**Объектно-ориентированное программирование. Введение**

Когда код разрастается, с ним становится сложно работать: добавлять функциональность, исправлять ошибки. Вы сами могли это заметить, когда работали над проектом.

Раньше программисты просто писали код — лишь бы компьютер понял. Со временем они осознали: чтобы работать с кодом, его необходимо как-то организовать. Код — не просто последовательность команд. Его можно и нужно объединять в логические блоки.

И разработчики придумали, как это сделать. Так появились парадигмы программирования — совокупности правил, которым вы следуете, когда пишете код.

Самая популярная парадигма — объектно-ориентированное программирование (ООП).

ООП — как раз совокупность правил и принципов, которые используют программисты, когда пишут программы. Это не единственная концепция создания кода, но наиболее распространённая. Поэтому расскажем об ООП.

Каким должен быть код по принципам ООП:

1. Используется множество объектов — они взаимодействуют между собой и образуют единую структуру, которой и является любая программа. Объекты хранят данные и взаимодействуют между собой: вызывают методы друг друга и передают в них информацию.
2. Соблюдаются принципы — инкапсуляция, наследование и полиморфизм. О них мы расскажем в последних уроках темы.

А в первых уроках мы разберём основы ООП. Вы узнаете, как работать с объектами и их методами, и познакомитесь с понятием класса.

**Объекты в ООП**

Всё вокруг можно рассматривать как объекты. Компьютер, на котором открыт курс, — объект. У него есть свойства: цвет, размер, яркость экрана. И методы: включить, выключить, нажать клавишу. Свойства и методы объекта часто связаны. Так нажатие кнопки увеличения громкости влияет на свойство «громкость».

Программа — тоже объект. Поэтому песню в плейлисте, [который вы делали в теме о DOM](https://praktikum.yandex.ru/trainer/web/lesson/16436625-8e57-4a67-8f2d-5ecf8255b504/task/308a95fb-3053-4d6a-b9ef-6e87b1e691ad/), можно описать объектом:

Скопировать кодJAVASCRIPT

const song = {

title: 'На заре',

artist: 'Альянс',

isLiked: false,

like: function () {

song.isLiked = !song.isLiked;

}

};

Объект хранит данные в свойствах title, artist и isLiked. Ещё у него есть метод like — это функциональность объекта. Данные и функциональность заключены в единую сущность и представлены в удобной форме.

В интернете их часто называют «свойствами» и «поведением», которыми в объектах выступают переменные и функции:

