ООП в JavaScript

Знакомство с парадигмой ООП. Реализация ООП в JavaScript. Методы создания объектов и их поведение.

[ООП](#_5yohl8ijtl9q)

[Создание объектов и работа с ними](#_tyjcwt)

[Синтаксическое создание объектов](#_3dy6vkm)

[Использование функции конструктора](#_4d34og8)

[Создание объекта методом Object.create](#_2s8eyo1)

[Реализация ООП в JavaScript](#_17dp8vu)

[Практика](#_3rdcrjn)

[Вывод](#_vk8s30x3xtqw)

[Домашнее задание](#_lnxbz9)

[Используемая литература](#_osgi74sob3gv)

# ООП

ООП – это подход к программированию, который применим в языке JavaScript наряду с процедурным подходом.

ООП (Объектно-ориентированное программирование) – парадигма программирования, в которой основными концепциями являются понятия «класс» и «объект».

В отличие от процедурного программирования, ООП следует следующим архитектурным идеям:

1. В качестве фундаментальных логических единиц применяются объекты, а не алгоритмы (функции).
2. Каждый объект является экземпляром четко определенного класса.
3. Классы образуют иерархии.

Код считается объектно-ориентированным только в случае, если выполнены все три указанных требования.

Теперь рассмотрим основные понятия ООП.

**Объект** – это мыслимая или реальная сущность, обладающая характерным поведением и отличительными особенностями и являющаяся важной в предметной области. Каждый объект имеет состояние, обладает четко определенным поведением и уникальной идентичностью.

**Состояние** (state) – совокупный результат поведения объекта: одно из охарактеризованных количественно стабильных условий, в которых объект может существовать. В любой момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств.

**Поведение** (behavior) – действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

**Identity** (уникальность) объекта состоит в том, что всегда можно определить, указывают две ссылки на один и тот же объект или на разные. При этом два объекта могут во всем быть похожими, их образ в памяти может представляться одинаковыми последовательностями байтов, но, тем не менее, их Identity может быть различна.

Разумеется, нужно дать определение классу. Формально **класс** – это шаблон поведения объектов определенного типа с заданными параметрами, определяющими состояние. Все экземпляры одного класса (объекты, порожденные от одного класса) имеют один и тот же набор свойств и общее поведение, то есть одинаково реагируют на одинаковые сообщения.

Говорят, что объект – это экземпляр класса. Говоря иначе, класс – это идея, а объект – ее реализация.

**Поля** (свойства) класса – данные, которые хранит класс.

**Метод** класса – функция, объявленная внутри класса.

Говоря об ООП, нельзя не упомянуть три основных принципа:

* **Инкапсуляция** – свойство приложения, позволяющее объединить данные и методы, которые работают с ними, в определенном классе. Зачастую в это понятие включают также сокрытие реализации, т.е. невозможность переопределить тот или иной метод или параметр напрямую.
* **Наследование** – свойство приложения, которое даёт возможность описать новый класс на основе уже существующего. При этом функциональность может частично или полностью заимствоваться. Класс, от которого производится наследование, называется родительским, а новый класс – потомком или дочерним классом.
* **Полиморфизм** – способность объекта использовать методы производного класса, который не существует на момент создания базового.

# Создание объектов и работа с ними

Язык JavaScript неспроста носит в своём имени часть «Java», ведь, как и в Java, в JS почти все содержимое кода является объектом. Исключение – null и undefined, которые не обрабатываются как объекты.

Любой примитивный тип (integer, char, boolean и т.д.) в JS ведёт себя и обрабатывается как объект. Примитивы могут быть назначены свойствам объектов, у них есть характеристики, идентичные характеристикам объектов.

## Синтаксическое создание объектов

JS «из коробки» уже имеет определенный набор встроенных объектов. При этом, разумеется, можно создавать в коде свои собственные объекты. В версиях языка 1.2 и выше есть возможность создавать объект при помощи инициализатора объекта, но это не единственный способ. Как и в классических ООП-языках, можно создать конструктор и инстанцировать с его помощью экземпляр класса, вызвав в коде вне описания класса оператор new.

