Разработка пользовательского веб-интерфейса

ООП JavaScript

Сергей Геннадьевич Синица

КубГУ, 2020

sin@kubsu.ru

Зачем нужно ООП?

Принципы ООП

Абстракция

Модульность

Полиморфизм

Инкапсуляция

Терминология

Пространство имён - Контейнер, который позволяет разработчикам связать весь функционал под уникальным, специфичным для приложения именем.

Класс - Определяет характеристики объекта. Класс является описанием шаблона свойств и методов объекта.

Объект - Экземпляр класса.

Свойство - Характеристика объекта, например, цвет.

Метод - Это подпрограмма или функция, связанная с классом.

Конструктор - Метод, вызываемый в момент создания экземпляра объекта. Он, как правило, имеет то же имя, что и класс, содержащий его.

Наследование - Класс может наследовать характеристики от другого класса.

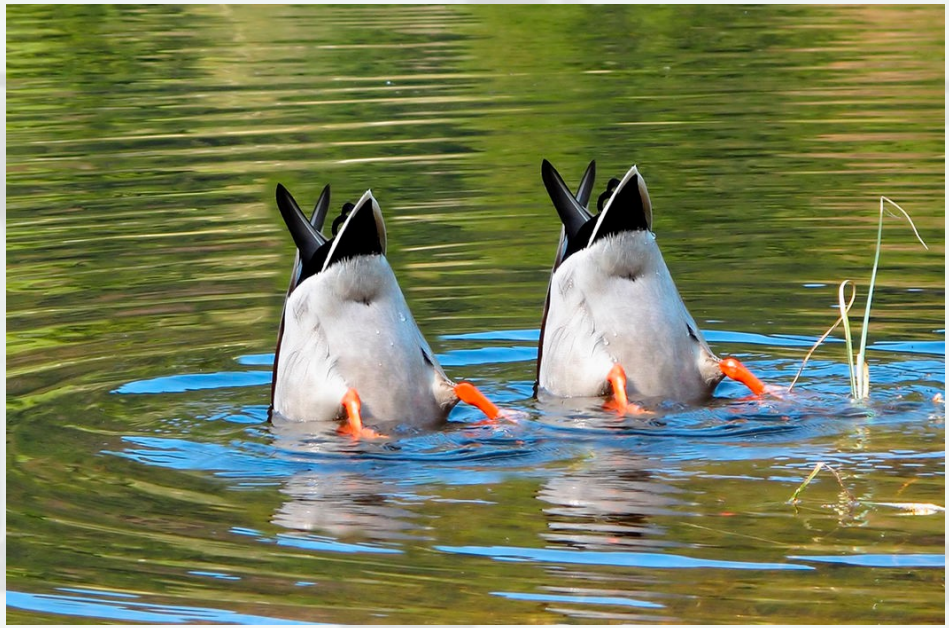
Инкапсуляция - Способ комплектации данных и методов, которые используют данные. Позволяет скрывать ненужные детали и скрывать сложность.

Абстракция - Отражение модели реальности в коде с помощью совокупности комплексных наследований, методов и свойств объекта. Позволяет задавать общее поведение на абстракном уровне.

Полиморфизм - Поли означает "много", а морфизм "формы". Различные классы могут объявить один и тот же метод или свойство. Код их использующий в зависимости от параметров будет работать, вызывая разную реализацию методов.

Прототипное программирование

Это модель ООП которая не использует классы, а вместо этого сначала выполняет поведение класса и затем использует его повторно, декорируя (или расширяя) существующие объекты прототипы.



Бесклассовое, прототипно-ориентированное, или экземплярно-ориентированное программирование.

