УДК 004 **Информационные технологии**

Мавлянов Азизбек Нажот Угли, *магистрант*
факультет программной инженерии и компьютерной техники
Университет ИТМО, Россия, г. Санкт-Петербург

ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ ТРАДИЦИОННЫХ КОНЦЕПЦИЙ ООП С ИСПОЛЬЗОВАНИЕМ ПРОТОТИПНОГО ПРОГРАММИРОВАНИЯ JAVASCRIPT

Аннотация: В настоящей статье отражены эффективные способы реализации традиционных концепций ООП (объектно-ориентированное программирование) с использованием прототипного программирования. Данные способы реализации ООП будут продемонстрированы на языке программирования JavaScript.

Ключевые слова: объектно-ориентированное программирование, прототипное программирование, объект, класс, функция.

Abstract: this article presents effective ways to implement traditional concepts of OOP (object-oriented programming) using prototype programming. These methods of implementing OOP will be demonstrated in the JavaScript programming language.

Keywords: object-oriented programming, prototype programming, object, class, function.

На сегодняшний день очень трудно представить экосистему веб-технологий без языка программирования JavaScript. Данный язык обладает сильными возможностями объектно-ориентированного программирования на основе прототипной модели. Благодаря этим возможностям наблюдается постепенный

переход веб-разработчиков из других языков программирования в экосистему JavaScript и активное использования данного языка. При переходе на данный язык среди разработчиков очень часто ведутся некоторые споры из-за различий в объектно-ориентированном JavaScript по сравнению с другими языками. Более того, наблюдается, что большинство разработчиков максимально стремятся реализовать традиционные концепции ООП использую существующие возможности JavaScript, что обуславливает актуальность данной работы.

Целью данной работы является демонстрация эффективной реализации концепций ООП с использованием прототипного программирования, позволяющая разработчикам применить функционал классического ООП на языке программирования JavaScript.

Необходимо отметить, что JavaScript – это язык, основанный на прототипах, который не содержит под капотом операторов класса (class), как, например, в C ++ или Java [2]. Иногда это сбивает с толку программистов, привыкших к языкам с выражением класса. Вместо этого JavaScript оперирует прототипами и функциями для того, чтобы реализовать функционал традиционных классов. Определить класс так же просто, как определить функцию.

Прежде чем начать демонстрацию, выделим список основных понятий, которые наиболее часто используются в экосистеме ООП:

- Объект;
- Класс;
- Инкапсуляция;
- Абстракция;
- Наследование;
- Полиморфизм.

На рисунке 1 можно увидеть создание объекта «book», со свойствами, такими как автор (author), год публикации (year) и название (title) JavaScript. Значения

свойств объектных литералов могут быть любыми типами данных, такими как функциональные литералы, массивы, строки, числа или логические значения.

```
const book = {
  title: "JavaScript: The Good Parts",
  author: "Douglas Crockford",
  year: 2020
};
```

Рисунок 1. Создания объекта в JavaScript

После создания объекта можно получить значения свойств и методов с помощью точечной записи. Например, мы можем получить значение заголовка с помощью «book.title». Мы можем также получить доступ к свойствам с помощью квадратных скобок «book[“title”]».

В JavaScript конструктор объекта такой же, как и обычная функция. Также его называют как «функция-конструктор» [4]. Он вызывается каждый раз при создании объекта. Функция-конструктор может быть использован с ключевым словом «new». На рисунке 2 можно увидеть кусок кода, в котором показано использование функция-конструктора для создания объекта «book1». Функция-конструктор объектов полезен, когда мы хотим создать несколько объектов с одинаковыми свойствами и методами.

```
function Book(title, author, year) {
  this.title = title;
  this.author = author;
  this.year = year;
}

const book1 = new Book("JavaScript: The Good Parts", "Douglas Crockford", 2020);

console.log(book1);

// > Book {
//     title: 'JavaScript: The Good Parts',
//     author: 'Douglas Crockford',
//     year: 2020
//   }
```

Рисунок 2. Создания объекта с помощью конструктор

В JavaScript класс – это не объект, а схема на основе, которой создаются объекты. Классы – это особые функции JavaScript [1]. Как показано на рисунке 3 функции могут быть определены с помощью выражения и объявления функций (function expression and declaration), а также классы могут быть определены точно таким же образом.