В JS есть особый способ создания объектов: вы можете записать объект в определённом формате непосредственно в коде, а интерпретатор при обработке кода автоматически создаст из этого кода настоящий объект.

Выглядит это следующим образом:

|  |
| --- |
| var obj = { width: 123,   property: "property value",   }; |

Разберём этот пример.

* Obj – это переменная, которая содержит наш новый объект.
* width и property являются идентификатором свойства (может иметь имя, число или строковый литерал), а каждый – значения, назначенные свойствам.

Если объект создан при помощи инициализатора объектов на высшем уровне скрипта, то JavaScript интерпретирует объект каждый раз, когда анализирует выражение, содержащее объект, записанный как литерал. Если пользоваться функцией-инициализатором, он будет создаваться каждый раз, когда функция вызывается.

Попробуем запрограммировать что-то, что не так сильно оторвано от реальности. Например, создадим объект audi, описывающий автомобиль.

|  |
| --- |
| var audi = {  color: "red",  wheels: 4,  engine: {  volume: 2.0,  power: 225 }  }; |

Обратите внимание, что в данном примере мы разместили в свойстве engine ещё один объект со своими собственными свойствами.

## Использование функции конструктора

Рассмотрим способ создания объектов через конструкторы. Он содержит два шага:

1. Создаём функцию-конструктор.
2. Создаём экземпляр класса при помощи ключевого слова new.

Продолжим нашу автомобильную тематику. Автомобиль марки Audi совершенно логично будет отнести к классу Автомобилей (Car). Создадим его, учитывая свойства, которые мы задали в предыдущем примере:

|  |
| --- |
| function Car(color, wheels, engine) {   this.color = color;  this.wheels = wheels;  this.engine = engine; }; |

Итак, внедрив конструктор класса, мы описали абстрактную машину (класс). Каждая машина (Audi, BMW, Nissan) будет являться экземпляром этого класса.

Очень важно в этом примере ключевое слово this, которое указывает, что мы присваиваем значения свойствам текущего объекта, т.е. это указатель на сам объект.

Теперь создание автомобиля становится гораздо проще:

|  |
| --- |
| var audi = new Car("red", 4, {volume: 2.0, power: 225}); var bmw = new Car("white", 4, {volume: 2.0, power: 194}); |

Мы создали два объекта класса Car. Audi и BMW являются ссылками на области памяти, в которых размещены соответствующие объекты, также мы поддерживаем случай, когда свойство основного объекта само по себе является объектом. При этом мы задаём его в синтаксическом виде, а можем описать конструктор класса Engine, создать нужные экземпляры этого класса, а затем передать их в качестве аргументов при создании класса Car.

Создание объектов – это здорово, но что с ними делать дальше? Допустим, мы хотим узнать, какого цвета машина, с которой мы работаем в данный момент. Тогда нужно будет вызвать следующую конструкцию:

|  |
| --- |
| audi.color; |

Похожим способом можно присвоить новое значение свойству объекта:

|  |
| --- |
| audi.color = "black"; |

Если же мы хотим получить значение мощности двигателя, то необходимо записывать адрес свойства в иерархическом порядке (т.е. так, как он хранится в самом объекте):

|  |
| --- |
| audi.engine.power; |

В JS всегда можно добавить новое свойство даже после создания объекта, но оно будет добавлено только одному экземпляру класса. Если вы хотите, чтобы свойство появилось у всех объектов, принадлежащих классу, нужно добавить это свойство в определение класса.

## Создание объекта методом Object.create

В JS доступен еще один специфический способ создания объектов – встроенный метод Object.create. Его основное преимущество в том, что он реализует наследование в JS и позволяет создавать объекты по прототипу, не определяя при этом конструктор.

Говоря проще, это означает, что когда вы ставите функции Car свойство Car.prototype = Vehicle – вы объявляете тем самым, что новые экземпляры класса Car будут иметь прототип Vehicle.

