

ПАВЕЛ АЛЕКСАНДРОВИЧ
ЗАБЕЛИН

**JAVA 2021:
ЛЕГКИЙ
СТАРТ**

Павел Забелин

JAVA 2021: лёгкий старт

«Издательские решения»

Забелин П. А.

JAVA 2021: лёгкий старт / П. А. Забелин — «Издательские решения»,

ISBN 978-5-00-515483-5

Главная цель этой книги — показать читателю, что программирование на Java, гораздо более проще, чем принято об этом думать. Как известно «хочешь лучше понять сам — расскажи об этом другому», что я и попытался сделать на страницах этой книге в меру своих сил и времени. Эта книга как раз вам поможет обрести базовые знания программирования и языка программирования Java, и избавит вас от проблем с пониманием основ программирования.

ISBN 978-5-00-515483-5

© Забелин П. А.

© Издательские решения

Содержание

Введение	7
Для кого эта книга?	8
Почему Java?	9
Что все-таки выбрать?	12
Про зарплаты	13
Часть I	14
Глава 0. Мы программируем	14
Мы программируем и пишем программы	15
Глава 1. Первая программа	16
Настраиваемся на программирование. Устанавливаем IDE	16
Глава 2. Данные	27
Типы данных	27
Переменные	28
Как долго живут переменные?	28
Объектно-ориентированное программирование (ООП)	29
Глава 3. Операции с примитивными типами данных	32
Арифметические операции	32
Преобразование типов	33
Инкремент и декремент	34
Сокращенные арифметические операции	34
Операции сравнения	35
Логические операции	36
Задания	37
Глава 4. Управление выполнением программы. Условия	39
Условный оператор if	39
Условный оператор else	40
Тернарный оператор	41
Оператор выбора switch-case	41
Задания	42
Глава 5. Управление выполнением программы. Циклы	44
Цикл while	44
Цикл do-while	45
Цикл со счетчиком for	46
Задания	48
Глава 6. Управление выполнением программы. Операторы перехода	49
Оператор перехода break	49
Оператор перехода continue	51
Оператор перехода return	52
Глава 7. Массивы	53
Создание одномерного массива	53
Многомерные массивы	54
Задания	55
Глава 8. Ввод данных	57
Заключение к первой части книги	60
Задания	61

Часть II	62
Глава 9. Объектно-ориентированное программирование	62
Классы и объекты – основные понятия	62
Принципы ООП: «Три кита на одной черепахе»	63
Глава 10. Наш зоопарк	65
Трудно ли быть Творцом?	67
Конструктор класса	70
Объект this	73
Пора раскрасить льва	73
Константы	73
Enumerations (перечисления)	74
Задания	75
Глава 11. Инкапсуляция – защищаем наших животных	76
Защищаем льва	76
Геттеры и сеттеры (getters и setters)	76
Глава 12. Расширяем Зоопарк	80
Лебеди – красивые и летают	81
Глава 13. Улучшаем менеджмент в Зоопарке – абстракция и наследование	84
Абстракция и наследование	84
Время ужина	87
Глава 14. Заботимся о каждом – полиморфизм	88
Объектно-ориентированный массив: ArrayList	88
На раз-два отчитайся	89
This is the end?	92
«А много еще учить?»	92
«Как мотивировать себя на изучение?»	92
«Будет ли продолжение?»	92

JAVA 2021: лёгкий старт

Павел Александрович Забелин

Посвящается моей музе Кицунэ ЗЛА

© Павел Александрович Забелин, 2020

ISBN 978-5-0051-5483-5

Создано в интеллектуальной издательской системе Ridero

Введение

Главная цель этой книги – показать читателю, что программирование на Java, гораздо более проще, чем принято об этом думать. Имея за плечами опыт программирования больше 15 лет, я относительно недавно увлекся программированием на Java и эта книга – неоконченная «сжатая» история самообучения. Как известно «хочешь лучше понять сам – расскажи об этом другому», что я и попытался сделать на страницах этой книге в меру своих сил и времени. Я прочитал несколько книг и прошел несколько курсов в интернете: в университете Skillbox, Udemy, Stepik, что и вам советую. Но прежде чем купить какие-либо курсы и начать их проходить, я рекомендую прочесть эту книгу: зачастую курсы грешат провалами в теории и скачками сложности преподаваемого материала, да и сложно определить начальный уровень подготовки студента. Эта книга как раз вам поможет обрести базовые знания программирования и языка программирования Java, и избавит вас от проблем с пониманием основ программирования.

Для кого эта книга?

Эта книга для любого, кто хочет научиться программировать. Программирование только лишь окутано завесой чего-то очень сложного. На своем пути я видел людей абсолютно различных профессий (мало относящихся к компьютерам), которые успешно освоили программирование. Программирование – это очень широкая область деятельности, которая позволяет проявить разнообразные способности и умения. К тому же «побочные эффекты» профессионального программирования, такие как возможность работы без привязки к месту жительства и достойная оплата труда, которая позволит проживать практически в любом уголке планеты, еще больше мотивируют попробовать погрузиться в мир ИТ. Не говоря уже о том, что человеческая цивилизация чем дальше, тем больше уходит в «цифровые миры» и возможность быть не только пользователем, но и создателем программ – это очень интересно.

Все, что вам потребуется для успешного прочтения этой книги и продуктивного усвоения материала, это: в первую очередь желание и намерение не сдаваться перед трудностями. Второе: компьютер с операционной системой Windows, в 2020 году вам подойдет компьютер из любой ценовой категории – Java без проблем работает на любых процессорах семейства Intel, ну и конечно лучше если у вашего компьютера будет хотя бы 4 гигабайта оперативной памяти. Конечно же вы можете использовать компьютер на базе ОС Linux или MacOS, только вам придется самостоятельно установить среду разработки JetBrains IntelliJ IDEA (это будет единственное отличие).

От вас не требуется каких-то особых познаний в информатике или программировании, все необходимое для того, чтобы начать программировать я и так вам расскажу в этой книге.

Да и вам не нужно знать высшую математику – это ответ на вопрос из топа страшных вопросов «что должен знать программист ДО того как станет писать код». То есть достаточно знать математику в пределах простейших математических операций и умения раскрыть скобки, ну и решать уравнения с одной переменной.

Большим плюсом будет умение читать и понимать английский язык, так исторически сложилось, что все в интернете найти ответ по программированию легче всего на английском языке. Плюс, также все новые и интересные книги и статьи публикуются на английском языке. Обратите на это внимание. При устройстве на работу знание английского также будет плюсом.

Почему Java?

Небольшой обзор текущего положения дел в языках программирования. Существует всемирный рейтинг языков программирования TIOBE (<https://www.tiobe.com/tiobe-index/>):

Dec 2019	Dec 2018	Change	Programming Language	Ratings	Change
1	1		Java	17.253%	+1.32%
2	2		C	16.086%	+1.80%
3	3		Python	10.308%	+1.93%
4	4		C++	6.196%	-1.37%
5	6	▲	C#	4.801%	+1.35%
6	5	▼	Visual Basic .NET	4.743%	-2.38%
7	7		JavaScript	2.090%	-0.97%
8	8		PHP	2.048%	-0.39%
9	9		SQL	1.843%	-0.34%
10	14	▲	Swift	1.490%	+0.27%
11	17	▲	Ruby	1.314%	+0.21%
12	11	▼	Delphi/Object Pascal	1.280%	-0.12%
13	10	▼	Objective-C	1.204%	-0.27%
14	12	▼	Assembly language	1.067%	-0.30%
15	15		Go	0.995%	-0.19%
16	16		R	0.995%	-0.12%
17	13	▼	MATLAB	0.986%	-0.30%
18	25	▲	D	0.930%	+0.42%
19	19		Visual Basic	0.929%	-0.05%
20	18	▼	Perl	0.899%	-0.11%

И здесь мы видим, практически неизменную пятерку лидеров: Java, C, Python, C++, C#. Но этот рейтинг имеет такое же отношение к реальности, как и прогноз погоды на неделю. Сейчас я расскажу про языки из топ-10, чтобы вы смогли попробовать оценить свои представления и желания в области программирования.

Стоит сразу сказать, что языки в рейтинге, с местами 10+ не очень подходят для начального обучения, ну кроме Ruby. Но они также востребованы, и лучше, как минимум на них посмотреть – может вам понравится. А может быть столкнетесь с ними, и жизнь заставит вас выучить какой-либо из них.

10-е место Swift. Детище Apple, замена ObjectiveC для платформы MacOSiOS (равно как Kotlin, замена Java на Android). Если вы хотите связать свою профессиональную деятельность с корпорацией Apple, то стоит начать его учить и не отвлекаться ни на что более. Он простой для изучения и даже есть курсы для детей. К тому же, на сегодняшний день специалистов со знанием его не так много.

9-е место SQL. Этот язык знать **обязательно**, потому что это язык «общения» с базами данных, сейчас ни одно приложение или сайт не может существовать без баз данных. Но учить его как первый язык смысла не имеет.

8-место PHP. «Домашний проект» датского программиста Расмуса Лердорфа, переросший в самый востребованный язык программирования сайтов в интернете. 80% сайтов используют PHP. Но это может быть и минусом – на PHP вы только сможете писать серверную часть (известны попытки писать приложения на нем, но это только в качестве экспериментов). PHP достаточно прост в изучении, и что может быть для некоторых решающим фактором, специалисты очень востребованы в многочисленных веб-студиях. Т.е. для цели: быстро изучить и пойти «зарабатывать, чтобы на жизнь хватало» – это язык номер 1.

