Объектно-ориенттированное программирование в Java Script

Определение класса

```
class Task {

}
console.log(typeof Task); // function

const course = new Task();
console.log(typeof Task); // object
```

оператор instanceof проверяет принадлежит ли объект данному классу

```
console.log(course instanceof Task); // true
```

Конструктор класса

конструктор вызывается когда мы создаем объект(instance)
с помощью ключевого слова new

```
class Course {
    constructor(id, title){
     this.id = id;
     this.title = title;
     }
}

const course = new Course(1, 'JavaScript');
console.log(course.title);  //  JavaScript
```

Класс содержит свойства и методы
Свойства как правило определяются в конструкторе класса

```
class Person {
    constructor(name){
     this.name = name;
    }
}
```

Доступ к свойствам и методам внутри класса осуществляется через ключевое свойство this

```
this.имя_свойства
this.имя_метода()
```

Доступ к свойствам и методам объекта осуществляется через имя объекта

```
let user = new Person("Hary");
console.log(user.name);
```

Статические свойства класса

Это свойства которые не принадлежат экземпляру объекта. То есть если создать несколько объектов от одного класса, статические свойства и методы будут общими для все этих объектов.

```
class Course {
    constructor(title){
    this.title = title;
    }
}
Course.duration = 45;

let c = new Course('JavaScript');

c.duration; // error - так нельзя вызывать статическое свойство

console.log(Course.duration);  //  45
```

!!! Статические свойства и методы вызываются относительно имени класса

Методы

```
class User {
    constructor(name){
    this.name = name;
    }
    greet(){
    console.log(`Hello ${this.name} `);
    }
}

const user1 = new User('Bill');
const user2 = new User('Tom');

user1.greet();
user2.greet();
```

Статические методы

- принадлежат не объекту а классу;
- вызываются относительно имени класса;
- в static методах недоступна переменная this

```
class Course {
    constructor(title){
        this.title = title;
    }

        static getCompanyInfo(){
            console.log('Design-class');
    }
}
```

```
const php = new Course('PHP');

php.getCompanyInfo();   // Error
console.log(Course.getCompanyInfo());
```

!!! в static методах недоступна переменная this

Наследование классов

```
class Vehicle { }

class Drone extends Vihicle {}
class Car extends Vihicle {}
```

Родительский класс -> super класс
Дочерний класс -> подкласс

Все свойства и методы родительского класса наследуются дочерним классом

```
let c = new Car();

console.log(c instanceof Car); // true

console.log(c instanceof Vehicle);   // true

console.log(c instanceof Object);   // true
```

Конструктор подкласса

```
class Vehicle {
    constructor(){
        console.log("Vehicle constructing");
    }
}

class Car extends Vehicle {}

const c = new Car(); // Vehicle constructing
```

То есть при создании объекта от подкласса в котором *не определен конструктор*, вызывается конструктор super класса

Но если мы определяем в подклассе конструктор, нужно обязательно в нем вызывать конструктор super класса

```
class Vehicle {
   constructor(){
       console.log("Vehicle constructing");
   }
}

class Car extends Vehicle {
   constructor(){
      super();
      console.log("Car constructing");
  }
}

const c = new Car();  // Vehicle constructing
                      // Car constructing
```

В наследуемый конструктор должны передаваться все свойства, которые объявлены в родительском конструкторе

```
class Vehicle {
   constructor(id){
       this.id = id;
   }
}

class Car extends Vehicle {
    constructor(id){
        super(id);
   }
}

const c = new Car('A111');
console.log(c.id); // A111
```

Наследование свойств

```
class Vehicle {
    constructor(){
        this.gpsEnabled = true;
    }
}

class Car extends Vehicle {
    constructor(){
    super();
   }
}

const c = new Car();
console.log(c.gpsEnabled)); // true
```

Как уже говорилось, все свойства и методы super класса наследуются в подклассе

Переопределение свойств

В подклассе можно переопределить методы и свойства super класса

```
class Vehicle {
    constructor(){
        this.gpsEnabled = true;
    }
}

class Car extends Vehicle {
    constructor(){
       super();
       this.gpsEnabled = false;
}

const c = new Car();
console.log( c.gpsEnabled ) ); // false
```

