Урок 2

ООП в JavaScript

Основные принципы объектно-ориентированного программирования и его реализация в JavaScript.

Долгое время в программировании применялся процедурный подход. Он заключается в написании большого количества независимых подпрограмм, каждая из которых решает определённую задачу. Данные обрабатываются в одной подпрограмме, передаются в другую и так далее.

Со временем появилась другая концепция, сосредоточенная не на действиях, алгоритмах и функциях, а на объектах – сущностях, которые обладают некоторыми свойствами и умеют выполнять определённые действия. Такой подход назвали объектно-ориентированным программированием, или ООП.

Главное отличие объектно-ориентированного подхода от процедурного – необходимость заранее и внимательно продумывать архитектуру программы. Нужно сразу понимать, какие модули и как будут работать с данными. Благодаря этому приложение становится легче расширять и поддерживать

Объекты и классы

Объект – это сущность, обладающая особенным поведением и отличительными характеристиками.

Для создания объектов используются **классы** – специальные шаблоны, которые определяют, какими свойствами будет обладать объект и что он умеет делать. Пример класса – автомобиль. Существует много разных комплектаций, марок и типов кузова, но по сути все они похожи, разница лишь в незначительных деталях.

Каждый объект обладает состоянием, которое описывается набором свойств. Как правило, этот набор не меняется от одного родственного объекта к другому. Изменяются лишь значения этих свойств.

Другая важная характеристика объекта – поведение. Она показывает, как объект изменяется и как взаимодействует с другими объектами. Обычно все действия объекта направлены либо на изменение его состояния, либо на передачу данных другим объектам.

Попробуем представить, как это работает в программировании. Пока что без сложных примеров и JavaScript-кода – просто чтобы понять концепцию.

Допустим, у нас есть меню, состоящее из пунктов. Создадим класс **ПунктМеню** и опишем его основные свойства. Каждый пункт меню как-то называется. Это значит, что у него есть свойство **название**. А ещё наш пункт меню может, например, менять цвет. Зададим классу ещё и свойство **цвет**:

```
класс ПунктМеню
  свойства
    цвет
    название
```

Теперь сделаем, чтобы все пункты меню умели что-нибудь делать. Например, менять цвет на красный. Добавим классу метод **Покраснеть**:

```
класс ПунктМеню
  свойства
    цвет
    название
  методы
    Покраснеть
```

Теперь мы можем создать с помощью этого шаблона любое количество объектов, каждый из которых будет иметь свойства **цвет**, **название** и метод **Покраснеть**. Такие объекты будут называться экземплярами класса **ПунктМеню**.

Это самый простой пример того, как работает ООП. Дальше можно усложнять. Например, класс можно создавать не с нуля, а на основе уже существующего. Тогда говорят, что один класс наследуется от другого. Класс-наследник обладает всеми свойствами и методами класса-родителя, но в него можно добавить какие-то новые свойства и методы или переопределить старые.

Наследование – один из трёх главных принципов ООП. Два других – инкапсуляция и полиморфизм.

Инкапсуляция – возможность объединить все свойства и методы и при желании скрыть часть из них. Это значит, что в классе содержится полный набор свойств и методов, необходимых для его работы. При этом мы можем сделать так, чтобы некоторые из них были недоступны напрямую у экземпляров класса. Так обычно делают для служебных свойств и методов, которые используются для внутренних преобразований данных в классе.

Полиморфизм – возможность объявить у разных классов один и тот же метод или свойство. Например, создадим на основе нашего класса **ПунктМеню** два класса-наследника – **ПунктГлавногоМеню** и **ПунктБоковогоМеню**. Для пункта главного меню сделаем так, чтобы при вызове метода **Покраснеть** становился красным только текст, а для бокового меню – текст и рамка вокруг него. Метод называется одинаково, но в зависимости от того, к какому пункту мы его применяем, будут выполняться разные действия.

Создание объектов

Теперь посмотрим, как работает ООП JavaScript. Самый простой способ создания объекта – записать его значение в переменную. Используем наш пример с пунктом меню:

```
const menuItem = {
  color: 'blue',
  name: 'Bar'
}
```

Теперь в переменной **menuItem** хранится объект, обладающий двумя свойствами – **color** и **name**. Можно добавить ему действие, т. е. метод:

```
const menuItem = {
  color: 'blue',
  name: 'Bar',
  makeRed: function () {
    this.color = 'red';
  }
}
```

Теперь мы можем вызвать метод через **menuItem.makeRed()**, и значение свойства **color** изменится на **red**.