7-е место JavaScript. Очень долгая история у этого языка программирования, можно сказать, что он появился вместе с интернетом (когда интернет стал доступным для массового использования). Но только в последние несколько лет он стал суперпопулярным. Для этого есть несколько причин: он стал удобным для написания больших проектов, кроме написания

простых скриптов «чтобы появлялось красивое окошко» теперь на нем можно писать практически все – клиентские приложения для iOS, MacOS, Android, Windows (фреймворк Electron), серверные приложения (фреймворк NodeJS). Он очень подходит для людей, которым нужна «движуха»: идеален для написания проектов на хакатонах, новые фреймворки (библиотеки программного кода, очень облегчающие жизнь программиста и делающие очень много черной работы) появляются каждый год – с ним не бывает скучно! Специалисты очень востребованы, можно сказать что «через 20 лет будет только JavaScript».

6-место VisualBasic.NET. Скажу честно: я не знаю почему он не только в топ-10, но и вообще почему он здесь. Единственное могу предположить, что до сих пор на нем пишут макросы для MS Office, ну и может быть в Америке есть много приложений которые до сих пор требуют поддержки и обновления.

5-е место C#. Основной язык программирования проприетарной (закрытой) платформы Microsoft. NET. Великолепный язык, постоянно развивается и будет развиваться, пока в него инвестирует деньги Microsoft. На нем можно писать все что угодно, только с одним ограничением: это все может работать там, где установлена Windows или работает виртуальная машина. NET и в отличии от Java, диапазон гораздо уже (пока что). С мобильными приложениями дела обстоят совсем печально – есть проект Xamarin, который использовать никто не советует. Но также есть игровой движок Unity, в котором используется C#, но это только для игр на большом количестве платформ. C# особенно любим на территории СНГ, т.к. любима Windows. Если вы хотите спокойно развиваться как программист, иметь поддержку крупной корпорации, никогда задаваться вопросом стабильной зарплаты – C# очень хорошо подходит для этого.

4-е место C++. Самый крутой язык программирования и также круто сложен. Писать можно все и подо все. Особенно востребован там, где требуется скорость исполнения кода: игры, мобильные игры, сервера. Если вы знаете C++, то выучить что-либо еще перестает быть проблемой. Еще раз повторю: самый сложный язык из нормальных (да-да есть еще и *ненормальные*, например, Brainfuck) и не сильно подходит как первый язык для изучения программирования, хотя в ВУЗах учебные программы могут начинаться именно с него.

3-е место Python. Самый простой и лучший язык для изучения программирования. Но, есть одно «НО»: на нем либо писать сайты (Flask\ Django), либо заниматься научными исследованиями в области искусственного интеллекта, big data, для которых нужны очень математические мозги в первую очередь. Стоит учесть, что простота его изучения довольно опасная вещь – студент в процессе обучения не видит, ЧТО реально стоит за многочисленными библиотеками (а они уже написаны на C и там все сложно).

2-е место C. Честно скажу не знаю почему, наверное, проектов на нем больше чем на C++. Он быстрее C++, но в нем приходиться писать более сложно.

1-е место Java. Но это не точно :). Просто языки в первой тройке любят меняться местами. Но раз книга про Java... Если вы прочитали и запомнили характеристики предыдущих языков, то можете сказать: а почему Java? Она ведь и не самая быстрая, и не самая простая в изучении, и в ней нет постоянных изменений и улучшений как в других языках. Дело в том, что Java это не только язык программирования, а это целая платформа (да C# – это тоже платформа, только меньше по охвату), которая позволяет вам писать программы для практически любых устройств: начиная от кофемолок, заканчивая огромными data центрами. Java использует большие корпорации, например Yandex и Google. Она надежна и безопасна, на ней пишут большие корпоративные системы. С одной стороны код написанный на Java выглядит многословным, но это же позволяет избежать критических и трудно заметных ошибок. Существуют тысячи библиотек с открытым кодом, которые можно использовать в своих проектах. На Java вы можете писать приложения для Android, и поверите это легче чем писать под iOS (на Objective C). И самое главное: это лучший язык для того, чтобы научиться писать правильные объектно-ориентированный код, с использованием хороших паттернов программирования.

ния. Хотите научиться писать красивый и понятный другим людям код (а это очень важное умение) – пишите его на Java. Огромным преимуществом Java является то, что этому языку много лет и за все эти годы в интернете накопилась огромная база знаний и ответов на многие проблемы, с которыми сталкиваются программисты на Java. И поэтому зачастую проще вбить запрос в Yandex и получить готовое решение, которое будет легче адаптировать к своей программе, чем «изобретать велосипед»

Что все-таки выбрать?

Здесь я позволю себе посмотреть на проблему выбора языка программирования с точки зрения требований рынка. И рынок диктует свои жесткие требования, и они растут из года в год. Первое десятилетие 21 века было «золотым веком» для того чтобы выбрать язык программирования по-вкусу: достаточно было выучить *только его*, и вы уже могли идти и трудоустраиваться. В 2020 году все намного сложнее – от вас будут требовать гораздо больших знаний даже на позиции джуниора. Вот что вам потребуется знать «в довесок» и чем вы будете заниматься, если выберете какой-либо из языков топ-10.

Swift. Вы будете писать приложения под iPhone\iPad. В основном это будут приложения с многочисленными окошками. Периодически от вас будут требовать написать что-либо на Objective C, потому что Swift еще слишком молод и нестабилен, чтобы были проекты только на нем.

PHP. Вы будете все время писать сайты, и это в лучшем случае. А скорее всего вы будете настраивать WordPress\Drupal\Joomla под требования заказчиков сайтов. Да есть маленькая вероятность, что вы найдете работу где надо будет писать сервер на PHP (т.е. не сайт, а, например, систему для мультиплерных игр), но обычно для этого используются другие технологии. А также вам придется (жизнь заставит) выучить HTML, CSS, JavaScript – сайты без них не могут существовать.

JavaScript. Тут есть несколько вариантов. Первый, самый распространённый: вы доучиваете еще HTML, CSS и идете писать сайты, веб-приложений, мобильные SPA (*single-page application*). Второй вариант: вы используете знание языка JavaScript для написание серверной части на NodeJS – и сейчас это очень востребовано, потому что это гораздо легче, чем писать сервер на C++/C#/Java. Третий вариант: вы доучиваете еще HTML, CSS и идете писать игры – Adobe Flash (основная технология для написания игр для браузеров) уже умер, – конкурентов нет. За последний год сформировался новый тренд (пятый путь): вы учите **основы** HTML CSS JavaScript, потом изучаете в деталях фреймворк ReactJS и вуаля! – вы реект-программист, – это такой мутант, порожденный трендом и спросом (но сейчас это **самый!** Простой способ войти в IT с достойной зарплатой – спрос на таких программистов просто зашкаливает).

C#. Вы не будете скорей всего писать на нем программы для Windows. С 99% вероятностью вы будете использовать технологию ASP.NET и писать сайты, да вам придется выучить HTML, CSS и JavaScript (на уровне основ). Второй вариант: писать игры на Unity. Третий вариант: вы все-же будете писать десктопные приложения, например, дописывать новые функции Skype.

C++\C. Вы сможете писать ВСЕ. Из очень востребованного сейчас: сервера, мобильные игры, системы управления дронами и автомобилями, системы безопасности и наблюдения.

Python. Тут все просто: идете в написание серверной части для сайтов, а так как никто не любит содержать много разработчиков без надобности – доучиваете HTML, CSS, JavaScript. Второй вариант, аналитик данных – дорога в банки, обычно там это самое востребованное. Третий вариант: вы не программист, а научный деятель.

Java. Разработка серверов и поддержка существующих систем в банках и корпорациях, которые вложили миллионы долларов в программные комплексы и хотят продолжать их развивать. Второй вариант – это разработка Android приложений. И здесь вам тоже есть что выбирать: стартапы, компании среднего размера, корпорации.

Про зарплаты

Мы же все живые люди и хотим применять наши знания и получать при этом не только интеллектуальное удовольствие, но и материальное вознаграждение. Если не сильно вдаваться в детали, в мире ИТ разработчиков принято относить к нескольким категориям компетентности.

Junior (джун, малыш) – разработчик, который выучил технологию в **теории**, но опыт коммерческой разработки у него равен нулю. Его основная задача – это не погоня за зарплатой, а за опытом, чтобы скорей перейти в следующий статус.

Middle (середняк) – разработчик с опытом, таких большинство. Выполнение поставленных задач – его зона ответственности.

Senior (сеньор) – настоящий профессионал, он знает технологию в нюансах, он сталкивался с огромным количеством «черной магии в коде» и у него есть необходимые «заклинания» чтобы эту магию рассеять. К нему прислушиваются, его советов спрашивают.

Architector (архитектор) – это вершина карьеры разработчика, дальше только управление проектами и человеческим ресурсом. Архитектор строит системы с учетом требований заказчика. Это как академик в науке.

Так вот, про зарплаты. Они конечно же разнятся в зависимости от:

– Заказчика – зарубежный заказчик платит больше и это «больше» может быть больше в разы.

– Востребованности – чем меньше специалистов, тем дороже (хотя может быть нет специалистов, потому что это уже никому не нужно)

– Технологии – заказчики очень падки на тренды, хотите получать больше – следите за трендами.

Если говорить о зарплатах, как о «средней температуре по больнице»:

Джун – это вилка зарплат от 400—500 долларов до 800—1000 долларов в месяц;

Миддл – это от 1500 до 3000 долларов в месяц;

Сеньор – это соответственно от 3000 до 5000—6000 долларов в месяц.

Еще раз, это очень неточные цифры. Бывают и Java миддлы, которые живут в глубинке и получают 800 долларов, бывают JavaScript миддлы, которые получают 7000 долларов. А бывают и разработчики-комбайны, которые «колбасят» и на 10к долларов в месяц. А есть PHP разработчики, которые за адаптивный шаблон на WordPress получили 2 000 000 долларов. Вы всегда можете отыскать актуальные цифры и для стран СНГ, и для Европы, и для Америки – это открытая и интересная информация. Также стоит знать, что независимо от выбранной технологии, на уровне «сеньор» зарплаты примерно равны +\– сотня баксов роли не играет.

Этим сфера ИТ и хороша – здесь ВСЕ зависит только от ваших знаний, умений, стараний и терпения. А дальше выбор за вами.