Прототипы

```
//Пример наследования в прототипном программировании
//на примере языка JavaScript

//Создание нового объекта
var foo = {name: "foo", one: 1, two: 2};

//Создание еще одного нового объекта
var bar = {two: "two", three: 3};

bar.__proto__ = foo; // foo теперь является прототипом для bar

//Если теперь мы попробуем получить доступ к полям foo из bar
//то все получится
bar.one // Равно 1

//Свои поля тоже доступны
bar.three // Равно 3

//Собственные поля выше по приоритету полей прототипов
bar.two; // Равняется "two"
```

Литерал объекта (object literal)

```
var person = {};

var person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age
+ ' years old. He likes ' + this.interests[0] + ' and ' +
this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
```

Точечная запись

```
person.name
person.name[0]
person.age
person.interests[1]
person.bio()
person.greeting()
```

Внутренние пространства имен (Sub-namespaces)

```
name: ['Bob', 'Smith'],

name[0]
name[1]

name : {
  first: 'Bob',
  last: 'Smith'
},

name.first
name.last

person.name.first
person.name.last
```

Пространства имен

```
// Глобальное пространство имён
var MYAPP = MYAPP || {};

// Подпространство имён
MYAPP.event = {};
```

```
// Создаём контейнер MYAPP.commonMethod для общих методов и свойств
MYAPP.commonMethod = {
  regExForName: "", // определяет регулярное выражение для валидации имени
  regExForPhone: "", // определяет регулярное выражение для валидации телефона
  validateName: function(name){
    // Сделать что-то с name, вы можете получить доступ к переменной regExForName
    // используя "this.regExForName"
  },

  validatePhoneNo: function(phoneNo){
    // Сделать что-то с номером телефона
  }
}


// Объект вместе с объявлением методов
MYAPP.event = {
    addListener: function(el, type, fn) {
    // код
    },
    removeListener: function(el, type, fn) {
    // код
    },
    getEvent: function(e) {
    // код
    }

    // Можно добавить другие свойства и методы
}


// Синтаксис использования метода addListener:
MYAPP.event.addListener("yourel", "type", callback);
```

Скобочная запись (Bracket notation)

```
person.age
person.name.first

person['age']
person['name']['first']
```

Math

Встроенный объект Math содержит методы (функции)
Math.sin()
Math.pow()
Math.sqrt()
Math.random()

…

и свойства (константы)
Math.PI

Запись элементов в объект

```
person.age = 45;
person['name']['last'] = 'Cratchit';

person['eyes'] = 'hazel';

person.farewell = function() {
  alert("Bye everybody!");
}

person['eyes']
person.farewell()
```

Динамические имена полей

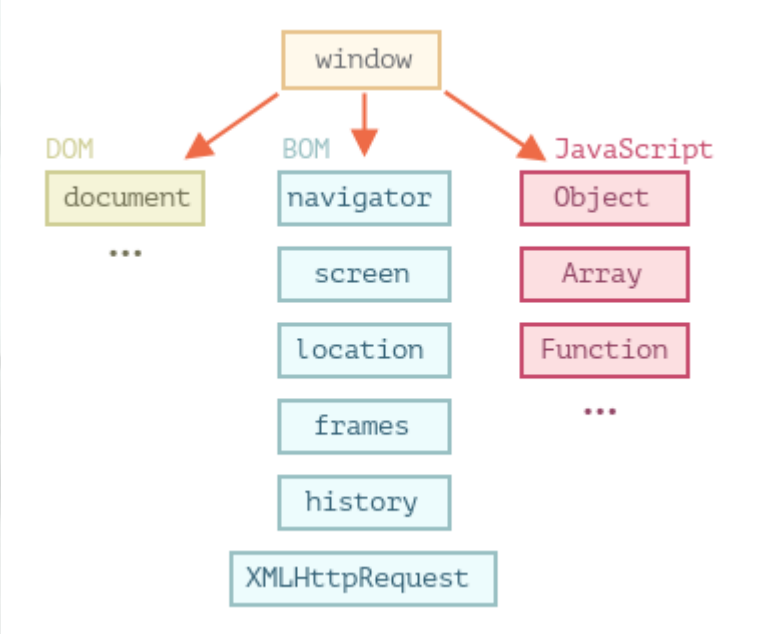
```
var myDataName = nameInput.value;
var myDataValue = nameValue.value;

person[myDataName] = myDataValue;
```

Встроенные объекты браузера

myString.split(',');

var myDiv = document.createElement('div');
var myVideo = document.querySelector('video');



https://learn.javascript.ru/browser-environment

Что такое "this"?

```
greeting: function() {
  alert('Hi! I\'m ' + this.name.first + '.');
}

var person1 = {
  name: 'Chris',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}

var person2 = {
  name: 'Brian',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

Класс?