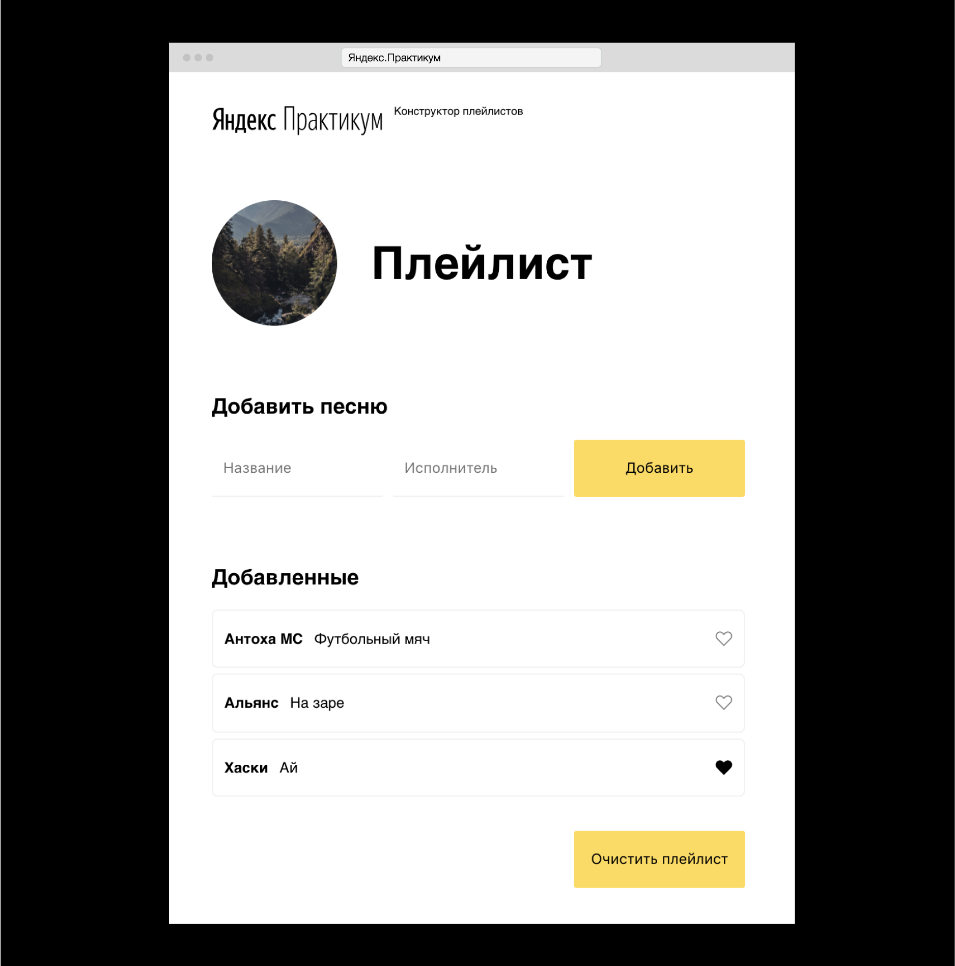
1. Свойства объекта — это переменные, его атрибуты. Ещё их называют полями объекта. В коде выше — это song.title, song.artist и song.isLiked.
2. Поведение объекта — это его функции. Их называют методами объекта. В коде выше song.like — метод объекта song.

Основная идея ООП — данные и функциональность программы объединены в объекты.

# Данные и функциональность вместе

В прошлом уроке мы говорили, что объекты позволяют хранить данные и функциональность программы вместе.

Вновь обратимся к примеру с плейлистом:



Тот самый плейлист

У объекта song строгая структура атрибутов: свойств и методов. Если мы хотим создать ещё одну песню, то её новый объект будет с аналогичной структурой:

Скопировать кодJAVASCRIPT

const song = {

title: 'Diary of Jane',

artist: 'Breaking Benjamin',

isLiked: false,

like: function () {

song.isLiked = !song.isLiked;

}

}

У объекта song есть свойства title, artist и isLiked — данные, которые он хранит. И метод like — функциональность, которая есть у песни.

Объект song — это способ объединить данные песни и её функциональность в одну сущность.

Но в плейлисте обычно несколько песен. Сделаем объект для каждой:

Скопировать кодJAVASCRIPT

const song1 = {

title: 'Футбольный мяч',

artist: 'Антоха MC',

isLiked: false,

like: function () {

song1.isLiked = !song1.isLiked;

}

};

const song2 = {

title: 'На заре',

artist: 'Альянс',

isLiked: false,

like: function () {

song2.isLiked = !song2.isLiked;

}

};

const song3 = {

title: 'Ай',

artist: 'Хаски',

isLiked: true,

like: function () {

song3.isLiked = !song3.isLiked;

}

};

Мы создали три объекта вручную. Но если в плейлисте тысяча песен, придётся вручную объявить тысячу таких переменных.

Вы уже знаете, что функции позволяют избавиться от таких повторений. Напишем функцию createSong, чтобы создавать новые объекты песен:

Скопировать кодJAVASCRIPT

*/\* объявляем функцию createSong,*

*она будет возвращать новые объекты песен \*/*

function createSong(title, artist) {

*// создаём новый объект песни\**

const newSong = {

title,

artist,

isLiked: false,

like: function () {

newSong.isLiked = !newSong.isLiked;

}

}

return newSong; *// возвращаем этот объект*

}

*// теперь создавать объекты песен гораздо проще*

const song1 = createSong('Футбольный мяч', 'Антоха MC');

const song2 = createSong('На заре', 'Альянс');

const song3 = createSong('Ай', 'Хаски');

*// тестируем, как данные и функциональность работают вместе*

console.log(song1.isLiked); *// false*

song1.like();

console.log(song1.isLiked); *// true*

При каждом вызове createSong возвращается новый объект песни. Все треки независимы друг от друга: это разные объекты.