Наследование и переопределение методов

```
class Vehicle {
    start() {
        console.log('Vehicle is starting');
    }
}

class Car extends Vehicle {
      start() {
          super.start();
          console.log('Car is starting');
      }
}

const c = new Car();
c.start(); // Car is starting
```

Наследование и переопределение статических методов

```
class Vehicle {
    start() {
        console.log('Vehicle is starting');
    }
    static getCompanyInfo() {
        console.log('Desing-class');
    }
}

class Car extends Vehicle {
     static getCompanyInfo() {
         super.getCompanyInfo();
         console.log('Desing-class again');
     }
}

let c = new Car();
c.start(); // Vehicle is starting

Car.getCompanyInfo(); // Design-class, Design-class again
```

Обработка ошибок

Например если написать

```
10 = 'string'
```

то получим ошибку

`Uncaught ReferenceError: - это называется исключением`

То есть произошла исключительная ситуация при выполнении кода, то есть ошибка

Тип `ReferenceError` - наследуется от объекта Error

Мы сами можем создавать исключения с помощью ключевого слова `throw` которое прерывает выполнение скрипта и заставляет интерпритатор искать ближайшую ветку `catch`, например

```
function test(n) {
    if(n > 5) {
     throw new Error('message about error')
    }
    return n + 10;
}
test(10);
console.log("продолжение кода");
```

Исключения можно перехватывать с помощью конструкции

```
try {

  //код который может вызвать исключение

} catch(err) {

    //  код который выполняется при возникновении  исключения в
        блоке try

} finally {

 // код который выполняется вне зависимости от того  было ли
    исключение в блоке try

}
```

Например

```
try {
      console.log("statr try");
      nonexists;
      console.log("end try");

} catch(err) {
    console.log("Error !!!", err);
} finally {
    console.log("finaly code");
}

console.log("Continue ...");
```

```
statr try

"Error !!!"
[object Error] { ... }

"finaly code"

"Continue execution code"
```

```
let response = '{"age": 22}';

try {
    let user = JSON.parse(response);
    if(!user.name) {
        throw new Error("no name");
    }
     console.log(user.name);
} catch(err) {
    console.log("Error !!!", err.message);
}
console.log("Continue execution code");
```

```
Error !!! no name
Continue execution
```

Functional Programming – это

1. Декларативное программирование
2. Не допускаются side-effects
3. Data immutable -> данные иммутабельные (то есть неизменяемые)

В JavaScript функции являбтся First-class object, или еще их называют Hight order function - функции высшего порядка

То есть функции

- можно присваивать переменной
- возвращать из функций
- передавать как аргумент в функции

Например

```
const add = (x,y) => x + y;

const log = fn =>  (...args) =>  {
    console.log(...args);
    return fn(...args);
};

const logAdd = log(add);
console.log(logAdd(1,2));
```

Функциональное программирование предусматривает, что функции
- pure function
- immutability - не меняют входящие данные
- currying
- composition

Pure function

```
const add = (x,y) => x + y;
```

Сколько не запускать эту функцию с одинаковыми аргументами, всегда будем получать один и тот же результат (predictable = предсказуемый)

А вот это не чистая функция

```
let x = 0;
const add = y =>  (x = x + y);
```

Immutability - функция не меняет входящие данные, а на основе входящих создает новые и возвращает их

Здесь меняются данные

```
let add = arr => arr.push(3);
const myArr = [1,2];

add(myArr);
console.log(myArr);
// [1, 2, 3]


add(myArr);
console.log(myArr);
// [1, 2, 3,3]
```

Здесь НЕ меняются данные

```
let add = arr => arr.concat(3);
const myArr = [1,2];

let result;
result = add(myArr);
console.log(result);
// [1,2,3]

result = add(myArr);
console.log(result);
// [1,2,3]
```

Currying => преобразование функции, принимающей несколько аргументов в набор функций, каждая из которых является функцией, принимающий один аргумент

Функцию

```
const add = (x,y) => x + y;
```

перезапишем как

```
const add = x => y => x + y;

const f = add(3);

console.log(f(2));
console.log(f(3));
```

Composition => функции могут быть комбинироваться для создания новых функций.

```
const add = (x,y) => x + y;
const square = x => x * x;

const addAndSquare  = (x,y) => square(add(x,y))
```