Разумеется, в меню может быть довольно много пунктов, и создавать каждый объект вручную долго и неудобно, поэтому используем полученные знания и объявим класс **MenuItem**. В JavaScript классы реализованы через функции-конструкторы. Это обычные функции, но с их помощью можно создавать

объекты, используя ключевое слово **new**.

```
function MenuItem() {
  this.color = 'blue',
  this.name = 'Bar',
  this.makeRed = function () {
    this.color = 'red';
  }
}
const item1 = new MenuItem();
const item2 = new MenuItem();
```

Теперь в переменных **item1** и **item2** содержатся объекты, обладающие свойствами **color** и **name** и методом **makeRed()**. Но сейчас оба объекта одинаковые, поэтому добавим возможность задавать свойства прямо при создании объекта. Для этого будем передавать значения как аргументы функции-конструктора:

```
function MenuItem(color, name) {
  this.color = color,
  this.name = name,
  this.makeRed = function () {
    this.color = 'red';
  }
}
const item1 = new MenuItem('black', 'Foo');
const item2 = new MenuItem('white', 'Bar');
```

Теперь каждый объект, созданный через функцию-конструктор, будет иметь разные свойства **color** и **name**.

```
console.log(item1.name);    // Foo
console.log(item2.name);    // Bar
```

Любому из созданных объектов можно добавить новое свойство напрямую, но на другие экземпляры класса **MenuItem** это никак не повлияет.

```
item1.width = 200;

console.log(item1.width)  // 200
console.log(item2.width)  // undefined
```

Если нужно добавить свойство всем экземплярам класса, то нужно добавить его в функцию-конструктор и задать значение при создании объекта:

```
function MenuItem(color, name, width) {
  this.color = color;
  this.name = name;
  this.width = width;
  this.makeRed = function () {
```

```
    this.color = 'red';
  }
}
```

Наследование

Мы записали метод **makeRed** в функции-конструкторе так же, как и свойства – напрямую. Такой подход возможен, но чаще в JavaScript для хранения методов класса есть специальное поле **prototype**. С его помощью можно реализовать наследование. Перепишем класс и запишем метод через **prototype**:

```
function MenuItem(color, name, width) {
  this.color = color;
  this.name = name;
  this.width = width;
}

MenuItem.prototype.makeRed = function () {
  this.color = 'red';
}

const item = new MenuItem('white', 'Foo', 200);
item.makeRed();
console.log(item);  // { color: 'red', name: 'Foo', width: 200 }
```

Всё работает. Теперь попробуем создать новый класс **MainMenuItem**, унаследовав его от класса **MenuItem**. Для начала создадим функцию-конструктор:

```
function MainMenuItem(color, name, width) {
  this.color = color;
  this.name = name;
  this.width = width;
}
```

Как видно, она делает то же самое, что и функция-конструктор **MenuItem**. Чтобы не дублировать код, вызовем **MenuItem**, передав ей соответствующий контекст и параметры:

```
function MainMenuItem(color, name, width) {
  MenuItem.call(this, color, name, width);
}
const mainItem = new MainMenuItem('black', 'Bar', 100);
console.log(mainItem);  // { color: 'black', name: 'Bar', width: 100 }
```

Если сейчас мы вызовем **mainItem.makeRed()**, получим ошибку, потому что в **prototype** класса **MainMenuItem** нет такого метода. Скопируем **prototype** родительского класса:

```
MainMenuItem.prototype = Object.create(MenuItem.prototype);
```

Object.create – ещё один способ создания объектов. Он позволяет создавать объект, копируя всю его цепочку прототипов. Теперь **mainItem.makeRed()** сработает. Но нужно помнить ещё В **prototype** есть поле **constructor**. В нём содержится функция-конструктор. Раз мы скопировали **prototype** у **MenuItem**, то сейчас там записана функция-конструктор **MenuItem**. Запишем туда правильное значение:

```
MainMenuItem.prototype.constructor = MainMenuItem;
```

Теперь всё в порядке.