Часть I

Глава 0. Мы программируем Мы программируем железо

В наше время мы уже настолько окружены техникой, которая способна выполнять разнообразные программы, что уже воспринимаем это как должное. Что там компьютеры... телефоны, телевизоры, пылесосы (!), видеокамеры, автомобили, даже лифты – все они уже исполняют программы. Но для того чтобы писать хорошие программы, надо хотя бы примерно понимать, как устроен компьютер. В мире программистов компьютеры называются «железом» (хотя сейчас там гораздо больше пластика, стекла и кремния). У некоторых были уроки информатики, но, наверное, как это обычно происходит, они проходили за компьютерными играми :)

Что-же такое компьютер? Компьютер – это устройство, в которое можно ввести какие-либо данные, эти данные могут быть обработаны программой, и в конечном итоге будет выведен результат. В нашей реальности мобильный телефон (это тоже компьютер), в котором запущена программа (Instagram), в который вы вводите данные (ваши нажимы и касания пальцев, когда вы скролите ленту), и в итоге видим результат – новые посты в ленте.

Компьютер – это сложное устройство, он состоит из нескольких модулей, которые нужны для разных задач. В первую очередь самый заметный модуль (устройство вывода) – монитор\дисплей, через него мы получаем всю визуальную информацию. Устройства ввода, по меньшей мере два: клавиатура и мышь (или, например, тачпад). А еще устройство ввода – это цифровой сканер или VR-шлем (и он же устройство вывода). Далее непосредственно сам компьютер, обычно мы видим его как системный блок или нижнюю часть ноутбука.

Самые главные части современного компьютера: процессор (или несколько процессоров), оперативная память, твердый диск, видеокарта, звуковая карта. Процессор – это «мозг» компьютера, но в отличие от человеческого мозга, он не думает, а исполняет команды. Оперативная память – это куда загружаются программы, чтобы выполнять их в текущий момент. Оперативная память не сохраняет данные после выключения питания. Твердый диск – хранит разнообразные данные и программы. Данные на нем не будут потеряны после выключения питания. Чтобы понять разницу между оперативной памятью и твердым диском, вспомните себя, когда вы считаете сколько денег потратили: вы в голове перемножаете количество товаров на их цену (оперативная память), а потом записываете промежуточный результат на бумажку (твердый диск). Манипуляция мыслями занимает гораздо меньше времени, чем запись и потом поиск информации на бумажке – также и с компьютером: он быстро работает с оперативной памятью и гораздо медленнее с твердым диском. Видеокарта занимается ускоренным просчетом данных для последующей отрисовки, как мы все знаем это особенно критично для видеоигр :) Звуковая карта преобразует цифровые данные в электрические импульсы, которые «понимают» аудиоколонки.

Программист на Java в 99% случаев взаимодействует (и иногда сталкивается с нюансами) с дисплеем, клавиатурой, мышкой, процессором, оперативной памятью и диском, а также с другими компьютерами (обычно серверами) которые присыпают данные и ждут ответов через интернет.

Мы программируем и пишем программы

Существует огромная пропасть между представлением данных для человека и для компьютера. Для компьютера данные представляются в виде электрических импульсов – есть ток, нет тока. Человек не может читать электричество, и он придумал различные абстракции. И поэтому наличие\отсутствие электрического тока представляется двумя цифрами 0 и 1, это уж позволяет использовать бинарную математику, это уже позволяет путем ввода нулей и единиц управлять процессором и соответственно писать программы. Но такие программы не читабельны – мы привыкли общаться словами и предложениями. И поэтому возникли разнообразные языки программирования, использовать которые для написания программ намного лучше, чем писать единицы и нолики. Сначала возникли языки программирования только чтобы просто программировать компьютеры, позже люди стали создавать языки программирования для работы в разных областях науки и бизнеса. Например, Fortran – для математиков и их формул, Cobol – для экономистов и биржевых операций, Assembler – для низкоуровневого программирования устройств, BASIC – для обучения школьников, Go – для программирования распределённых систем, PHP – для упрощенного написания веб-сайтов. Некоторые языки были созданы как следующий эволюционный шаг: Delphi – эволюция Pascal, Java – эволюция C, C# – эволюция C++ и Java.

Но Java создавали не только для того, чтобы сделать улучшенную версию языка C. Когда компьютер превратился из «монстра научных институтов» в персональный компьютер, стали появляться, одна за другой, операционные системы; возникло большое количество производителей компьютерного железа, которые выпускали кучу всяких устройств, программировать которые обычному программисту, без погружения в глубины документации и драйверов (управляющих железом микропрограмм) было невозможно. И вот в светлый ум доктора информатики Джеймса Гослинга пришла идея создать такую технологию, которая позволила бы программисту писать такой код, который мог бы запускаться на **любом железе**. И его идея была им воплощена в язык программирования Java и виртуальную машину Java.

В чем суть? Виртуальная машина Java это программный компьютер внутри компьютера обычного на базе операционной системы Windows\MacOS\Unix\.... Виртуальная машина Java (Java Virtual Machine – JVM) специально создается для каждой операционной системы. И JVM знает, как «общаться» с операционной системой и через нее со всеми устройствами, которые могут быть подключены к компьютеру. В чем выгода для программиста? – программист пишет свой код только один раз (!) и ему в своем коде не надо знать на каком компьютере, с какой операционной системой его код будет запускаться. Это огромная экономия времени и денег. Именно поэтому Java код очень надежный и безопасный, он с одной стороны может ограничивать программиста, когда тот хочет «пошалить», а с другой стороны не дает программисту написать код, который может что-то сломать.

Стоит заметить, что концепция «виртуальных машин» получила дальнейшее развитие в других языках программирования, главным образом из-за безопасности. Например, виртуальная машина JavaScript, она встроена в браузер, дает определенные возможности для работы с документом в браузере, для обращения к серверу, но полностью запрещает произвольное считывание данных пользователя с диска. Очень похожая на JVM виртуальная машина Microsoft. NET.

Глава 1. Первая программа

Что такое программирование

Программирование – это процесс написания команд, которые потом будет выполнять компьютер. Очень важно понимать, что компьютер **не умеет думать**. Все, что компьютер **делает: он исполняет команды**. У программиста может складываться впечатление, что происходит какое-то «колдунство» и компьютер вытворяет самолично с программой, что захочет. Но на самом деле, это будет только означать, что программист не учел каких-то особенностей функционирования программы или библиотек, которые он использует, или нюансов как работает внешний источник данных, к которому он обращается. Чем профессиональней программист – тем меньше «неожиданных чудес» можно ожидать от написанного им кода.

Программирование на Java состоит из нескольких этапов:

- Написание программы на языке Java в редакторе
- Компилирование программы в байт-код (код понятный виртуальной машине Java) с помощью программы-компилятора
 - Исправление ошибок компиляции (compilation errors), если такие произошли в процессе компиляции
 - Запуск программы в виртуальной машине Java.
 - Исправление ошибок выполнения (runtime errors), если видим, что «что-то пошло не так»
 - Повторение пунктов 2—5 пока мы не получили работающую по нашему замыслу программу.

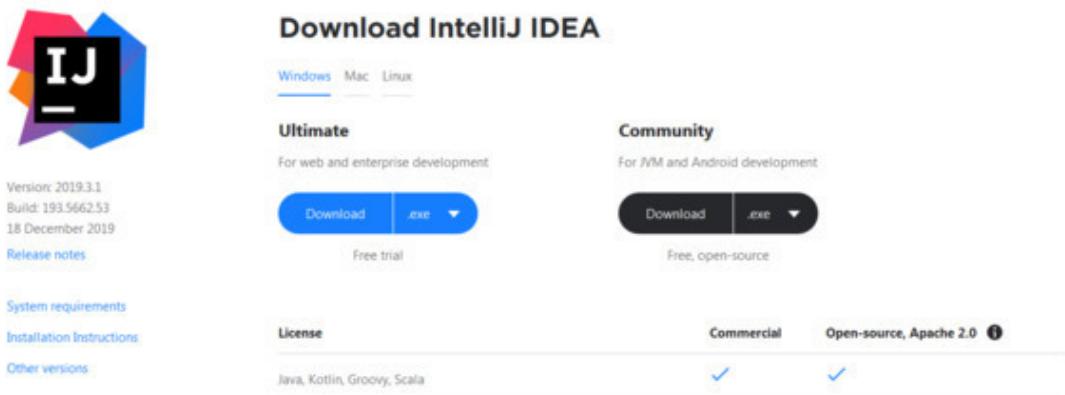
Можно писать код в одном из текстовых редакторов (Notepad, Notepad++, Atom, Sublime) и потом дальше через командную строку запускать компилятор, а потом запускать программу. Но все это громоздко и неудобно, именно поэтому программисты написали специальные программы, в которых можно делать полный цикл разработки программы гораздо проще и удобнее. Такие программы называются IDE (Integrated Development Environment) – интегрированная среда разработки, в ней происходит и написание программы, и компиляция, и выявление ошибок, и запуск программы. К тому же, большинство из них еще и подсказывают разработчику, что и в каком случае можно использовать и где он возможно уже совершил ошибку.

В мире Java-программирования есть несколько популярных IDE: IntelliJ IDEA, Eclipse, NetBeans. NetBeans самая редко используемая IDE на текущее время. Eclipse – это **бесплатная** IDE, с тысячами полезных плагинов, облегчающая жизнь разработчика. Поэтому, вполне возможно, что в крупной компании, в которую вы придетете работать, будут использовать именно Eclipse. И это стоит учитывать, потому что на самом деле вы захотите пользоваться только одной IDE: IntelliJ IDEA – лучшая и самая удобная IDE на текущий момент для написания программ на Java.

Настраиваемся на программирование. Устанавливаем IDE

Так как предполагается, что мы изучаем с нуля, то мы не будем заморачиваться с установкой виртуальной машиной Java и полного пакета для разработчика Java SDK. Достаточно будет скачать текущую версию IntelliJ IDEA, в которой есть все что нам будет нужно для старта.