Существует метод «Object.create», который позволяет создать новый объект, использую существующий объект в качестве прототипа. На рисунке 3 можно увидеть реализацию вышесказанного.

```
23  const Book = function (bookName) {
24    this.bookName = bookName;
25  };
26
27  let newBook = function (bookName) {
28    Book.call(this, bookName);
29  };
30
31  newBook.prototype = Object.create(Book.prototype);
32  const book1 = newBook("OOP in JavaScript");
```

Рисунок 3. Реализация класса «Book»

Очень часто трактуется, что инкапсуляция – это сокрытие информации или данных [5]. Это относится к способности объекта выполнять свои функции без раскрытия каких-либо подробностей выполнения вызывающему. Другими словами, частная переменная видна только текущей функции и недоступна для глобальной области или других функций. На рисунке 4 приведен пример кода реализации инкапсуляции. В данном коде заголовок и автор видны только внутри области действия функции «Book», а метод «summary» виден вызывающему «Book». Таким образом, название и автор заключены внутри «Book».

```
34  const Book = function (title, author) {
35    let title = title;
36    let author = author;
37
38    return {
39      summary: function () {
40        console.log(`${title} written by ${author}.`);
41      },
42    };
43  };
44  const book1 = new Book("Douglas Crockford", "JavaScript: The Good Parts");
45  book1.summary();
```

Рисунок 4. Реализация инкапсуляции

Как известно, абстракция – это сокрытие реализации. Это способ скрыть детали реализации и показать вызывающему только основные функции. Другими словами, он скрывает несущественные детали и показывает только то, что необходимо внешнему миру. Отсутствие абстракции приведет к проблемам с ремонтопригодностью кода [3]. На рисунке 5 приведен пример реализации абстракции в JavaScript. Из рисунка видно, что реализации методов «giveTitle» «giveSummary» скрыты от внешнего мира. В данным случае можно лишь обратиться к самим методам, а не к их реализации, так как внутренний механизм работы методов инкапсулированы.

```
47  const Book = function (getTitle, getAuthor) {
48    // Private variables / properties
49    let title = getTitle;
50    let author = getAuthor;
51    // Public method
52    this.giveTitle = function () {
53      return title;
54    };
55
56    // Private method
57    const summary = function () {
58      return `${title} written by ${author}.`;
59    };
60    // Public method that has access to private method.
61    this.giveSummary = function () {
62      return summary();
63    };
64  };
65  const book1 = new Book("Douglas Crockford", "JavaScript: The Good Parts");
66  book1.giveTitle(); // "JavaScript: The Good Parts"
67  book1.summary(); // Uncaught TypeError: book1.summary is not a function
68  book1.giveSummary(); // "JavaScript: The Good Parts written by Douglas Crockford."
```

Рисунок 5. Реализация абстракции

Как правило, JavaScript не является языком на основе классов. Несмотря на то, что слово «class» был введен в ES6 как синтаксический сахар, JavaScript под капотом использует прототипы для реализации ООП функционала [6]. В JavaScript наследование достигается с помощью прототипа. Этот шаблон называется шаблоном делегирования поведения или прототипным наследованием. На рисунке 6 можно увидеть код реализации прототипного наследования. В данным коде создается функция-конструктор «Vehicle» в качестве суперкласса. Далее, создается метод «start» класса «Vehicle». После создания данного метода создается класс «Car», который наследует класс «Vehicle». Внутри класса «Car» вызывается метод «call» класса «Vehicle». В данным случае метод «call» выполняет функцию супер-конструктора из классического ООП, передав ему свой контекст с помощью «this» и аргумент «name». Далее, реализуется наследование класса «Car» от класса «Vehicle». Для этого прототип «Car.prototype» класса «Car» приравнивается к результату метода «Object.create», на вход, которого передается прототип «Vehicle.prototype» класса «Vehicle». После этого создается метод «run» класса «Car». Так как класс «Car» наследует класс «Vehicle», внутри реализации метода