Таким образом, любое свойство, которое запросило приложение, сначала ищется в Car. Если его там нет, интерпретатор продолжит поиск в Car.prototype, т.е. в Vehicle.

|  |
| --- |
| function Vehicle() {  this.x = 0;  this.y = 0;  this.z = 0;  this.color = "white"; } Vehicle.prototype.move = function(x, y, z){  this.x = x;  this.y = y;  this.z = z; } function Car(){ *// вызываем родительский конструктор*  Vehicle.call(this); } Car.prototype = Object.create(Vehicle.prototype); Car.prototype.constructor = Car; var audi = new Car(); console.log('Audi - это машина?', audi instanceof Car); *// true* console.log('Audi - это средство передвижения?', audi instanceof Vehicle);*// true* |

Также не запрещается добавлять новые свойства к ранее заданному классу, используя свойство prototype. Так мы определяем свойство не только для текущего экземпляра, но для всего класса.

|  |
| --- |
| Car.prototype.color = null; bmw.color = "black"; |

При этом неважно, когда был создан объект данного типа – до добавления нового свойства или после, – свойство появится у всех объектов.

# Реализация ООП в JavaScript

На заре эпохи программирования существовал подход, с которым вы уже хорошо знакомы из «Базового курса JavaScript», – процедурный. Его суть состоит в том, что приложение содержит функции и их вызовы.

Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, требовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял функциональные требования, что еще более усложняло процесс создания программного обеспечения.

Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности.

Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало объектно-ориентированное программирование (ООП).

Применяя ООП, мы перестаем относиться к процессам как к последовательности функций, но начинаем применять более естественный нашему мышлению подход. Мы разделяем элементы процесса работы приложения на сущности (объекты) и описываем ход работы на их уровне. Объекты создаются, изменяют свое состояние, работают друг с другом, уничтожаются.

Если раньше детали интерфейса генерировались функциями, то при применении ООП любая сущность, будь то пользователь, компонент меню или ссылка, является логически завершенным элементом среды, объектом, имеющим методы и данные.

Методы и свойства объекта в ООП языках делятся на две большие группы:

* Внутренний интерфейс – методы и свойства, которые доступны только из других методов того же конкретного объекта.
* Внешний интерфейс – методы и свойства, которые доступны для всех объектов в текущем запуске приложения.

В автомобиле у вас есть руль, педали, селектор коробки передач – это внешний интерфейс управления объектом класса Car. Рулевые тяги, гидроусилитель, дроссельная заслонка, тормозная система и сама коробка передач недоступны для прямого воздействия (водитель не может толкать колеса в стороны руками на ходу, чтобы повернуть) – это внутренний интерфейс объекта класса Car.

Таким образом, при получении ссылки на объект нам достаточно знать, какие внешние методы управления он имеет. Зачастую о его внутреннем устройстве знать и не требуется.

Вспомним три столпа ООП: наследование, инкапсуляция и полиморфизм. Про реализацию наследования мы уже успели поговорить, затронув тему прототипов. Теперь обсудим инкапсуляцию.

**Инкапсуляция** JavaScript реализована через разделение данных объекта внутри класса. В отличие от классических ООП языков, JS не имеет разных степеней инкапсуляции (public, protected и private). Это означает, что ограничения доступа к данным по сути нет. Да и для JS это не столь актуально.

|  |
| --- |
| function Car() {  this.vinCode = "someVinCode"; } Car.prototype.setVin = function(my\_vin) {  this.vinCode = my\_vin;  alert("Vin code:" + this.vinCode);  } var c1 = new Car(); c1.setVin("vin1"); var c2 = new Car(); c2.setVin("vin2"); |

При вызове метода setVin каждого из созданных экземпляров класса Car мы получим разный результат. Этим мы демонстрируем, что данные о состоянии конкретного объекта инкапсулированы внутри класса.