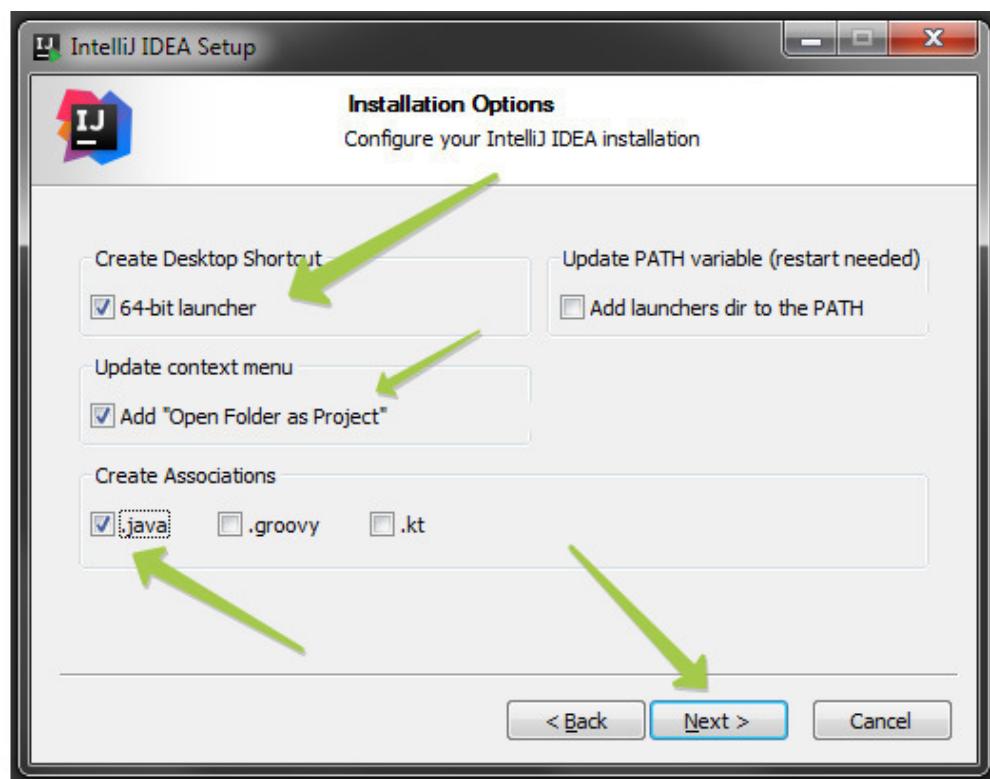
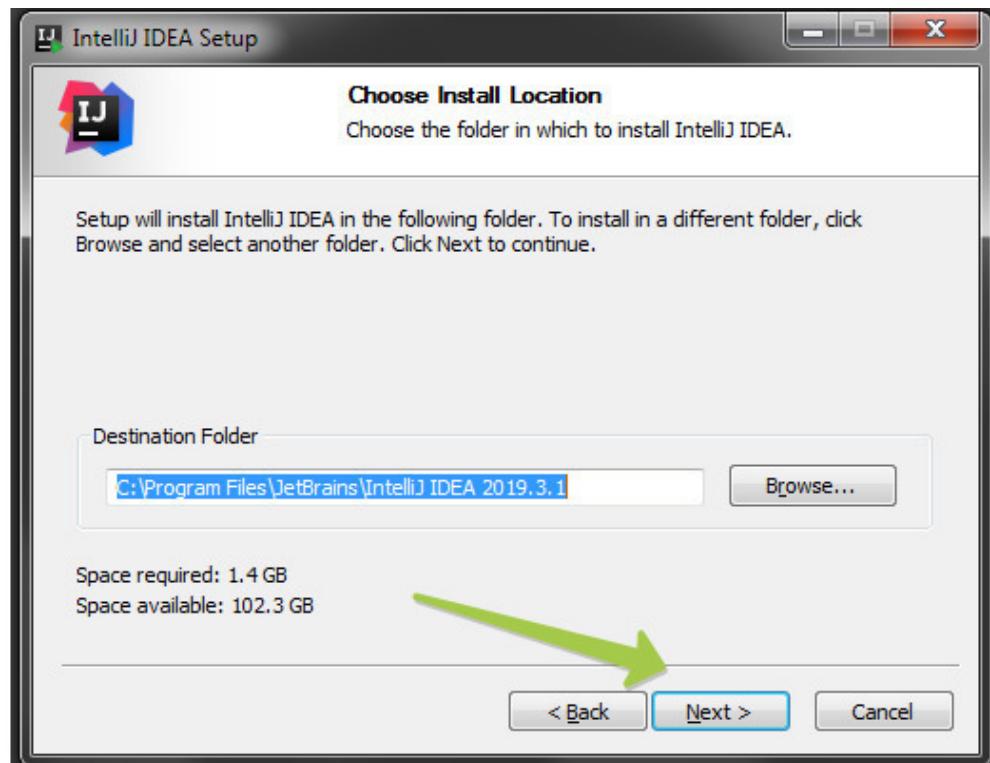
Зададим на сайт <https://www.jetbrains.com/idea/download/>

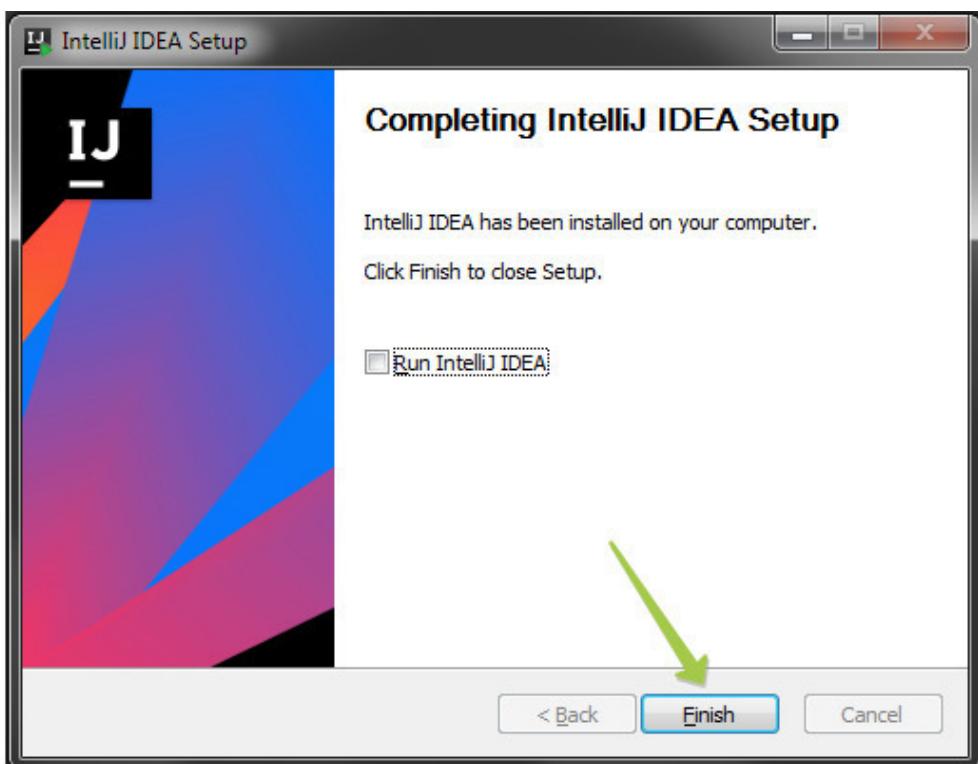
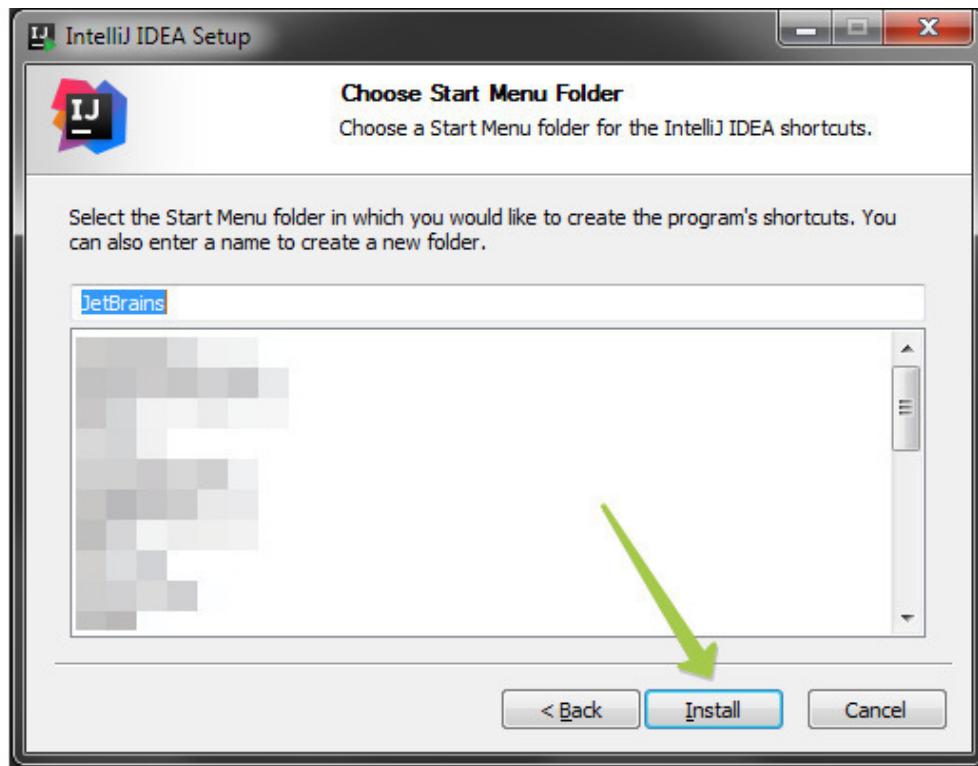