«run» можно вызвать метод «start» класса «Vehicle» с помощью служебного слово «this». Далее, создаются два объекта «c1» и «c2» с помощью функция-конструктора «Car», на вход которого передаются такие аргументы как «Lada» и «Ravon». В конце вызывается метод «run» у обоих объектов «c1» и «c2» для вывода результатов. Из полученных результатов можно увидеть, что вызов метода «start» был произведен успешно, поскольку класс «Car» наследует практически все свойства и методы класса «Vehicle», в том числе и метод «start».

```
// Vehicle - superclass
function Vehicle(name) {
  this.name = name;
}
// superclass method
Vehicle.prototype.start = function () {
  return "engine of " + this.name + " starting...";
};
// Car - subclass
function Car(name) {
  Vehicle.call(this, name); // call super constructor.
}
// subclass extends superclass
Car.prototype = Object.create(Vehicle.prototype);
// subclass method
Car.prototype.run = function () {
  console.log("Hi " + this.start());
};
// instances of subclass
var c1 = new Car("Lada");
var c2 = new Car("Ravon");
// accessing the subclass method which internally access superclass method
c1.run(); // "Hi engine of Lada starting..."
c2.run(); // "Hi engine of Ravon starting..."
```

Рисунок 6. Реализация прототипного наследования

На рисунке 7 можно увидеть вышеописанную реализацию прототипного наследования в виде диаграммы. Данная диаграмма иллюстрирует взаимосвязь между классами и объектами, которые оперируют прототипами для реализации ООП функционала.

