::endl;
}
```

```cpp
#include <iostream>

class Shape {
public:
    void setColor(std::string color) {
        this->color = color;
    }
    std::string getColor() {
        return color;
    }
protected:
    std::string color;
};

class Circle : public Shape {
public:
    void setRadius(double radius) {
        this->radius = radius;
    }
    double getRadius() {
        return radius;
    }
private:
    double radius;
};

int main() {
    Circle myCircle;
    myCircle.setColor("Red");
    myCircle.setRadius(5.0);

    std::cout << "Color: " << myCircle.getColor() << std::endl;
    std::cout << "Radius: " << myCircle.getRadius() << std::endl;

    return 0;
}
```

```javascript
class Shape {
  constructor(color) {
    this.color = color;
  }

  setColor(color) {
    this.color = color;
  }

  getColor() {
    return this.color;
  }
}

class Circle extends Shape {
  constructor(color, radius) {
    super(color);
    this.radius = radius;
  }

  setRadius(radius) {
    this.radius = radius;
  }

  getRadius() {
    return this.radius;
  }
}

let myCircle = new Circle('Red', 5.0);
console.log("Color: " + myCircle.getColor());
console.log("Radius: " + myCircle.getRadius());
```

Прототипно-базирано ООП

Прототип?

- Първоначален модел на даден продукт

- Дава базова функционалност, която, в последствие, може да бъде подобрена и/или надградена

Прототипи в JS (общо казано)

- Всеки обект има прототип
- Нов обект може да бъде създаден използвайки съществуващ прототип
- На всеки обект може индивидуално да бъде добавяна функционалност

Как се създават обекти?

- Чрез използването на литерал - {}, което използва конструктор ф-ята на Object

- Използване на персонализирана конструктор функция

- Извикването и с ключовата дума “new” създава обект

- Всяка (конструктор) функция има прототип

- Инициализираме празен обект:

```
1   let myObj = {}
```

- Създава се от конструктор функцията на Object и има неговия прототип:

- Инициализираме обект с функцията Person:

```
1   function Person(name) {
2       this.name = name;
3   }
4
5   let john = new Person("John");
```

```
john: Person {name: 'John'}
  name: 'John'
  [[Prototype]]: Object
    constructor: ƒ Person(name) {\r\n    this.name = name;\r\n}
    [[Prototype]]: Object
```

Прототип на функция

- Всяка функция има прототипен обект

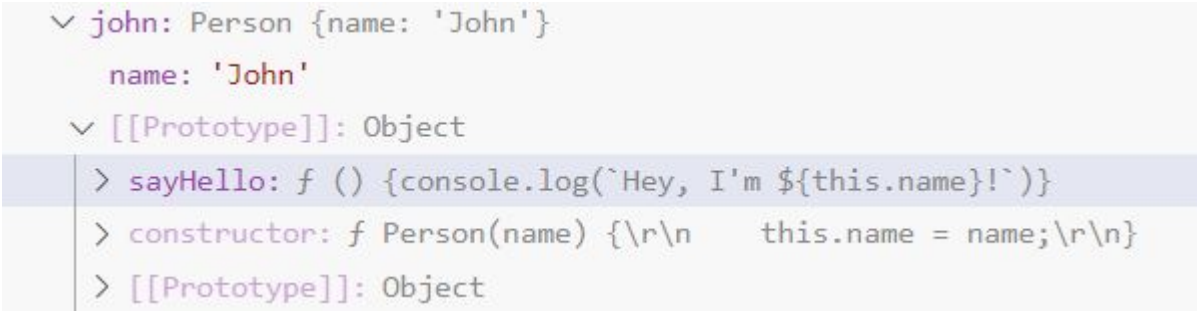
- Този обект първоначално е празен, а след това приема свойства и методи добавени към него

- Ключов аспект в прототипното наследяване

- Използваме свойството .prototype за да добавим метод към пртотипа на Person:

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayHello = function() {console.log(`Hey, I'm ${this.name}!`)}

let john = new Person("John");

john.sayHello() // Hey, I'm John!
```

- Става част от прототип обекта, а не конкретно “john”:



Прототипните свойства са споделени

- Когато свойство се добави към прототипа на дадена функция, то за него се заделя памет само веднъж и всеки обект сочи към това място в паметта

- На пръв поглед би изглеждало сякаш наподобява статично свойство, но това не е така

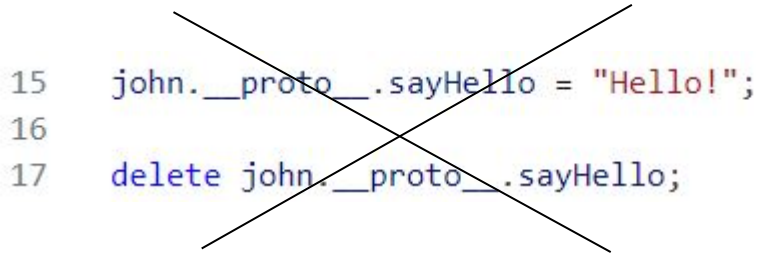
- Прототипните функции имат достъп до контекста на даден обект и могат да бъдат достъпвани чрез него

- В този пример създаваме още един обект използвайки конструктора Person и когато извикаме sayHello() чрез него, прототипната функция успява да достъпи неговия контекст:

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayHello = function() {console.log(`Hey, I'm ${this.name}!`)}

let john = new Person("John");
let peter = new Person("Peter")

john.sayHello() // Hey, I'm John!
peter.sayHello() // Hey, I'm Peter!
```

__proto__

- Въпреки, че имаме възможността да променяме самия прототип през даден обект, това обикновено не е препоръчително
- Идеята е това да бъдат споделени свойства, които да бъдат еднакви за всички класове и тяхната промяна да се случва централизирано през конструктор функцията



- Better to use:

  Object.getPrototypeOf() and Object.setPrototypeOf()

- Какво се случва, ако се опитаме да променим прототипно свойство директно?

```
5   Person.prototype.sayHello = function() {console.log(`Hey, I'm ${this.name}!`)}
6
7   let john = new Person("John");
8   let peter = new Person("Peter")
9
10  john.sayHello = function() {console.log('Hey, my name is John!')}
```

- Създава се ново свойство на конкретния обект:

```
▼ john: Person {name: 'John', sayHello: ƒ}
    name: 'John'
  ▶ sayHello: ƒ () {console.log('Hey, my name is John!')}
  ▼ [[Prototype]]: Object
    ▶ sayHello: ƒ () {console.log(`Hey, I'm ${this.name}!`)}
    ▶ constructor: ƒ Person(name) {\r\n    this.name = name;\r\n}
    ▶ [[Prototype]]: Object
```

Прототипни вериги

this в JavaScript

Благодарим ви за вниманието!