выбираем версию для скачивания: Ultimate (т.е. полную, но платную, хотя есть пробный период) или Community (тоже достаточную для наших целей)

Скачиваем и запускаем. Проходим через мастер установки, соглашаясь со всем что предлагают:



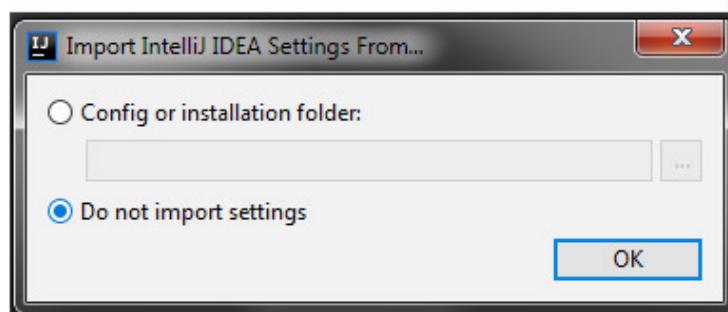




На Рабочем столе появится иконка приложения:



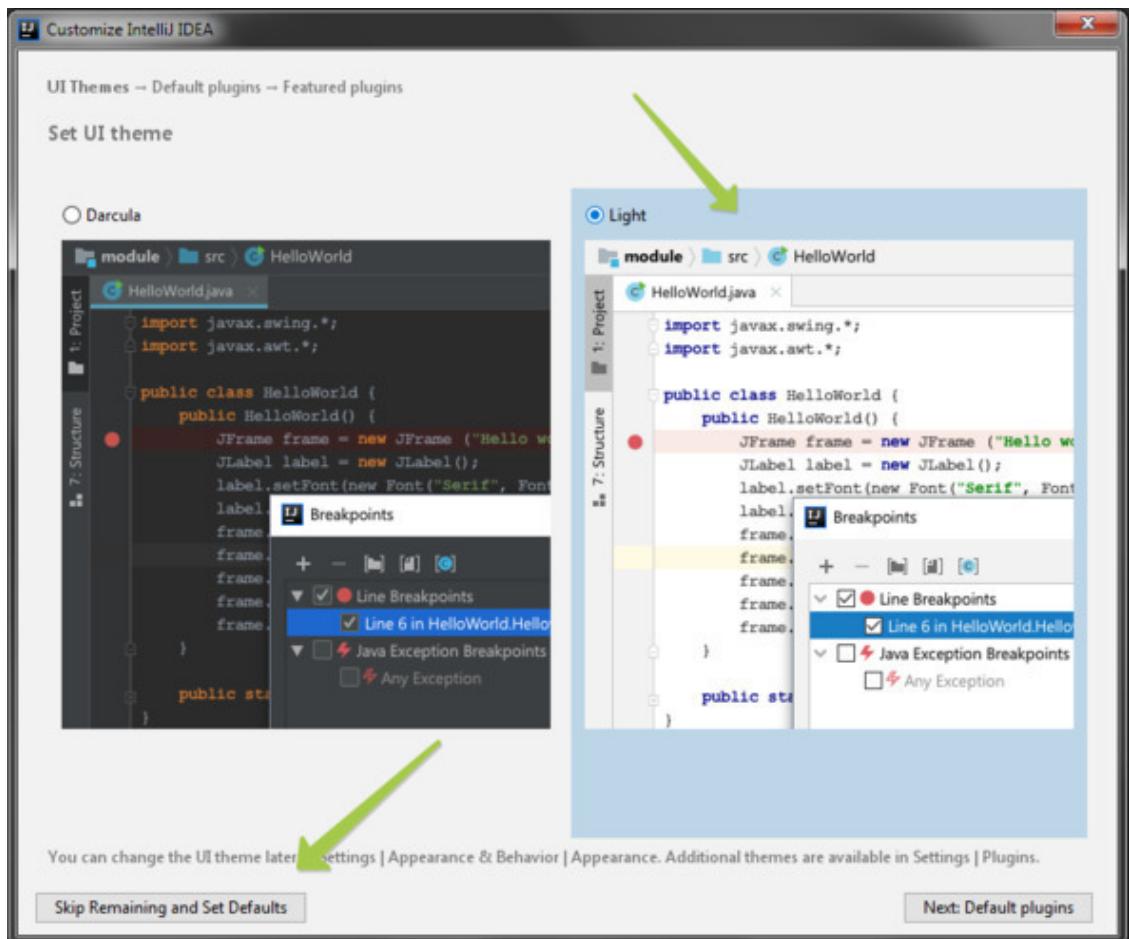
Дважды кликаем для запуска IntelliJ IDEA. И снова проходим еще через несколько экранов настройки (это будет только единожды):



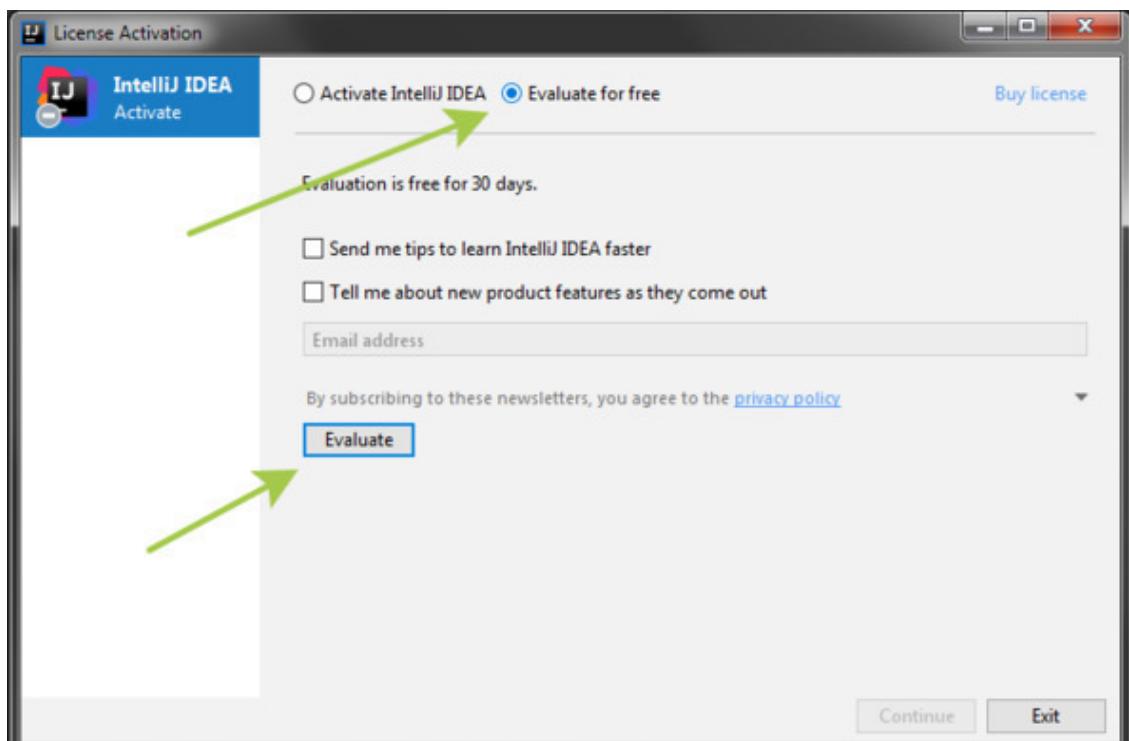
не импортируем никакие настройки



Соглашаемся...



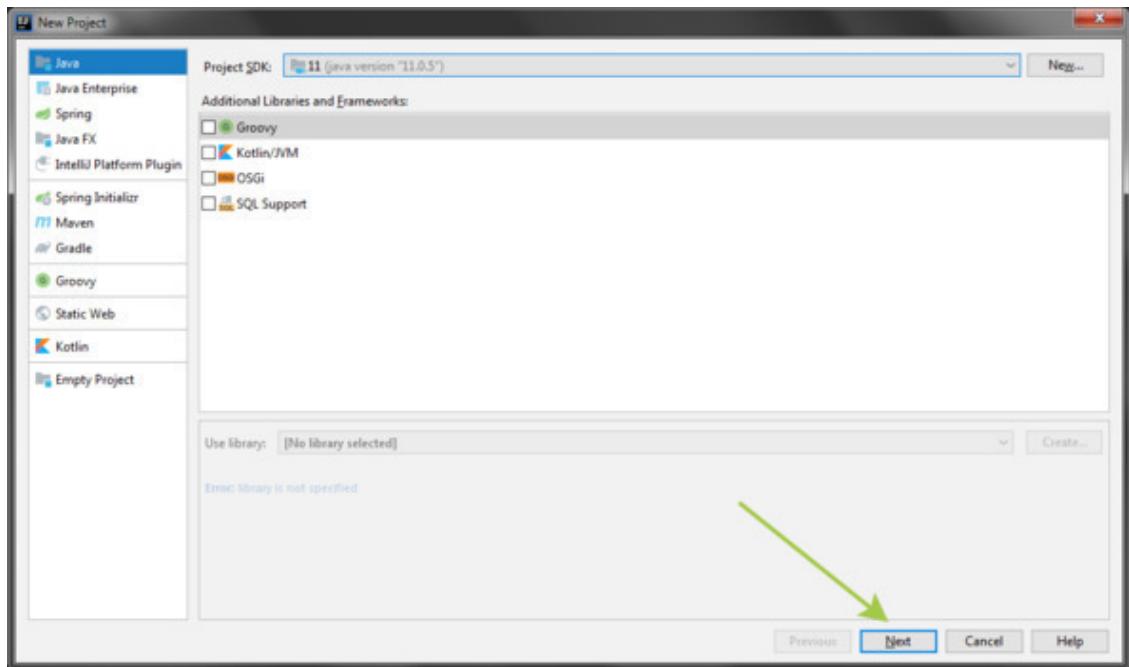
Выбираем светлую тему (это всегда можно изменить в настройках)