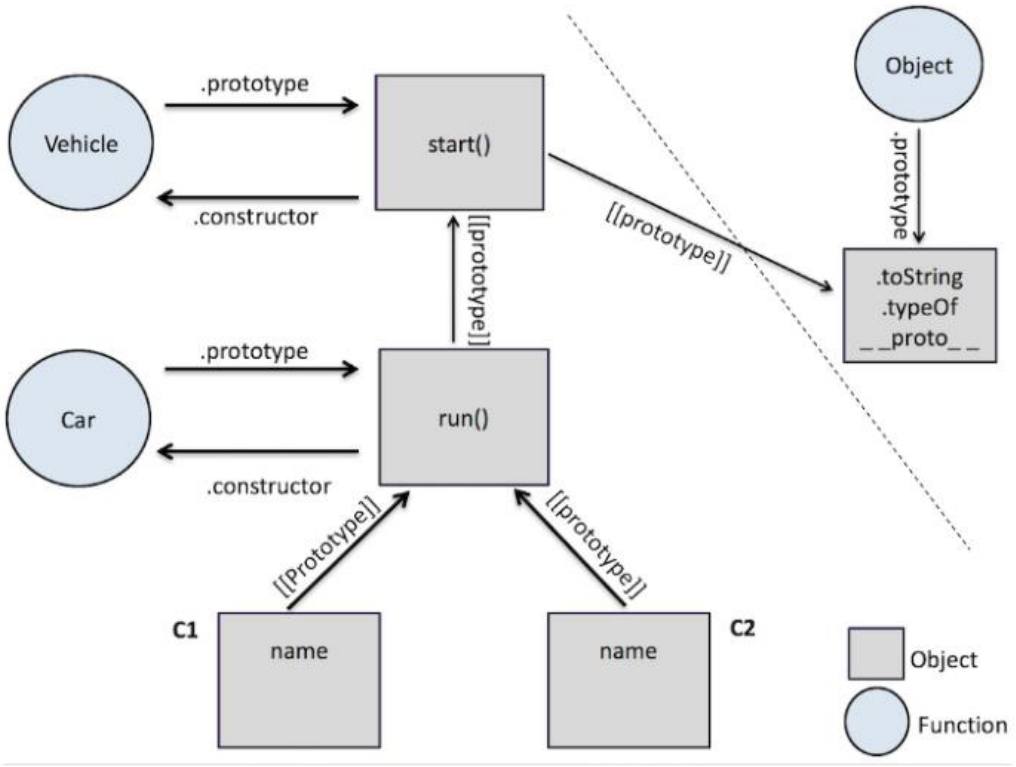

Рисунок 7. Диаграмма реализации прототипного наследования

Как известно, полиморфизм является одним из основных парадигм ООП. Если говорить кратко, полиморфизм – это способность вызывать один и тот же метод на разных объектах, и переопределять каждый из них чтобы они реагировали по-своему [3]. На рисунке 8 можно увидеть пример кода реализации полиморфизма в JavaScript. В данным примере создается функция-конструктор «book1». Далее, для него реализуется метод summary в прототипе. После этого создается еще один функция-конструктор «book2». Выражение «Object.create(book1.prototype)» приравнивается к прототипу данного конструктора. Таким образом осуществляется наследования «book2» от «book1». На данным этапе функция-конструктору «book2» будет доступен метод summary функция-конструктора «book1», поскольку «book2» наследует «book1». Несмотря на то, что метод summary уже определен в функция-конструкторе «book1», функция-конструктор «book2» переопределяет данный метод по-своему. Далее, можно увидеть аналогичную картину с функция-

конструктором «book3». В конце кода создается массив books из объектов функция-конструкторов «book1», «book2» и «book3», и методом перебора «forEach» выводятся результаты вызовов метода «summary» из каждого объекта. Из результата можно увидеть, что каждый метод был действительно переопределен и вывел собственный результат, а не тот результат, который был изначально прописан в родительском методе «summary» функция-конструктора «book1».

```
95   let book1 = function () {};
96   book1.prototype.summary = function () {
97     return "summary of book1";
98   };
99   let book2 = function () {};
100  book2.prototype = Object.create(book1.prototype);
101  book2.prototype.summary = function () {
102    return "summary of book2";
103  };
104  let book3 = function () {};
105  book3.prototype = Object.create(book1.prototype);
106  book3.prototype.summary = function () {
107    return "summary of book3";
108  };
109
110  var books = [new book1(), new book2(), new book3()];
111  books.forEach(function (book) {
112    console.log(book.summary());
113  });
114
115  // summary of book1
116  // summary of book2
117  // summary of book3
```

Рисунок 8. Пример кода реализации полиморфизма в JavaScript

В результате данной работы можно заключить, что используя существующие синтаксические элементы языка программирования JavaScript, достаточно эффективно можно реализовать практически все концепции классического ООП.

Библиографический список:

1. Crockford, D. JavaScript: The Good Parts, First Edition – US. – 2008 Vol. 228. – P. 92 – 115.

2. Flanagan, D. JavaScript: The Definitive Guide, Sixth Edition – US. – 2011 Vol. 1019. – P.199 – 246.

3. Marijn, H. Выразительный JavaScript – США. – 2014 – №3 С. 34-73.

4. Patro, N.C. JavaScript — Inheritance, delegation patterns and Object linking [Электронный ресурс] Режим доступа: https://codeburst.io/javascript-inheritance-25fe61ab9f85 (дата обращения: 07.10.2020).

5. Вступление в объектно-ориентированный JavaScript. [Электронный ресурс] Режим доступа: https://developer.mozilla.org/ru/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript (дата обращения: 07.10.2020).

6. П.А Васильев Обновление языка программирования javascript (es5) - ecmascript 2015 (es6). [Электронный ресурс] Режим доступа: https://cyberleninka.ru/article/n/obnovlenie-yazyka-programmirovaniya-javascript-es5-ecmascript-2015-es6 (дата обращения: 07.10.2020).