```
var Person = function () {};

var person1 = new Person();
var person2 = new Person();
```

Конструктор:

```
var Person = function () {
  console.log('instance created');
};

var person1 = new Person();
var person2 = new Person();
```

Свойства

```
var Person = function (firstName) {
  this.firstName = firstName;
  console.log('Person instantiated');
};

var person1 = new Person('Alice');
var person2 = new Person('Bob');

// Выводит свойство firstName в консоль
console.log('person1 is ' + person1.firstName); // выведет
"person1 is Alice"
console.log('person2 is ' + person2.firstName); // выведет
"person2 is Bob"
```

Методы

```
var Person = function (firstName) {
  this.firstName = firstName;
};

Person.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");

// вызываем метод sayHello() класса Person
person1.sayHello(); // выведет "Hello, I'm Alice"
person2.sayHello(); // выведет "Hello, I'm Bob"
```

Наследование

```
const person = {
  isHuman: false,
  printIntroduction: function () {
    console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
  }
};

// Object.create() method creates a new object,
// using an existing object as the prototype of the newly created object.
const me = Object.create(person);

me.name = "Matthew"; // "name" is a property set on "me", but not on "person"
me.isHuman = true; // inherited properties can be overwritten

me.printIntroduction();
// expected output: "My name is Matthew. Am I human? true"
```

Наследование

```
// Shape - superclass
function Shape() {
  this.x = 0;
  this.y = 0;
}

// superclass method
Shape.prototype.move = function(x, y) {
  this.x += x;
  this.y += y;
  console.info('Shape moved.');
};

// Rectangle - subclass
function Rectangle() {
  Shape.call(this); // call super constructor.
}

// subclass extends superclass
Rectangle.prototype = Object.create(Shape.prototype);

//If you don't set Object.prototype.constructor to Rectangle,
//it will take prototype.constructor of Shape (parent).
//To avoid that, we set the prototype.constructor to Rectangle (child).
Rectangle.prototype.constructor = Rectangle;
```

Наследование

```
var rect = new Rectangle();

console.log('Is rect an instance of Rectangle?', rect instanceof Rectangle); // true
console.log('Is rect an instance of Shape?', rect instanceof Shape); // true
rect.move(1, 1); // Outputs, 'Shape moved.'
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create

__proto__ vs Object.prototype

Свойство __proto__ объекта Object.prototype является свойством доступа (комбинацией геттера и сеттера), которое расширяет внутренний прототип [[Prototype]] объекта (являющийся объектом или null), через который осуществлялся доступ.

Только недавно было стандартизировано в новой спецификации ECMAScript 6.

Очень медленная реализация! Используем лучше Object.create()

Пример:

```
var shape = {}, circle = new Circle();

// Установка прототипа объекта
shape.__proto__ = circle;
// Получение прототипа объекта
console.log(shape.__proto__ === circle); // true
```

Наследование пример

```
// Определяем конструктор Person
var Person = function(firstName) {
  this.firstName = firstName;
};


// Добавляем пару методов в Person.prototype
Person.prototype.walk = function(){
  console.log("I am walking!");
};


Person.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName);
};


// Определяем конструктор Student
function Student(firstName, subject) {
  // Вызываем конструктор родителя, убедившись (используя Function#call)
  // что "this" в момент вызова установлен корректно
  Person.call(this, firstName);


  // Инициируем свойства класса Student
  this.subject = subject;
};
```

Наследование пример

```
// Создаём объект Student.prototype, который наследуется от Person.prototype.
// Примечание: Рспространённая ошибка здесь, это использование "new Person()", чтобы создать
// Student.prototype. Это неверно по нескольким причинам, не в последнюю очередь
// потому, что нам нечего передать в Person в качестве аргумента "firstName"
// Правильное место для вызова Person показано выше, где мы вызываем
// его в конструкторе Student.
Student.prototype = Object.create(Person.prototype); // Смотрите примечание выше


// Устанавливаем свойство "constructor" для ссылки на класс Student
Student.prototype.constructor = Student;


// Заменяем метод "sayHello"
Student.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName + ". I'm studying "
              + this.subject + ".");
};
```