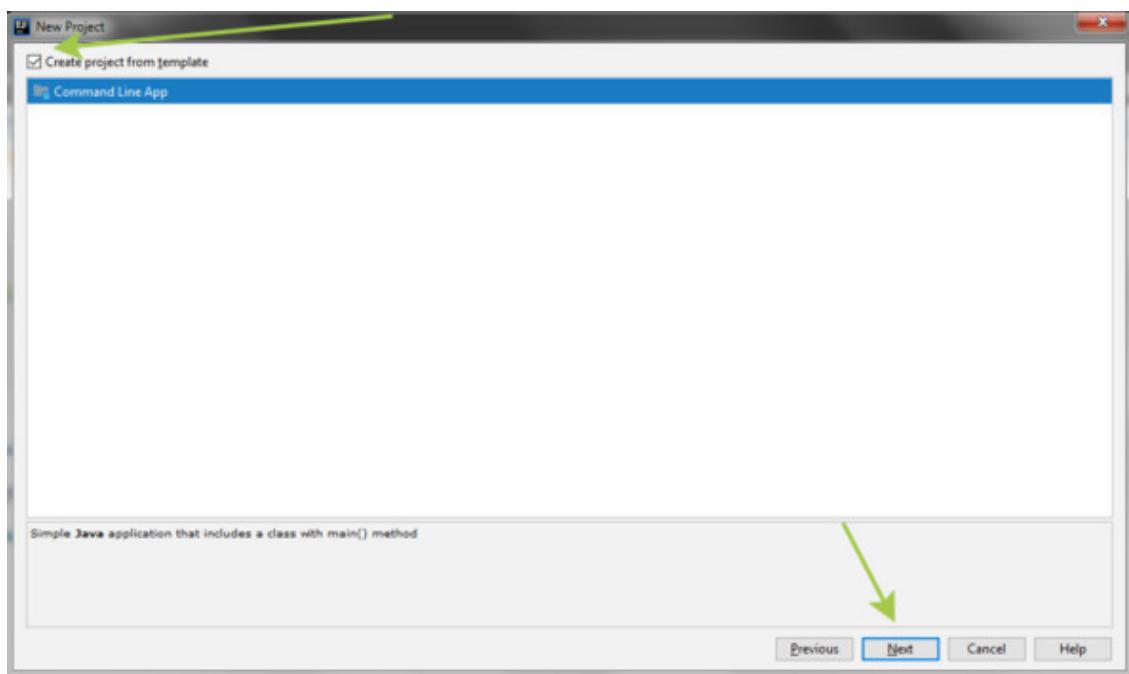
Я скачал версию Ultimate, поэтому выбираю пробный период



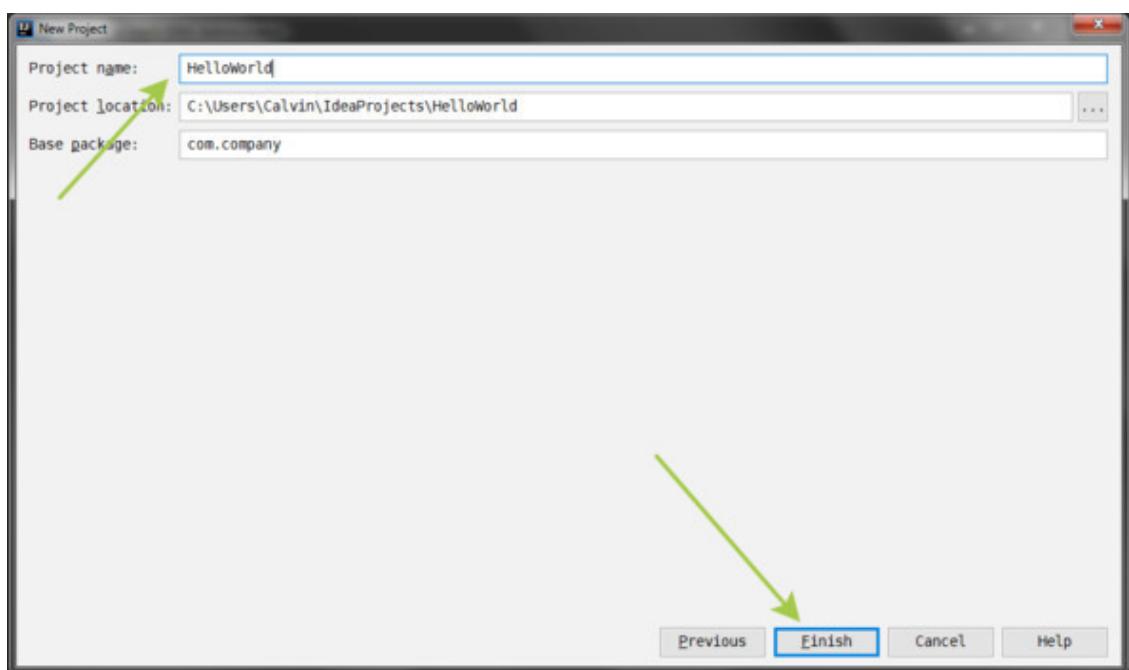
И вот финальный экран, где нужно кликнуть на Create New Project.



Слева должно быть выбрано Java, в центре вверху Project SDK: 11 (это версия Java, идущая вместе с IntelliJ IDEA), нажимаем Next.



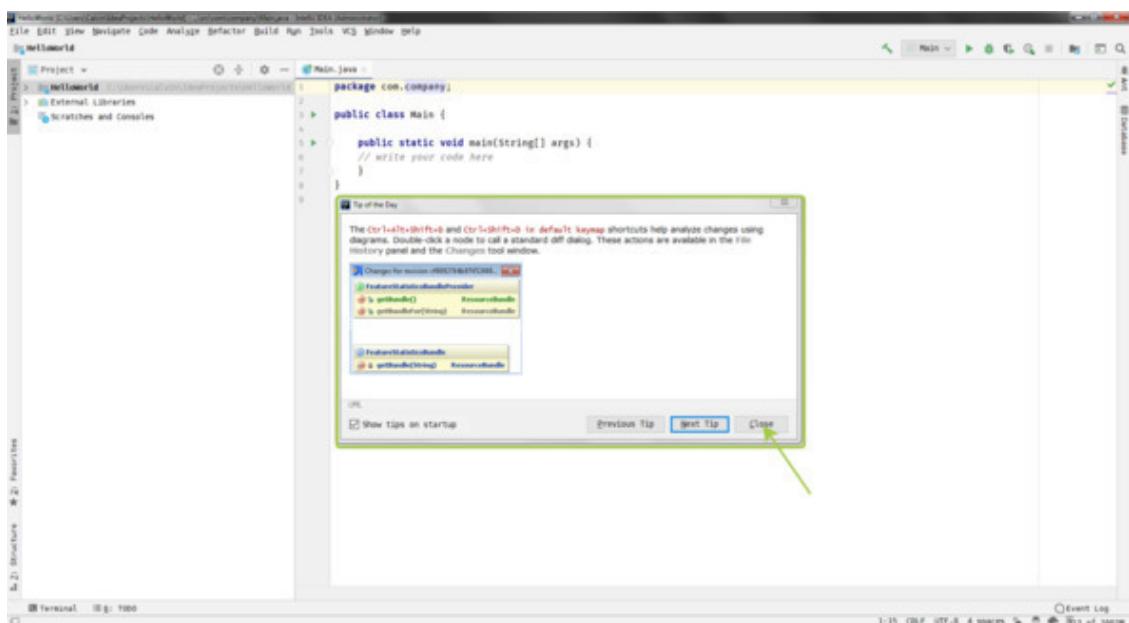
Ставим галочку Create project from template, выбираем Command Line App и кликаем Next.



В поле Project name вводим HelloWorld (это название нашего проекта) и нажимаем Finish.

Никогда не используйте русских букв (кириллицы) в названиях проектов, классов и т. д.

IDEA немного «подумав» откроет нам основное окно, в котором мы будем разрабатывать наши программы, и также откроется поверх маленькое окошко «Tip of the Day» – просто закройте его кликнув **Close**.

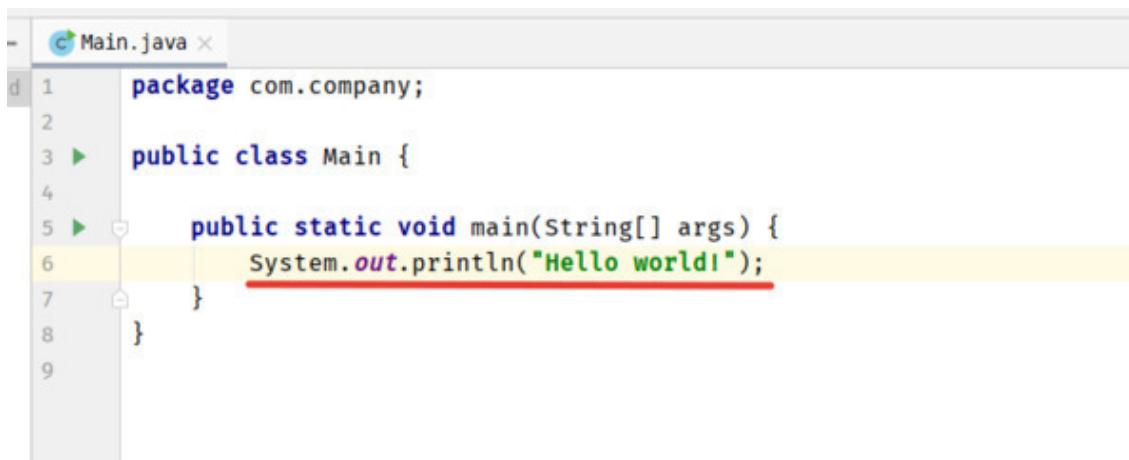


Теперь вам нужно написать свою первую строку кода, вместо надписи:

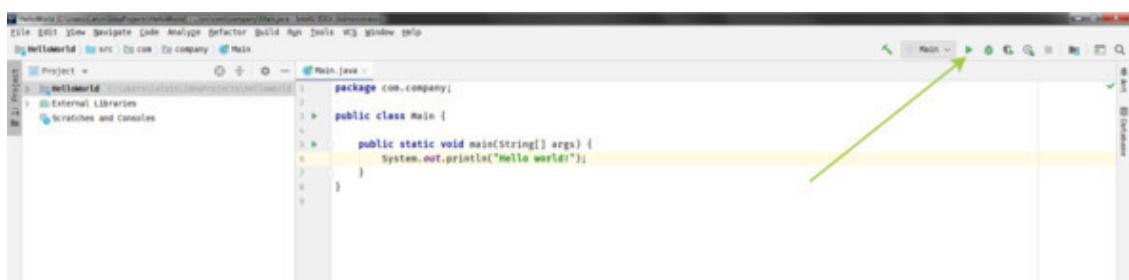
// write your code here.

