



ООП в JavaScript

1. Ключевое слово «this»

Если методу объекта внутри требуется доступ к информации того объекта в котором он находится, метод может использовать **ключевое слово *this***.

Значение ***this*** – это объект «перед точкой», который используется для вызова метода.

значением *this* будет *user* (ссылка на объект *user*).

```
let user = {  
  name: "John",  
  age: 30,  
  sayHi() {  
    // "this" - это "текущий объект".  
    alert(this.name); // user.name  
  }  
};  
user.sayHi(); // выполнение метода
```

```
/*sayHi() {  
  alert(user.name);  
}*/  
let admin = user;  
user = null; // перезапишем  
переменную для наглядности,  
теперь она не хранит ссылку  
на объект.  
  
admin.sayHi(); // TypeError
```

2. FunctionDeclaration vs FunctionExpression

Function Declaration (Объявление функции):

```
function sayHi() {  
  alert("Hello");  
}
```

Имя функции
(переменная)

Function Expression (Функциональное выражение):

```
let sayHi = function () {  
  alert("Hello");  
};
```

переменная

т.к. ф-ция - часть выражения

Как бы не создавалась функция – она является ЗНАЧЕНИЕМ, представляющим действие. Это особое значение, т.к. мы можем с его помощью вызвать выполнение ф-ции или вывести её код.

```
alert ( sayHi ); // вывод кода функции  
alert ( sayHi() ); // вызов функции
```

**Function Declaration* – может быть вызвана раньше, чем объявлена т.к. обрабатывается «движком» перед выполнением всех инструкций, но при этом имеет область видимости блока { . . . } в котором находится.

```
function sayHi() { // создали  
  alert("Hello");  
}
```

```
let func = sayHi; // копируем в перем.  
let val = sayHi(); // запись результата в val  
func(); // вызываем копию  
sayHi(); // вызываем оригинал
```

3. Объектная ориентация

JS является **не стандартным** объектно-ориентированным языком. *JS* основан на прототипах (унаследованные классы не наследуются прямо от базового класса, а создаются путём клонирования базового класса, являющегося прототипом). Это очень удобно использовать для реализации 3-х парадигм ООП, создавая тем самым ощущение ООП.

Объекты *JS* поддерживают инкапсуляцию и наследование, и полиморфизм с помощью свойства **prototype**

ООП в *JS* имеет ряд преимуществ. Являясь интерпретируемым языком *JS* не требует: объявление методов и полей класса в конструкторе как другие ЯП с поддержкой ООП (C++). Члены класса могут быть добавлены к классу в любой момент времени, поля класса не обязаны иметь фиксированный тип данных и могут менять его в любое время.

Обычный синтаксис создания «литерального» объекта {...} позволяет создать **только один объект**. Но часто необходимо создать множество похожих, однотипных объектов. Это можно сделать при помощи **функции-конструктора и оператора "new"**.

4. Создание класса на JS

Чтобы создать объекты по некоторому «шаблону» при помощи оператора **new** в JS, надо объявить **функцию-конструктор** (класс) :

```
..  
function Point() {  
}  
..
```

Объект класса Point
создается с помощью
оператора new:

```
let c = new Point();
```

Для добавления полей к классу используется оператор **this**, за которым следует имя поля. Напомним, методы и поля могут создаваться **везде** в JS, а не только в теле **функции-конструктора**.

```
function Point(x, y) {  
  this.m_x = x;  
  this.m_y = y;  
}
```

Обращение к
полям объекта
через переменную


```
let p = new Point(5, 3);  
alert(p.m_x);
```

Функции-конструкторы технически являются обычными функциями. Но есть соглашения:

1. Имя функции-конструктора должно начинаться с **заглавной буквы**.
2. Функция-конструктор должна выполняться **только** с помощью оператора **"new"**.