Наследование пример

```
// Добавляем метод "sayGoodBye"
Student.prototype.sayGoodBye = function(){
  console.log("Goodbye!");
};

// Пример использования:
var student1 = new Student("Janet", "Applied Physics");
student1.sayHello();   // "Hello, I'm Janet. I'm studying Applied
Physics."
student1.walk();       // "I am walking!"
student1.sayGoodBye(); // "Goodbye!"

// Проверяем, что instanceof работает корректно
console.log(student1 instanceof Person);  // true
console.log(student1 instanceof Student); // true
```

Множественное наследование, подмешивания

```
function MyClass() {
  SuperClass.call(this);
  OtherSuperClass.call(this);
}

// inherit one class
MyClass.prototype = Object.create(SuperClass.prototype);
// mixin another
Object.assign(MyClass.prototype, OtherSuperClass.prototype);
// re-assign constructor
MyClass.prototype.constructor = MyClass;

MyClass.prototype.myMethod = function() {
  // do something
};

/* Object.assign() copies properties from the OtherSuperClass
prototype to the MyClass prototype, making them available to all
instances of MyClass. Object.assign() was introduced with ES2015 and
can be polyfilled. If support for older browsers is necessary,
jQuery.extend() or _.assign() can be used.*/
```

ES6 классы

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

Подъём (hoisting)

Разница между объявлением функции (function declaration) и объявлением класса (class declaration) в том, что объявление функции совершает подъём, в то время как объявление класса — нет. Поэтому вначале необходимо объявить ваш класс и только затем работать с ним, а код же вроде следующего сгенерирует исключение типа:

```
var p = new Rectangle(); // ReferenceError

class Rectangle {}
```

Подъём (hoisting)

```
function hoist() {
  console.log(message);
  var message='Hoisting is all the rage!'
}

hoist();
```

эквивалентно

```
function hoist() {
  var message;
  console.log(message);
  message='Hoisting is all the rage!'
}

hoist(); // Вывод: undefined
```

ES6 let и const

```
console.log(hoist); // Вывод: Uncaught ReferenceError: can't access
lexical declaration 'hoist' before initialization
let hoist = 'The variable has been hoisted.';
console.log(hoist2); // Uncaught ReferenceError: hoist2 is not defined

const PI = 3.142;

PI = 22/7; // Давайте изменим значение PI
console.log(PI); // Вывод: TypeError: Assignment to constant variable.


// let и const обладают лексической видимостью, «поднимаются» вверх
блока кода {}

Переменная, объявленная через var, видна везде в функции.

Переменная, объявленная через let, видна только в рамках блока {...},
в котором объявлена.
```

Выражение класса

```
// безымянный
var Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

// именованный
var Rectangle = class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Методы прототипа

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

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.height * this.width;
  }
}

const square = new Rectangle(10, 10);

console.log(square.area);
```

Статические методы

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.hypot(dx, dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));

// Статические методы вызываются без инстанцирования их класса, и не могут быть вызваны у экземпляров (instance) класса.
```

Упаковка в прототипных и статических методах

Когда статический или прототипный метод вызывается без привязки к "this" объекта (или когда "this" является типом boolean, string, number, undefined, null), тогда "this" будет иметь значение "undefined" внутри вызываемой функции. Автоупаковка не будет произведена. Поведение будет таким же как если бы мы писали код в нестрогом режиме.

```
class Animal {
  speak() {
    return this;
  }
  static eat() {
    return this;
  }
}

let obj = new Animal();
obj.speak(); // Animal {}
let speak = obj.speak;
speak(); // undefined

Animal.eat() // class Animal
let eat = Animal.eat;
eat(); // undefined
```

Упаковка в прототипных и статических методах

Если мы напишем этот же код используя классы основанные на функциях, тогда произойдет автоупаковка основанная на значении "this", в течение которого функция была вызвана.