Напишите там:

System.out.println ("Hello world!");



И запустите вашу первую программу нажатием на зеленый треугольник:



В итоге ваша программы выведет, в специальной панели IDEA:
Hello world!

```
package com.company;
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Main > main()

Run: Main > "C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.1\jbr\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.1\lib\javassist-agent.jar" -Dfile.encoding=UTF-8

Hello world!

Process finished with exit code 0

То, что написано ниже: «Process finished with exit code 0», означает, что программа завершилась без каких-либо явных ошибок. Поздравляю всех, кто дошел до этого последнего шага и успешно запустил первую программу!

Если же у вас что-то не получилось с установкой IntelliJ IDEA или с созданием программы – не переживайте и не волнуйтесь. Вы можете поступить как настоящий программист: вбейте в Google\Yandex «как установить IntelliJ IDEA», «создание Hello world в IntelliJ IDEA» и уже на первой странице результатов вашего поиска вы обязательно найдете статью, в которой пошагово вам покажут, как это сделать. Так же вы можете эти запросы вбить в YouTube – в 2020 году уже существует достаточно большое количество видеороликов с подобной информацией.

Не бойтесь пользоваться поисковиками, сейчас информации более чем достаточно, более того в книгах не найти ответы на все свои вопросы.

Глава 2. Данные

Программирование – это процесс написания команд, которые потом будет выполнять компьютер, чтобы манипулировать данными. Основная причина, почему компьютеры стали очень распространенными устройствами – они умеют очень быстро обрабатывать огромные объемы данных. Любая задача для программиста – это задача про управление и преобразования данных, чтобы получить результат – другие данные. Соответственно данные надо где-то хранить: исходные данные, чтобы их прочитать и начать использовать, результаты – для сохранения промежуточных результатов (например, при очень сложных расчетах) или для финальных значений.

Данные, используемые программой, хранятся в памяти компьютера. Такая память называется оперативной памятью (и чем ее больше, тем быстрее работает компьютер, тем больше вкладок можно открыть в браузере :)). Память состоит из маленьких ячеек памяти. Каждая ячейка хранит одно значение 0 или 1, и называется **бит**. Принято считать, что 8 бит – это **1 байт**. 1024 байта – это **1 килобайт**. 1024 килобайт – это **1 мегабайт**. 1024 мегабайт – это **1 гигабайт**. Ну а потом уже следуют тера-, пета-, экса-.... байты.

Программисту, в процессе работы, приходится использовать большие и маленькие числа, которые могут занимать разное количество памяти. Для того, чтобы программа работала быстрее и старалась не использовать больше памяти, чем необходимо, программист использует **типы данных**.

Типы данных

Использование типов данных – это добровольное ограничение использования памяти. Например, программист знает, что в расчетах будет использоваться значение не больше 77, поэтому он использует тип данных **short**, для хранения этого значений. Для больших значений – соответственно другие типы данных. Вообще в Java существует 8 примитивных типов данных:

boolean – хранит один бит информации;

byte – хранит один байт информации, и соответственно целочисленные значения от -128 до 127 включительно;

short – хранит два байта информации, и соответственно целочисленные значение от -32768 до 32767 включительно;

int – хранит четыре байта информации, и соответственно целочисленные значение от -2147483648 до 2147483647 включительно;

long – хранит восемь байт информации, и соответственно целочисленные значения от -9223372036854775808 (-2 в степени 63) до 9223372036854775807 (2 в степени 63 минус 1) включительно;

float – хранит четыре байта информации, и соответственно значения с плавающей точкой от 1.4×10^{-45} до 3.4×10^{38} ;

double – хранит восемь байт информации, и соответственно значения с плавающей точкой от 4.9×10^{-324} до 1.7×10^{308} ;

char – хранит 16 бит (2 байта), и соответственно символ в формате UTF.

То есть, когда нужно сохранить число 77 в памяти, и программист определил, что надо использовать для этого тип данных **short**, виртуальная машина Java выделит из общей памяти два байта.

В основном используют **int** и **float**, т.к. они покрывают большинство нужд.

Остается только один вопрос: как программист узнает *где расположены эти два байта и как к ним обратиться*. Именно для этого существуют переменные.

Переменные

Переменная – это **указатель** на область памяти определенного типа. Вы можете думать обо всем этом как о большом складе. Вы приходите на склад (вся свободная память компьютера) и говорите «мне нужно содержимое коробки с таким-то именем» (переменная) и заведующий склада (виртуальная машина Java) дает вам это содержимое. Также вы можете не только взять, но и положить туда то, что вам необходимо. Но если коробка маленькая, а вы пытаетесь засунуть туда большое содержимое – заведующий склада не даст вам это сделать и скажет, что вы ошибаетесь.

Несколько примеров как это выглядит в коде:

```
int box1;  
int box2 = 70;  
box1 = 50;  
int box3 = box1 + box2;  
System.out.println (box3);
```

В первой строке мы определяем переменную **box1** типа **int**.

Во второй строке мы определяем переменную **box2** типа **int**, при этом мы сразу кладем туда (или как говорят программисты «присваиваем») значение **100**.

В третьей строке мы присваиваем переменной **box1** значение **50** (программисты еще говорят «переменная **box1** проинициализирована значением 50»). Если мы этого не сделаем, то получим ошибку на этапе компиляции нашей программы: компилятор скажет, что нельзя в программе использовать переменные, у которых нет значений.

В четвертой строке мы складываем содержимое **box1** и **box2** и присваиваем новой переменной **box3** тоже типа **int**.

В пятой строке выводим на экран (или как еще говорят «распечатываем на экране») значение переменной **box3** (120).

Как долго живут переменные?

Чем больше и сложнее программа, тем больше различных данных придется хранить в различных переменных. Причем зачастую нам нужны переменные только на какое-то время совершения какой-то операции или нескольких операций. Стоит ли хранить даже такие «временные переменные»? – конечно же нет. И как раз для этого была придумана «область видимости переменной», которое определяет, как долго переменная «будет жить», т.е. будет доступна для использования. На практике область видимости определяется фигурными скобками {} – переменная объявленная и проинициализированная внутри фигурных скобок «умирает» как только поток выполнения программы выйдет за эти скобки. Отсюда следует несколько областей видимости:

- локальная переменная объявленная внутри метода (доступна только внутри метода)
- переменная объявленная внутри класса (доступна только внутри объекта, порожденного из класса)
- статическая переменная класса (доступна все время, т.к. класс «живет» все время пока исполняется программа)

Пример кода:

```
public class FooClass {  
    public static int a = 10;
```

```
public int b = 11;