**Полиморфизм** в JS также завязан на prototype. Все методы и свойства в классе определяются через свойство prototype, а различные классы могут определять методы с одинаковыми именами. Проще говоря, у всех объектов в JS есть первичный предок – класс Object. От него все классы наследуются по умолчанию. При наследовании создаются классы, например, Car и Plane, но и у Car, и у Plane скорее всего будет метод move(), который будет описывать перемещение. Однако поведение Car и Plane при выполнении метода move() разное. Это и есть полиморфизм.

# Практика

Подойдем ближе к основной теме наших занятий – реализации сайта. Реализуем при помощи ООП некий HTML-блок, а затем унаследуем от него сущность меню.

Наш основной класс будет называться Container:

|  |
| --- |
| function Container() {  this.id = "";  this.className= "";  this.htmlCode = ""; } Container.prototype.render = function() {  return this.htmlCode; } |

Мы создали основной блок – высокоуровневую абстракцию, от которой будут наследоваться другие конкретные узконаправленные блоки. Все они будут иметь метод render, который возвращает готовый HTML-код, способный к встраиванию прямо в страницу.

|  |
| --- |
| function Menu(my\_id, my\_class, my\_items){  Container.call(this);  this.id = my\_id;  this.className = my\_class;  this.items = my\_items; } Menu.prototype = Object..create(Container.prototype); Menu.prototype.constructor = Menu; Menu.prototype.render = function(){  } var menu = new Menu("my\_menu", "menu\_class", {}); console.log(menu.render()); |

Мы чуть обновили работу с меню, создав конкретный класс-потомок, реализующий метод меню. Также мы предусмотрели свойство для хранения пунктов меню в виде объекта и сразу же переопределили метод вывода на экран, но пока наш функционал ничего особенного не делает.

Пункт меню – это также сложный объект, который содержит в себе имя пункта, ссылку на страницу сайта. Стоит вынести его в отдельную сущность.

|  |
| --- |
| function MenuItem(my\_href, my\_name){  Container.call(this);  this.className = "menu-item";  this.href = my\_href;  this.name = my\_name; } MenuItem.prototype = Object.create(Container.prototype); MenuItem.prototype.constructor = MenuItem; MenuItem.prototype.render = function(){  return "<li class='"+this.className+"' href='"+ this.href +"'>" + this.itemName + "</li>"; } var m\_item1 = new MenuItem("/", "Главная"); var m\_item2 = new MenuItem("/catalogue/", "Каталог"); var m\_item3 = new MenuItem("/gallery/", "Галерея"); var m\_items = {0: m\_item1, 1: m\_item2, 2: m\_item3}; |

Теперь мы можем передать пункты меню в само меню и приступить непосредственно к рендерингу элемента. Для этого нужно обойти все элементы меню и сгенерировать их HTML-код.

|  |
| --- |
| Menu.prototype.render = function(){  var result = "<ul class='"+this.className+"' id='"+this.id+"'>";  for(var item in this.items){  if(this.items[item] instanceof MenuItem){  result += this.items[item].render();  }  }  result += "</ul>";  return result; } |

Обратите внимание на особенность цикла for...in. Он позволяет по порядку обойти свойства объекта с начала до конца. На каждой его итерации мы задаем переменную item, в которую помещаем имя свойства объекта.

После этого мы проверяем, является ли поданный на вход элемент экземпляром класса MenuItem. Если это так, мы вызываем метод render уже у MenuItem, т.к. он тоже является наследником родительского класса Container.

В конце мы просто добавим полученный элемент прямо в body:

|  |
| --- |
| var menu = new Menu("my\_menu", "My\_class", m\_items); var div = document.write(menu.render()); |

В результате получим примерно следующее:



# Вывод

В данном примере мы создали прототип для создания различных меню на нашем сайте.

Также мы создали родительский прототип для создания тех или иных элементов сайта в будущем, чем помогли себе ускорить разработку, создав элемент MenuItem.

Подобным подходом мы используем все три основополагающих принципа ООП. Мы наследуемся от родительского прототипа, инкапсулируем в каждом конкретном классе свои специфические данные и переопределяем метод render для каждой дочерней реализации родительского класса.