5. «Под капотом» функции-конструктора

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
} //; не ставится  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```



```
function User(name) {  
  // this = {}; (неявно)  
  
  // добавляет свойства к this  
  this.name = name;  
  this.isAdmin = false;  
  
  // return this; (неявно)  
}
```

Когда функция вызывается как `new Point(...)`, происходит следующее:

1. Создаётся новый **пустой объект**, и он присваивается `this`.
2. Выполняется тело функции. Обычно оно модифицирует `this`, добавляя туда новые св-ва.
3. Возвращается значение `this`.

```
let user = new User("Jack");
```

=

```
let user = {  
  name: "Jack",  
  isAdmin: false  
};
```

*Обычно конструкторы **не имеют** оператора `return`. Их задача – записать все необходимое в `this`, и это автоматически становится результатом.

Но если `return` всё же есть, то применяется простое правило:

1. При вызове `return` с объектом, вместо `this` вернётся объект.
2. При вызове `return` с примитивным значением, оно проигнорируется.
3. Другими словами, `return` с объектом возвращает этот объект, во всех остальных случаях возвращается `this`.

6. Пример создание класса в JS

```
// создание методов внутри класса
function Point(x, y) {
  this.m_x = x;
  this.m_y = y;
  this.setX = function(x) {
    this.m_x = x;
  }
  this.x = function(){
    return this.m_x;
  }
}
```

Основной целью **конструкторов** является — реализация кода для многократного создания однотипных объектов.

```
// создание объекта и вывод значения на экран
let pt = new Point(20, 30);
alert("X=" + pt.x() + ";Y=" + pt.y());
```

Задание значений по умолчанию для полей класса в JS (используем оператор `||`):

```
this.m_x = x || 0;
this.m_y = y || 0;
this.m_str = str || "";
```

Создание экземпляров (объектов) класса:

```
let pt = new Point;      // x=0, y=0
let pt1= new Point(3);   // x=3, y=0
let pt2 = new Point(2, 5); // x=2, y=5
```

7. Инкапсуляция в JS

Инкапсуляция, изолирует данные одного экземпляра класса от данных в другом экземпляре того же самого класса.

В JS отсутствует разграничение доступа к членам класса при помощи спецификаторов: **private**, **public**, **protected** (За икл. использования **let**!). В следствии этого к членам класса можно обращаться откуда угодно.

Инкапсуляция – важная часть объектно-ориентированного программирования, чтобы данные в разных экземплярах класса были *отделены друг от друга*; это реализуется в JS с помощью оператора **this**. Однако, в отличие от других объектно-ориентированных языков программирования, JS не ограничивает доступ к данным внутри экземпляра класса.

Закрытие членов класса можно реализовать при помощи использования ключевого слова **let** (член класса становится локальным !)

```
function Point(x, y) {  
  this.m_x = x || 0;  
  this.m_y = y || 0;  
  let privateVar = 8;  
  let privateMeth = function() {  
    alert("private variable value:" +  
      privateVar);  
  }  
  this.setX = function(x) {  
    privateMeth();  
    this.m_x = x;  
  }  
  ...  
}
```

```
let pt = new Point(20, 30);  
alert(pt.privateVar);
```

error
(undefined)

8. Наследование

Чтобы унаследовать существующий класс в **JS**, необходимо создать функцию конструктора нового класса и из него при помощи функции **call ()** запустить конструктор наследуемого класса.

Наследуемый класс

```
function ThreeDPoint(x, y, z) {  
  Point.call(this, x, y);  
  this.m_z = z;  
  this.setZ = function(z) {  
    this.m_z = z;  
  }  
  this.z = function() {  
    return this.m_z;  
  }  
}
```

Новый класс

Создание и вызов объекта нового класса

```
let pt = new ThreeDPoint(20, 30, 40);  
alert("X=" + pt.x() + ";Y=" + pt.y() + ";Z=" + pt.z());
```

Добавление новых методов

```
...  
pt.test = function() {  
  return "Test";  
}  
pt.x = function () {  
  return -1;  
}
```

Вызов добавленных методов

```
...  
alert("X=" + pt.x() + "; " + pt.test());
```

*После того как объект создан, при необходимости можно добавлять к нему новые методы и переопределять уже существующие.

9. Прототипное наследование

Повторное использование функционала одного класса в другом позволяет **прототипное наследование**.