Так удобнее создавать песни, но у подхода есть проблема: каждый объект содержит свою собственную функцию like. И если мы добавим функциональность редактирования или удаления песни, она будет жить в отдельных объектах.

Ещё один важный недостаток — расход памяти. Если у каждого объекта своя функция, их все придётся хранить в оперативной памяти. И каждый созданный объект потребует отдельные ячейки памяти для своей функции. Поскольку эти функции выполняют похожие операции и мало чем отличаются друг от друга, создавать их множество нецелесообразно.

Но можно поступить иначе. В следующем уроке мы расскажем, как хранить общую функциональность в одном месте, а в объектах песен — только то, что принадлежит конкретному треку.

# Краткое погружение в this

В прошлом уроке для каждого объекта создавалась и хранилась в памяти функция like. И каждая из них — самостоятельная: song1.like === song2.like // false.

Это ресурсозатратно. Каждая функция like выполняет похожее действие: заменяет состояние поля isLiked своего объекта на противоположное. Поэтому логичнее создать одну функцию like и позволить ей изменять свойства любого объекта, который вызывает её как метод.

Для начала объявим like вне контекста функции createSong и передадим её новым объектам по ссылке:

Скопировать кодJAVASCRIPT

function like() {

}

function createSong(title, artist) {

const newSong = {

title,

artist,

isLiked: false,

like: like

}

return newSong;

}

Теперь у объектов в атрибуте like будет ссылка на единственную функцию, к которой у них есть доступ: song1.like === song2.like //true.

Мы не случайно убрали код из тела like. До этого каждая новая функция like ссылалась на свой объект newSong. Поэтому возникает вопрос: как сделать её универсальной, как позволить like работать со всеми объектами, которые создаёт функция createSong?

Чтобы решить эту проблему, нужно познакомиться с this.

Свойство this — это ключевое слово, которое доступно внутри любой функции. В зависимости от способа вызова функции this принимает разные значения. Мы разберём все 5 возможных вариантов в следующем спринте.

А начнём с нашего примера:

Скопировать кодJAVASCRIPT

function like() {

this.isLiked = !this.isLiked;

}

function createSong(title, artist) {

const newSong = {

title,

artist,

isLiked: false,

like: like

}

return newSong;

}

const song1 = createSong('Футбольный мяч', 'Антоха MC');

song1.like(); *// внутри функции like this — это song1*

console.log(song1.isLiked); *// true — сработало*

Здесь this — объект, на котором вызвана функция. Если функцию вызывают как метод объекта, свойство this хранит ссылку на объект, на котором она вызвана:

Скопировать кодJAVASCRIPT

const obj = {

prop: 'Свойство',

method: function () {

console.log(this); *// выведем значение this*

}

}

*// { prop: "Свойство", method: ƒ } — это и есть объект obj, на котором была вызвана функция method.*

obj.method();



В консоль выводится obj — объект, на котором вызван метод method

В нашем примере функция like вернёт значение свойства isLiked объекта song1, на котором она вызвана как метод:

Скопировать кодJAVASCRIPT

*/\* здесь внутри функции like*

*значением this будет объект song1,*

*так как метод вызван именно на нём \*/*

song1.like();

Значение this зависит только от того, на каком объекте вызвана функция, а не в каком объекте она хранится.

Подумайте, каким будет значение this внутри функции like здесь:

Скопировать кодJAVASCRIPT

song2.like();

song3.like();

Ещё раз: если функция вызвана как метод объекта, то в момент вызова значение this внутри неё — это объект, на котором она вызвана.