Классы в ES2015

В стандарте ES2015 появился удобный способ создания и наследования классов:

```
class MenuItem {
  // описание класса
}
```

При объявлении класса можно сразу описать функцию-конструктор. В ней можно в качестве аргументов передать значения, которые запишутся в качестве свойств:

```
class MenuItem {
  constructor(color, name) {
    this.color = color;
    this.name = name;
  }
}
```

Теперь при создании экземпляра класса **MenuItem** автоматически будет срабатывать функция **constructor**. Кроме неё в класс можно добавить и другие методы:

```
class MenuItem {
  constructor(color, name) {
    this.color = color;
    this.name = name;
  }
  makeRed() {
    this.color = 'red';
  }
}
```

Теперь можно не беспокоиться о **prototype** и просто указывать методы. Они сразу будут наследоваться как надо. Кстати, о наследовании. Чтобы указать родителя при объявлении класса, используется ключевое слово **extends**:

```
class MainMenuItem extends MenuItem {
  makeBlue() {
    this.color = 'blue';
  }
}
```

Теперь экземпляры класса **MainMenuItem** будут создаваться через ту же функцию-конструктор, и у них будет метод **makeBlue()**.

Важно помнить, что в ES2015 класс нельзя вызвать без слова **new**, это вызовет ошибку.

Практика

Продолжаем работать с нашим интернет-магазином. Как говорилось выше, ООП предполагает продумывание архитектуры, поэтому для начала решим, с какими сущностями мы будем работать.

Прежде всего у нас есть товар, создадим для него класс **GoodsItem**. На текущем этапе у него будет два свойства – **title** и **price** – и один метод **render,** который будет возвращать html-разметку. Этот метод мы уже описали на прошлом уроке.

```
class GoodsItem {
  constructor(title, price) {
    this.title = title;
    this.price = price;
  }
  render() {
    return `<div
class="goods-item"><h3>${this.title}</h3><p>${this.price}</p></div>`;
  }
}
```

Вторым классом будет список товаров **GoodsList**. В качестве свойства добавим массив со списком товаров, но изначально сделаем его пустым:

```
class GoodsList {
  constructor() {
    this.goods = [];
  }
}
```

Теперь нам нужно сделать метод для заполнения списка. Позже мы будем получать данные с сервера, поэтому создадим метод **fetchGoods**, но пока запишем в массив **goods** статичный список товаров:

```
class GoodsList {
  constructor() {
    this.goods = [];
  }
  fetchGoods() {
    this.goods = [
      { title: 'Shirt', price: 150 },
      { title: 'Socks', price: 50 },
      { title: 'Jacket', price: 350 },
      { title: 'Shoes', price: 250 },
    ];
```

```
  }
}
```

Ещё один метод – вывод списка товаров. Создадим для этого действия метод **render()**. Для каждого элемента массива **goods** будем создавать экземпляр класса **GoodsItem** и запрашивать его разметку.

```
class GoodsList {
  constructor() {
    this.goods = [];
  }
  fetchGoods() {
    this.goods = [
      { title: 'Shirt', price: 150 },
      { title: 'Socks', price: 50 },
      { title: 'Jacket', price: 350 },
      { title: 'Shoes', price: 250 },
    ];
  }
  render() {
    let listHtml = '';
    this.goods.forEach(good => {
      const goodItem = new GoodsItem(good.title, good.price);
      listHtml += goodItem.render();
    });
    document.querySelector('.goods-list').innerHTML = listHtml;
  }
}
```

Теперь, чтобы вывести список, нужно создать экземпляр класса **GoodsList**, вызвать для него метод **fetchGoods**, чтобы записать список товаров в свойство **goods**, и вызвать **render()**.

```
const list = new GoodsList();
list.fetchGoods();
list.render();
```

Практическое задание

1. Добавьте пустые классы для Корзины товаров и Элемента корзины товаров. Продумайте, какие методы понадобятся для работы с этими сущностями.
2. Добавьте для **GoodsList** метод, определяющий суммарную стоимость всех товаров.
3. * Некая сеть фастфуда предлагает несколько видов гамбургеров:

    a. Маленький (50 рублей, 20 калорий).
    b. Большой (100 рублей, 40 калорий).

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

    a. С сыром (+10 рублей, +20 калорий).
    b. С салатом (+20 рублей, +5 калорий).

c. 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. Можно использовать примерную архитектуру класса со следующей страницы, но можно использовать и свою.

```
class Hamburger {
  constructor(size, stuffing) { ... }
  addTopping(topping) {     // Добавить добавку }
  removeTopping(topping) { // Убрать добавку }
  getToppings(topping) {    // Получить список добавок }
  getSize() {               // Узнать размер гамбургера }
  getStuffing() {           // Узнать начинку гамбургера }
  calculatePrice() {        // Узнать цену }
  calculateCalories() {     // Узнать калорийность }
}
```

Дополнительные материалы

1. ООП в функциональном стиле.
2. ООП в прототипном стиле.

Используемая литература

1. Современный учебник Javascript.
2. Alberto Montalesi. The Complete Guide to Modern JavaScript.