Мы четко разделили внешний и внутренний интерфейсы, позволяя будущим разработчикам, которые будут работать с нашим кодом, довольно свободно (но не менее аккуратно!) менять внутренние свойства и методы.

Также мы делаем использование кода настолько простым, насколько это возможно. Все любят пользоваться функциональными вещами, простыми на первый взгляд.

# Домашнее задание

1. Улучшить базовый класс, добавив в него общий для всех метод remove(), который удаляет соответствующий DOM-узел.
2. Создать наследника класса Menu – новый класс должен уметь строить меню со вложенными пунктами, т.е с подменю. Подсказка: главный секрет в обходе объекта пунктов меню и проверке типов.
3. \* Некая сеть фаст-фудов предлагает несколько видов гамбургеров:

* маленький (50 рублей, 20 калорий);
* большой (100 рублей, 40 калорий).

Гамбургер может быть с одним из нескольких видов начинок (обязательно):

* с сыром (+ 10 рублей, + 20 калорий);
* с салатом (+ 20 рублей, + 5 калорий);
* с картофелем (+ 15 рублей, + 10 калорий).

Дополнительно гамбургер можно посыпать приправой (+ 15 рублей, 0 калорий) и полить майонезом (+ 20 рублей, + 5 калорий).

Напишите программу, рассчитывающую стоимость и калорийность гамбургера.

Используйте ООП подход (подсказка: нужен класс Гамбургер, константы, методы для выбора опций и расчета нужных величин).

|  |
| --- |
| /\*\* \* Класс, объекты которого описывают параметры гамбургера.  \*  \* @constructor \* @param size Размер \* @param stuffing Начинка \* @throws {HamburgerException} При неправильном использовании \*/ function Hamburger(size, stuffing) { ... }  /\* Размеры, виды начинок и добавок \*/ Hamburger.SIZE\_SMALL = ... Hamburger.SIZE\_LARGE = ... Hamburger.STUFFING\_CHEESE = ... Hamburger.STUFFING\_SALAD = ... Hamburger.STUFFING\_POTATO = ... Hamburger.TOPPING\_MAYO = ... Hamburger.TOPPING\_SPICE = ... /\*\* \* Добавить добавку к гамбургеру. Можно добавить несколько \* добавок, при условии, что они разные. \*  \* @param topping Тип добавки \* @throws {HamburgerException} При неправильном использовании \*/ Hamburger.prototype.addTopping = function (topping) ... /\*\*  \* Убрать добавку, при условии, что она ранее была   \* добавлена.  \*   \* @param topping Тип добавки  \* @throws {HamburgerException} При неправильном использовании  \*/ Hamburger.prototype.removeTopping = function (topping) ... /\*\*  \* Получить список добавок.  \*  \* @return {Array} Массив добавленных добавок, содержит константы  \* Hamburger.TOPPING\_\*  \*/ Hamburger.prototype.getToppings = function () ... /\*\*  \* Узнать размер гамбургера  \*/ Hamburger.prototype.getSize = function () ... /\*\*  \* Узнать начинку гамбургера  \*/ Hamburger.prototype.getStuffing = function () ... /\*\*  \* Узнать цену гамбургера  \* @return {Number} Цена в тугриках  \*/ Hamburger.prototype.calculatePrice = function () ... /\*\*  \* Узнать калорийность  \* @return {Number} Калорийность в калориях  \*/ Hamburger.prototype.calculateCalories = function () ... /\*\*  \* Представляет информацию об ошибке в ходе работы с гамбургером.   \* Подробности хранятся в свойстве message.  \* @constructor   \*/ function HamburgerException (...) { ... } |

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Mozilla Developer Network](https://developer.mozilla.org/).
2. [MSDN](http://msdn.microsoft.com/).
3. [Safari Developer Library.](https://developer.apple.com/library/safari/navigation/index.html)
4. [Современный учебник JavaScript.](http://learn.javascript.ru)