Поэтому ответ таков:

Скопировать кодJAVASCRIPT

song2.like(); *// здесь this будет song2*

song3.like(); *// а здесь this будет song3*

Три урока теории позади — начинаем практику.

# Классы

В прошлом уроке мы разобрали, как создавать объекты с уникальными данными так, чтобы их общая функциональность хранилась в одном месте и была доступна всем этим объектам. Этот способ экономит память во время работы приложения.

Но похожего результата можно добиться классами. Об этом и расскажем.

# Ключевое слово class

Класс — это описание того, какие данные и функциональность будут у объекта. Это своего рода чертёж, на основании которого создают объекты — экземпляры данного класса. Класс устанавливает в свои экземпляры свойства (данные) и методы (функциональность).

До этого мы создавали объект song вызовом функции createSong. Теперь сделаем это через синтаксис классов, который появился в JS с выходом стандарта ES6.

Чтобы объявить класс, используем ключевое слово class. После него напишем имя класса — переменную, к которой обратимся, если захотим его использовать.

В языках программирования классы принято называть с заглавной буквы. Если поступить иначе, всё будет работать, но лучше придерживаться общепринятых правил. Это поможет позже выявить переменные, которые содержат класс:

Скопировать кодJAVASCRIPT

class Song {

constructor(title, artist) {

this.title = title;

this.artist = artist;

}

like() {

this.isLiked = !this.isLiked;

}

}

const song = new Song('Start Over', 'Any Given Day');

На первый взгляд, между функцией createSong и классом Song нет принципиальных отличий. Они вскроются позже, когда мы разберём наследование — одну из основных парадигм ООП.

# Метод constructor

Любой класс содержит метод constructor, который вызывается при создании нового объекта этого класса. Метод нужен, чтобы заполнить будущий объект данными.

Запись new Song('Start Over', 'Any Given Day') возвращает новый объект. Ключевое слово new важно в вызове класса. Оно означает, что результат этого вызова возвращает новый объект — экземпляр класса, который вызывают. Попытка сделать это без new приведёт к ошибке: «Конструктор класса нельзя вызвать без ключевого слова new».

Свойство this в методе constructor — ссылка на этот возвращаемый объект. Переданные параметры title и artist записываются в свойства объекта this.title и this.artist, который мы заботливо сохраняем в переменную song.

Необязательно описывать метод constructor внутри класса. Тогда метод будет создан неявно и с пустым телом: constructor() {}. После этого никакие данные не будут добавляться в создаваемые объекты.

Но в классе могут быть и другие методы. К ним также имеют доступ все объекты, которые созданы через new Song().

Скопировать кодJAVASCRIPT

const song1 = new Song('Start Over', 'Any Given Day');

const song2 = new Song('Bitter End', 'The Veer Union');

song1.like === song2.like *// true*

Все объекты, которые созданы через класс Song, делят между собой метод like. И этот метод хранится в памяти в единственном экземпляре. Поэтому использование классов решает проблему расходования памяти, о которой мы говорили в прошлых уроках.

class PodcastEpisode {

constructor(title, artist, guest, duration) {

this.title = title;

this.artist = artist;

this.isLiked = false;

this.guest = guest;

this.duration = duration;

}

like() {

this.isLiked = !this.isLiked;

}

getFormattedDuration() {

const minutes = Math.floor(this.duration / 60); *// целое количество минут*

const seconds = this.duration % 60; *// остаток от деления на 60*

return `${minutes}:${seconds > 9 ? seconds : '0' + seconds}`

}

getEpisodeInfo() {

return `${this.artist}. "${this.title}" - ${this.guest} (${this.getFormattedDuration()} сек.)`;

}

# Инкапсуляция

Мы живём в мире гаджетов, но не обязаны в них разбираться. Не важно, какие процессы выполняются в компьютере при запуске — достаточно нажать кнопку ″Power″, чтобы он заработал. Поэтому с точки зрения программирования кнопка включения — часть интерфейса ПК.