В JS объекты имеют специальное скрытое свойство **[[Prototype]]**, которое либо равно **null**, либо ссылается на другой объект. Этот объект наз. «**прототип**»

```
let animal = {
  eats: true,
};
let rabbit = {
  jumps: true,
};
rabbit.__proto__ = animal;
alert(rabbit.eats);
```

Когда запрашивается св-во у object, а оно отсутствует, JS автоматически берет его из его прототипа.

```
let animal = {
  eats: true,
};
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype = animal;
let rabbit = new Rabbit("Stepan");
alert(rabbit.eats);
```

Св-во «**prototype**»
ф-конструктора

«Прототип
object»

[[Prototype]]

«object»

При создании объекта через **new Rabbit**, новому объекту запишется «**animal**» в св-во **[[Prototype]]**

10. Прототипное наследование

```
function Point(x, y) { // констр-р баз-го класса
  this.m_x = x;
  this.m_y = y;
}
function ThreeDPoint(x, y, z) { // констр-р доч-го класса
  this.base = Point;
  this.base(x, y);
  this.m_z = z;
}
// создание экземпляра дочернего класса
ThreeDPoint.prototype = new Point;
var pt = new ThreeDPoint(1, 2, 3);
print("X=" + pt.m_x + ";Y=" + pt.m_y + ";Z=" + pt.m_z);

delete pt.m_z;           // удаление поля из класса
pt.hasOwnProperty('m_z'); //=>true проверка
                          // существ. поля в классе
```

obj instanceof Class; // принадлежность объекта к классу

```
pt instanceof ThreeDPoint; //=> true
pt instanceof Point;       //=> true
pt instanceof Date;        //=> false
```

11. Перегрузка методов

Перегрузка методов используется чтобы изменить поведение уже существующих объектов.

```
function Point(x, y){  
  this.m_x=x;  
  this.m_y=y;  
  this.x = function(){  
    return this.m_x;  
  };  
}
```

```
// создание объекта  
let pt = new Point(1, 2);  
alert("X=" + pt.x() ); // 1  
  
// добавление нового функционала  
function myX () {  
  return 1234;  
}  
// перегрузка метода x()  
pt.x = myX; // замена x() -> myX()  
alert("X=" + pt.x() ); // 1234
```

12. JSON (Java Script Object Notation)

JS предоставляет методы:

JSON.stringify () - преобразование объекта, массива в *JSON(строку)*. Строка становится *JSON*-форматированным или сериализованным объектом.

JSON.parse() – преобразование *JSON* обратно в объект или массив

```
let student = {  
  name: 'John',  
  age: 30,  
  wife: null,  
  courses: ['html', 'css', 'js'],  
};
```

```
let json = JSON.stringify(student);  
alert(typeof json);  
alert(json);
```

Объект в формате JSON имеет отличие от объектного литерала:

- Строки используют двойные кавычки;
- Имена свойств также заключены в двойные кавычки
- **JSON.stringify () пропускает** при преобразовании: (свойства-функции, символьные ключи и значения, свойства со знач. **undefined**)
- Встроенные объекты поддерживаются и конвертируются автоматически.

```
let student_str='{"name":"John", "age":30, "courses":["html","css","js"], "wife":null}';  
let student = JSON.parse(student_str);  
alert(student.age);
```

13. Домашка #5

Почему наедаются оба хомяка?

У нас есть два хомяка:
шустрый (**speedy**) и ленивый (**lazy**);
оба наследуют от общего объекта
hamster.

Когда мы кормим одного хомяка,
второй тоже наедается.
Почему? Как это исправить?

```
let hamster = {  
  stomach: [ ],  
  eat(food) {  
    this.stomach.push(food);  
  }  
};  
  
let speedy = {  
  __proto__: hamster  
};  
  
let lazy = {  
  __proto__: hamster  
};  
// Этот хомяк нашёл еду  
speedy.eat("apple");  
alert( speedy.stomach ); // apple  
// У этого хомяка тоже есть еда. Почему? Исправьте  
alert( lazy.stomach ); // apple
```