```
function Animal() { }

Animal.prototype.speak = function(){
  return this;
}

Animal.eat = function() {
  return this;
}

let obj = new Animal();
let speak = obj.speak;
speak(); // глобальный объект

let eat = Animal.eat;
eat(); // глобальный объект
```

“use strict;”

Режим strict (строгий режим), введенный в ECMAScript 5, позволяет использовать более строгий вариант JavaScript. Это не просто подмножество языка: в нем сознательно используется семантика, отличающаяся от обычно принятой.

Строгий режим принёс ряд изменений в обычную семантику JavaScript:

- заменяет исключениями некоторые ошибки, которые интерпретатор JavaScript ранее молча пропускал;

- исправляет ошибки, которые мешали движкам JavaScript выполнять оптимизацию -- в некоторых случаях код в строгом режиме может быть оптимизирован для более быстрого выполнения, чем код в обычном режиме;

- запрещает использовать некоторые элементы синтаксиса, которые, вероятно, в следующих версиях ECMAScript получат особый смысл.

```
function strict() {
  // Строгий режим на уровне функции
  "use strict";
  function nested() { return "И я тоже!"; }
  return "Привет! Я функция в строгом режиме! " + nested();
}

function notStrict() { return "Я не strict."; }
```

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Strict_mode

“use strict;” и this

Значение, передаваемое в функцию как this, в строгом режиме не приводится к объекту (не "упаковывается"). В обычной функции this всегда представляет собой объект: либо это непосредственно объект, в случае вызова с this, представляющим объект-значение; либо значение, упакованное в объект, в случае вызова с this типа Boolean, string, или number; либо глобальный объект, если тип this это undefined или null. (Для точного определения конкретного this используйте call, apply, или bind.) Автоматическая упаковка не только снижает производительность, но и выставляет на показ глобальный объект, что в браузерах является угрозой безопасности, потому что глобальный объект предоставляет доступ к функциональности, которая должна быть ограничена в среде "безопасного" JavaScript. Таким образом, для функции в строгом режиме точно определённый this не упаковывается в объект, а если не определён точно, this является undefined:

```
"use strict";
function fun() { return this; }
console.assert(fun() === undefined);
console.assert(fun.call(2) === 2); // заменяет this
console.assert(fun.apply(null) === null); // аналогично, но для переменного числа аргументов
console.assert(fun.call(undefined) === undefined);
console.assert(fun.bind(true)() === true); // привязка контекста функции
```

Наследование классов с помощью extends

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

  speak() {
    console.log(this.name + ' издает звук.');
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' лает.');
  }
}

var d = new Dog('Митци');
d.speak();
```

Работает с функциями

```
function Animal (name) {
  this.name = name;
}
Animal.prototype.speak = function () {
  console.log(this.name + ' издает звук.');
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' лает.');
  }
}

var d = new Dog('Митци');
d.speak();
```

setPrototypeOf()

Обратите внимание, что классы не могут расширять обычные (non-constructible) объекты. Если вам необходимо создать наследование от обычного объекта, в качестве замены можно использовать Object.setPrototypeOf():

```
var Animal = {
  speak() {
    console.log(this.name + ' издает звук.');
  }
};

class Dog {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' лает.');
  }
}
Object.setPrototypeOf(Dog.prototype, Animal);

var d = new Dog('Митци');
d.speak();
```

Модули ES6

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
- import позволяет импортировать функциональность из других модулей.

Например, если у нас есть файл sayHi.js, который экспортирует функцию:

```
// sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

…Тогда другой файл может импортировать её и использовать:

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

Директива import загружает модуль по пути ./sayHi.js относительно текущего файла и записывает экспортированную функцию sayHi в соответствующую переменную.

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">

Модули ES6

Модули всегда работают в strict-режиме.

Модули из внешних файлов загружаются отложенно.

Каждый модуль имеет свою собственную область видимости.

Код в модуле выполняется только один раз при импорте.

В модуле «this» не определён на верхнем уровне.

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</script>
```