Это и есть инкапсуляция — когда внутренняя реализация устройства скрыта от внешнего мира. Пользователю просто и удобно управлять таким гаджетом: вся его сложность спрятана «под капотом».

# Пользовательский интерфейс

Разработчики часто говорят о «пользовательском интерфейсе» — визуальном представлении программы: меню, кнопках, переключателях, полях ввода и др. Он нужен, чтобы любой желающий мог с ней взаимодействовать. В итоге получается прослойка между программным кодом и пользователем.

У каждого сложного устройства должен быть пользовательский интерфейс, чтобы сделать его удобнее в использовании. Завести автомобиль ключом зажигания гораздо проще, чем открыть капот и пытаться разобраться, как запустить двигатель.

# Принцип интерфейсов в ООП

К объектам в ООП-коде применим тот же принцип. Внешние объекты интерфейса — его публичные методы, которые можно смело вызывать во внешнем коде.

У объектов есть и приватные методы и свойства. Их отличие от публичных в том, что их не используют во внешнем коде — они только для внутренней реализации объекта. Мы подробно это разберём в примерах ниже.

Только недавно приватные методы и свойства были реализованы в JS на уровне языка. Раньше их только номинально считали приватными или как-нибудь эмулировали эту функциональность.

Мы расскажем об эмуляции: она более распространена

.

# Приватные методы и свойства

Между разработчиками существует негласное соглашение — именовать приватные свойства и методы знаком нижнего подчёркивания \_. Так все поймут, что свойство внутреннее и использовать его напрямую во внешнем коде нежелательно:

Скопировать кодJAVASCRIPT

class Car {

constructor(maxGasTankValue, fuelConsumption) {

this.\_gasTankValue = 0;

this.\_maxGasTankValue = maxGasTankValue;

this.\_fuelConsumption = fuelConsumption; *// литров на 100км*

}

\_getAvailableGasValue(gasValue) {

if (gasValue < 0) return 0;

if (gasValue > this.\_maxGasTankValue) return this.\_maxGasTankValue;

return gasValue;

}

refuel(gasValue) {

this.\_gasTankValue = this.\_getAvailableGasValue(gasValue);

}

getDistance() {

return this.\_gasTankValue / this.\_fuelConsumption \* 100;

}

}

const car = new Car(70, 9);

car.refuel(45);

console.log(car.\_gasTankValue); *// 45. Свойство на самом деле не приватное. Его можно легко изменить*

console.log(car.getDistance()); *// 500*

В этом примере ничто не мешает установить отрицательное количество бензина выражением car.\_gasTankValue = -1. Такое прямое вмешательство во внутреннюю реализацию приведёт к ошибке в работе объекта.

Но таких ошибок можно избежать. Когда работаете с объектом класса, не используйте приватные методы и не обращайтесь, и ничего не записывайте в приватные свойства извне.

# Действительно приватные свойства и методы

Можно создавать и действительно приватные методы и свойства. Для этого используется символ решётки # в названии.

И если \_ нужно лишь для условного обозначения свойства или метода как приватного, то # действительно делает его таковым:

Скопировать кодJAVASCRIPT

class Car {

#gasTankValue

#maxGasTankValue

#fuelConsumption

constructor(maxGasTankValue, fuelConsumption) {

this.#gasTankValue = 0;

this.#maxGasTankValue = maxGasTankValue;

this.#fuelConsumption = fuelConsumption; *// литров на 100км*

}

#getAvailableGasValue(gasValue) {

if (gasValue < 0) return 0;

if (gasValue > this.#maxGasTankValue) return this.#maxGasTankValue;

return gasValue;

}

refuel(gasValue) {

this.#gasTankValue = this.#getAvailableGasValue(gasValue);

}

getDistance() {

return this.#gasTankValue / this.#fuelConsumption \* 100;

}

}

var car = new Car(70, 9);

*// Эта строка приведёт к ошибке. Доступ к приватным свойствам извне отсутствует*

car.#gasTankValue = -10

Новый синтаксис # не поддерживается многими браузерами. Для полноценной работы нужны специальные программные средства. Они трансформируют код в новом синтаксисе в код по старому стандарту. Мы расскажем о # позже, а пока будем использовать \_.

В отличие от приватных полей, обозначенных через \_, действительно приватные поля класса необходимо объявлять отдельно. Это можно сделать перед вызовом constructor().

В примере выше у класса Car два публичных метода. Метод refuel устанавливает количество горючего, getDistance получает дистанцию, которую проедет автомобиль на заранее установленном количестве бензина.

Создадим объект класса:

Скопировать кодJAVASCRIPT

const car = new Car(54, 5.5)

Можно вызвать метод car.refuel(100) заправки автомобиля или метод car.getDistance() возврата количества километров, на которые хватит топлива.

Метод refuel использует приватный метод \_getAvailableGasValue. Он не позволит поместить в приватное же свойство \_gasTankValue:

* отрицательное количество литров бензина;
* количество, которое превышает объём бака автомобиля. Его объём мы установили первым параметром при вызове new Car().

Когда мы используем публичные методы объекта, мы одновременно вызываем приватные методы и читаем или записываем данные в приватные свойства. Но эту информацию знать необязательно: приватные свойства инкапсулированы внутри класса. Часть их данных и функциональности скрыты от внешнего окружения.

Так инкапсуляция представила объект из 3 методов и 3 свойств как интерфейс из 2 публичных методов, с которыми просто работать.

**Наследование**

Наследование в ООП — это возможность создать класс на основе других классов.

Каждый из нас сталкивался с наследованием в биологии — передачей генома из поколения в поколение среди определённого биологического вида.

Принципы наследования нашли применение и в программировании. С наследованием разработчики научились использовать уже написанные классы снова. Так программы избавились от дублирования кода и их стало гораздо проще поддерживать.

Разберём это на примере:

Скопировать кодJAVASCRIPT

class Student {

constructor(name, cohort) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = null;

this.\_trainingDuration = null;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration

}

}

}

Класс Student создаёт объект с информацией о студентах. Его параметры — имя студента и номер когорты, а профессия this.\_profession и продолжительность обучения this.\_trainingDuration равны null.

В прошлом уроке вы узнали правила чтения и изменения значений приватных свойств. Если им следовать, то не получится изменить снаружи свойства this.\_profession и this.\_trainingDuration. У объекта класса Student нет публичных интерфейсов, чтобы изменить эти приватные свойства.

Можно добавить классу параметры и установить эти два свойства через них:

Скопировать кодJAVASCRIPT

class Student {

constructor(name, cohort, profession, trainingDuration) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = profession;

this.\_trainingDuration = trainingDuration;

}

}

Но лучше создавать в классе объект с минимумом параметров.

Что мы тогда имеем:

1. У объекта класса приватные значения свойств.
2. Эти значения прописаны статически.
3. Они не устанавливаются параметрами, которые переданы в конструктор класса.

Поэтому логично создать для каждого типа объекта отдельный класс:

Скопировать кодJAVASCRIPT

*// класс студента веб-разработчика*

class WebDeveloperStudent {

constructor(name, cohort) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration

}

}

}

*// класс студента Пайтон-разработчика*

class PythonDeveloperStudent {

constructor(name, cohort) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = 'Python developer';

this.\_trainingDuration = 9;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration

}

}

}

*// класс студента аналитика данных*

class DataAnalystStudent {

constructor(name, cohort) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = 'Data analyst';

this.\_trainingDuration = 6;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration

}

}

}

Но тогда придётся в каждом классе дублировать:

* метод getInfo и 4 свойства в нём;
* установку значений this.\_name и this.\_cohort;
* изменения getInfo, если они появятся.

Проблему дублирования кода в классах решает наследование.

С ключевыми словами extends можно создать новый класс, который наследует от исходного все его свойства и методы:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

}

class PythonDeveloperStudent extends Student {

}

class DataAnalystStudent extends Student {

}

Ключевое слово extends позволяет этим 3 дочерним классам наследовать методы родительского класса Student.

Но нам нужно присвоить каждому классу уникальные значения свойств \_profession и \_trainingDuration. Поэтому объявим метод constructor в каждом классе и присвоим этим свойствам правильные значения:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

}

class PythonDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Python developer';

this.\_trainingDuration = 9;

}

}

class DataAnalystStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Data analyst';

this.\_trainingDuration = 6;

}

}

В каждом конструкторе дочернего класса мы вызываем ключевое слово super — ссылку на родительский класс. Поэтому вызов super() вызывает метод constructor класса, на который super и ссылается. В нашем случае это класс Student.

Но если класс наследуется через extends и нужно определить в дочернем классе метод constructor — вызываем super и передаём ему нужные свойства, как в примере выше. Иначе появится ошибка и скрипт перестанет выполняться.

С ключевыми словами super и extends можно перенести инициализацию свойств this.\_name и this.\_cohort в конструктор родителя. Так мы избавимся от дублирования этих свойств в конструкторах дочерних классов.

Каждый дочерний класс переопределяет свойства this.\_profession и this.\_trainingDuration своего родительского класса. Поэтому вызов getInfo объектов дочерних классов вернёт значения этих свойств в соответствии с их классами:

Скопировать кодJAVASCRIPT

const student1 = new WebDeveloperStudent("Петя Васечкин", 1);

const student2 = new DataAnalystStudent("Маша Иванова", 3);

student1.getInfo();

*/\**

*{*

*name: "Петя Васечкин",*

*cohort: 1,*

*profession: "Web developer",*

*traningDuration: 10*

*}*

*\*/*

student2.getInfo();

*/\**

*{*

*name: "Маша Иванова",*

*cohort: 3,*

*profession: "Data analyst",*

*traningDuration: 6*

*}*

*\*/*

Переходите к заданиям и проделайте то же самое самостоятельно.

# Полиморфизм

В прошлом уроке мы разбирали пример наследования метода getInfo родительского класса дочерними.

Это очень удобно. Мы один раз объявляем метод и после используем его для всех объектов дочерних классов:

Скопировать кодJAVASCRIPT

class Student {

constructor(name, cohort) {

this.\_name = name;

this.\_cohort = cohort;

this.\_profession = null;

this.\_trainingDuration = null;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration

}

}

}

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

}

class PythonDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Python developer';

this.\_trainingDuration = 9;

}

}

class DesignerStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Designer';

this.\_trainingDuration = 6;

}

}

Здесь всё те же 3 дочерних класса студентов разных профессий, которые наследуются от родительского Student. Ключевое слово super избавляет от необходимости описывать в каждом из них метод getInfo со свойствами this.\_name и this.\_cohort.

Достаточно вызвать super() в конструкторе дочернего класса, чтобы вернуть объект с унаследованными свойствами и методами родительского:

Скопировать кодJAVASCRIPT

const student1 = new WebDeveloperStudent('Петя Васечкин', 1);

const student2 = new PythonDeveloperStudent('Павел Морозов', 2);

const student3 = new DesignerStudent('Маша Иванова', 3);

Напишем функцию, которая примет массив объектов студентов и вернёт массив с информацией о них:

Скопировать кодJAVASCRIPT

function getInfoList(students) {

if (!Array.isArray(students)) return [];

return students.map(student => student.getInfo());

}

getInfoList([student1, student2, student3]);

# Переопределение методов

Теперь добавим поле language в объект информации о студенте. В language находится название языка программирования, который изучает студент на этом курсе.

Веб-разработчику — Javascript, Python-разработчику — Python (вот это поворот!) Будущие дизайнеры не изучают программирование, поэтому реализация метода getInfo класса DesignerStudent отличается от классов WebDeveloperStudent и PythonDeveloperStudent.

Для этого нужно переопределить методы.

Переопределение методов — это возможность дочерних классов не наследовать определённый родительский метод, а реализовать свой собственный.

Перепишем класс WebDeveloperStudent:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration,

language: 'Javascript'

}

}

}

Теперь getInfo объектов дочернего класса WebDeveloperStudent отличается от родительского Student. Этот метод — функция getInfo, которая объявлена в классе WebDeveloperStudent:

Скопировать кодJAVASCRIPT

const studentWithoutProfession = new Student('Павел Чехов', 10);

const student1 = new WebDeveloperStudent('Иван Данко', 11);

const student2 = new WebDeveloperStudent('Наталья Романова', 6);

console.log(student1.getInfo === student2.getInfo) *// true; объекты класса WebDeveloperStudent имеют общий метод.*

console.log(studentWithoutProfession.getInfo === student1.getInfo) *// false; Это разные функции.*

Повторим эту операцию и для класса PythonDeveloperStudent:

Скопировать кодJAVASCRIPT

class PythonDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Python developer';

this.\_trainingDuration = 10;

}

getInfo() {

return {

name: this.\_name,

cohort: this.\_cohort,

profession: this.\_profession,

trainingDuration: this.\_trainingDuration,

language: 'Python'

}

}

}

Теперь объекты дочерних классов WebDeveloperStudent и PythonDeveloperStudent используют собственный метод getInfo, а DesignerStudent — метод getInfo родительского Student:

Скопировать кодJAVASCRIPT

getInfoList([student1, student2, student3]);

# Полиморфизм

Возможность объектов с одинаковым интерфейсом иметь разную реализацию — это и есть полиморфизм. Функция getInfoList поочерёдно вызывает метод getInfo в переданных объектах. Но теперь у каждого объекта своя реализация getInfo, и она зависит от их класса.

В примере выше мы переопределили метод родительского класса. Это привело к избыточному дублированию кода: его основная часть повторяется в каждом дочернем классе. Но это легко исправить, если расширить метод родительского класса.

Внутри метода дочернего класса применим ключевое слово super из прошлого урока. Так мы сможем дальше пользоваться функциональностью родительского класса. Пусть функция this.getInfo() теперь вызывает именно метод getInfo дочернего класса: мы же его переопределили. Но возможность вызывать getInfo родительского никуда не делась.

Используем конструкцию super.getInfo(), чтобы вызвать метод getInfo класса Student:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

getInfo() {

return super.getInfo();

}

}

Конструкция super.getInfo — это метод родительского класса Student, который возвращает объект с информацией о студенте.

Но в плане функциональности это равносильно тому, если бы мы ничего не переопределяли. Мы можем удалить из класса WebDeveloperStudent реализацию метода getInfo — ничего не изменится. Класс WebDeveloperStudent унаследует метод родительского класса Student:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

super(name, cohort);

this.\_profession = 'Web developer';

this.\_trainingDuration = 10;

}

}

const studentWithoutProfession = new Student('Василий Зайцев', 3);

const student1 = new WebDeveloperStudent('Наталья Ченкова', 4);

*// student1 использует метод getInfo родительского класса Student*

console.log(studentWithoutProfession.getInfo === student1.getInfo) *// true*

# Расширение методов

Но это не то, что нам нужно. Мы же хотели в объект информации о студенте добавить язык программирования, который он изучает на курсе.

И вот как это сделать:

Скопировать кодJAVASCRIPT

class WebDeveloperStudent extends Student {

constructor(name, cohort) {

*// ...*

}

getInfo() {

const info = super.getInfo();

info.language = 'Javascript';

return info;

}

}

Теперь метод getInfo класса WebDeveloperStudent свойством language расширяет содержимое объекта с информацией о студенте и возвращает его.

# Объектно-ориентированное программирование. Заключение

Поздравляем! Вы написали свой первый код в объектно-ориентированном стиле. ООП — не единственная парадигма программирования, но пока что самая популярная. Вы ещё не раз столкнётесь с написанным в этом стиле кодом.

Теперь вы знаете об инкапсуляции, наследовании и полиморфизме. Но пока непонятно, как применять это в реальных интерфейсах. Если не терпится узнать — переходите к следующей теме: она как раз об этом.