public static void some() {
    int c = 12;
}

public void test() {
    int d = 13;
    {
        doSomething1();
        int e = 14;
        doSomething2();
        doSomething3();
    }
    int f = 15;
}
}
```

Представим, что программа создает в какой-то момент времени объект из класса FooClass. Тогда:

- переменная a будет «живеть» пока исполняется программа,
- переменная b будет жить пока жив объект класса FooClass,
- переменная c будет жить только пока программа выполняет метод some (),
- переменная d будет жить только пока программа выполняет метод test (),
- переменная e будет жить только пока программа выполняет команды которые находятся внутри фигурных скобок в методе test (),
- переменная f будет жить только пока программа выполняет метод test ()

Я понимаю что все это звучит сложно и непонятно, пока что, но потом, когда возникнет вопрос «почему переменная должна быть в этих скобочках?» просто перечитайте эту часть.

Объектно-ориентированное программирование (ООП)

Когда компьютеры только появились, и появились первые языки программирования, для написания программ было достаточно использования примитивных типов данных. Но мощность компьютеров росла год от года, и кроме сугубо научных программ стало все больше появляться программ для обслуживания других областей человеческой жизнедеятельности. Вместо одного программиста который писал программу за несколько месяцев, стали появляться команды программистов, которые уже писали проекты, разработка и последующая поддержка которых требовала уже годы. А это уже требовало писать такие программы и что самое главное, использовать такие языки программирования, которые бы позволили программистам тружиться независимо друг от друга над разными частями проекта. Началась эра объектно-ориентированного программирования (ООП).

Как говорит само название ООП – это программирование с использованием объектов. Объект – это некая структура данных, которая не только может содержать набор данных разных типов- атрибутов, но также и **методы** (способы) манипулирования ими.

Один из самых простых примеров для объяснения ООП – обычный автомобиль. Что такое автомобиль, используя **первый принцип ООП, абстракцию**, мы можем сказать, что это система взаимосвязанных объектов: колеса на раме, двигатель, система зажигания двигателя, система передач и руль, газ и тормоз. Человек садится в машину, вставляет ключ зажигания и инициирует систему зажигания, система зажигания запускает двигатель, человек пере-

ключает передачу с нейтральной на первую и двигатель начинает передавать обороты на колеса. Машина начинает двигаться и человек с помощью педалей газа, тормоза и руля управляет скоростью и направлением движения автомобиля.

Если бы мы писали программу-симулятор автомобиля, мы бы создали несколько **объектов, с методами**:

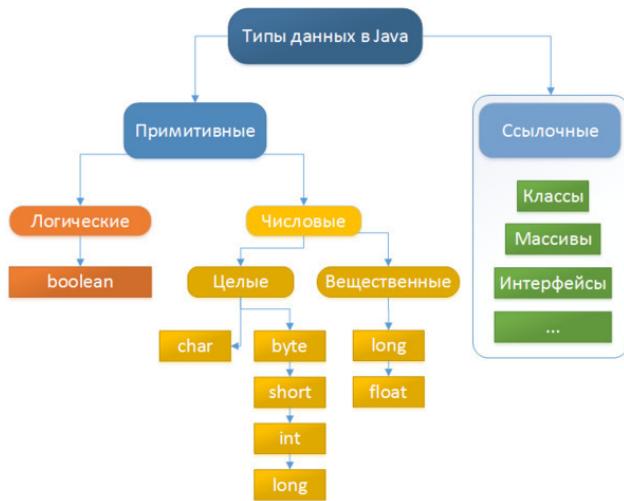
Объект	Методы
Колеса	поворот направо, поворот налево, вращать
Двигатель	запустить, остановить, увеличить обороты, уменьшить обороты
Педаль газа	нажать, отжать
Педаль тормоза	нажать, отжать
Руль	поворнуть направо, повернуть налево
Система передач	переключить передачу
Система зажигания	вставить ключ, вынуть ключ

И система взаимодействия объектов друг с другом, выглядела бы, например, так:



Стрелки с действиями, написанными поверх – это как раз методы, вызываемые у тех объектов, к которым они направлены. Как уже было сказано ранее, объекты также могут **хранить данные внутри себя**. Например, колеса могут хранить текущее значение оборотов в минуту; двигатель – скорость движения и расход топлива; система передач – текущую передачу и максимальное количество передач. Насколько будет детализирована программная модель, как объекты будут взаимодействовать между собой, какой конкретно объект что будет хранить – это ответственность программиста. И он делает путем написания **классов** – это шаблоны, по которым создаются объекты. В нашем случае будет объект класса Car (Машина), который на схеме не отображен. И этот объект класса Car содержал бы объекты, созданные из классов IgnitionSystem (Система зажигания), TransmissionSystem (Система передач), AccelerationPedal (Педаль газа), BreakPedal (Педаль тормоза), SteeringWheel (Руль), Engine (Двигатель), Wheels (Колеса). Создание объекта из класса еще программисты называют «инстанированием» (от англ. Instance – экземпляр). Для работы с объектом класса Car у вас была бы объявлена переменная типа Car, в которую JVM передала ссылку (адрес на участок в памяти) на объект, при его создании из класса Car.

Таким образом все типы данных в Java можно показать в виде такой схемы:



Обратите внимание, в категории **Ссылочные** типы стоит троеточие, здесь имеется в виду что количество ссылочных типов данных не ограничено, так как вы можете создавать неограниченное количество классов.

Глава 3. Операции с примитивными типами данных

Арифметические операции

Java поддерживает арифметические операции над числами, которые вы проходили в начальных классах школы: сложение, вычитания, умножение, деление.

Для начала создадим новый проект в IntelliJ IDEA и назовем его ArifmOperations. Далее добавим туда следующий код (особенностью IntelliJ IDEA является то, что вам не надо сохранять файлы она сама это сделает):

```
int a = 15;  
int b = 20;  
int c = a + b;  
System.out.println ("Сумма:" + c); // вывод: Сумма:35
```

Этот код очень похож на код из предыдущей главы. Здесь объявляются две переменные **a** и **b**, которые сразу инициализируются значениями **15** и **20** соответственно. Потом объявляется еще одна переменная **c** которой присваивают сумму значений переменных **a** и **b**. В четвертой строке используется метод **println** класса **System** для вывода результата. Этот метод мы будем практически всегда использовать для вывода на экран нужных нам данных. Так же вы можете заметить, что внутри метода **println** есть строка «Сумма:», которая складывается с переменной **c** (программисты говорят **конкатенируется**). Так делают, когда надо скомбинировать какой-то статический текст с динамическими данными.

На что еще важно обратить внимание в этом коде? Во-первых, в Java каждая строка должна завершаться **точкой с запятой**, если вы этого не сделаете, то IntelliJ IDEA вам подсветит это место как ошибку:



Второе важное замечание: в месте где распечатывается результат еще были добавлены две косые черты и написано

```
//вывод: Сумма:35
```

Две косые черты (или как их еще бы назвали *два слэша*) обозначают комментарий в коде. Комментарии – это очень полезная вещь в программировании. Комментарии используют для того, чтобы оставлять пояснения о том, что код делает или, как в нашем случае, что ожидается получить в результате. Второе применение комментариев: когда вам надо быстро «выключить» какие-то куски кода, чтобы, например, понять где у вас происходит ошибка в логике работы программы. Удалять код не вариант, потому что можно получить ошибку при повторном его наборе. Есть два вида комментариев – односстрочный, через два слэша, и многострочный через открывающие **/*** и закрывающий ***/**

```

1 > public class Main {
2
3 >     public static void main(String[] args) {
4 >         /*
5 >             Многострочный комментарий
6 >             позволяет написать много интересного
7 >             для программистов, которые будут работать
8 >             с вашим кодом
9 >
10 >         int a = 15;
11 >         int b = 20;
12 >         int c = a + b;
13 >         System.out.println("Сумма:" + c); // вывод: Сумма:35
14 >     }
15
16

```

Как вы можете догадаться, что остальные арифметические операции производятся подобным образом. Добавим еще несколько строк:

```

c = a - b;
System.out.println ("Разность:" + c);
// вывод: Разность:-5
c = a * b;
System.out.println ("Произведение:" + c);
// вывод: Произведение:300
c = a / b;
System.out.println ("Частное:" + c);
// вывод: Частное:0

```

В итоге, вся программа будет выглядеть так:

```

1 > public class Main {
2
3 >     public static void main(String[] args) {
4 >         /*
5 >             Многострочный комментарий
6 >             позволяет написать много интересного
7 >             для программистов, которые будут работать
8 >             с вашим кодом
9 >
10 >         int a = 15;
11 >         int b = 20;
12 >         int c = a + b;
13 >         System.out.println("Сумма:" + c); // вывод: Сумма:35
14 >         c = a - b;
15 >         System.out.println("Разность:" + c); // вывод: Разность:-5
16 >         c = a * b;
17 >         System.out.println("Произведение:" + c); // вывод: Произведение:300
18 >         c = a / b;
19 >         System.out.println("Частное:" + c); // вывод: Частное:0
20
21

```

Внимательный читатель может спросить: «а почему результат от деления – ноль?». Ответ очень простой: так как тип переменной **c** был объявлен как **int**, т.е. целочисленный, то дробная часть результата 0.75 была автоматически отброшена и остался только ноль.

Преобразование типов

Если мы все-таки хотим получить правильный результат деления, то, казалось бы, все что нужно сделать, это изменить тип переменной **c** на **double** (если кто помнит этот тип вмещает в себя вещественные числа или числа с плавающей точкой или говоря проще числа с дробной частью). Объявим новую переменную и присвоим туда результат деления:

```

double d = a / b;
System.out.println ("Новая попытка Частное:" + d);

```

После запуска программы мы получим:

Новая попытка Частное:0.0

И это следствие строгой типизации в Java. JVM считает что если в коде одно целочисленное значение на другое целочисленное значение, то результат тоже надо выдавать целочисленным. Если нас не устраивает такое поведение по-умолчанию, то можно явно задать тип результата:

```
double f = (double) a / b;  
System.out.println ("«Еще одна попытка Частное:» + f);
```

После запуска программы мы наконец-то получим корректный результат:

Еще одна попытка Частное:0.75

Таким образом, (double) это указание, полученный результат привести к типу **double**, а не **int**. И такой же синтаксис используется при приведении типов одних объектов к другим.

Инкремент и декремент

Инкремент – это увеличение значения переменной на единицу, в то время как, декремент – это уменьшение значения переменной на единицу. То есть когда нам надо увеличить или уменьшить значение переменной на единицу мы можем написать стандартный код:

```
int a = 3;  
a = a + 1; // результат: 4  
int b = 5;  
b = b - 1; // результат: 4
```

А можем написать и вот так:

```
int a = 3;  
a++; // Инкремент: переменная a теперь будет иметь значение 4  
int b = 5;  
b--; // Декремент: переменная b теперь будет иметь значение 4
```

Инкремент\декремент бывает двух видов: префиксный и постфиксный. Префиксный – это когда перед использованием значения переменной производиться операция увеличения\уменьшения, постфиксный это когда увеличение\уменьшение производиться уже после использования значения переменной. Небольшой пример:

```
int a = 10;  
int b = ++a;  
// сначала значение в переменной a становится 11,  
// затем оно присваивается b  
int c = a++;  
// переменной c присваивается 11 и только  
// потом значение a увеличивается на 1 и уже равно 12
```

Раньше (в ранних версиях JVM) такие операции были быстрее в исполнении, чем обычна запись по увеличению\уменьшению переменной на единицу. Сейчас основной смысл использования – это выглядит короче в коде.

Сокращенные арифметические операции

Можно сказать, что инкремент\декремент – это частные случаи сокращенных операций. Существует более общая **сокращенная** запись, используемая для таких целей:

```
int a = 3;  
a += 1; // Как и инкремент добавляет единицу, результат: 4
```

```
a += 5; // А теперь можно и так, результат: 9
a -= 1; // результат: 8
a *= 5; // Можно также и сокращенно записывать умножение,
результат: 40
int b = 10;
a /= b; // И делить сокращенно, и даже на значение другой переменной,
результат: 4
```

Особых рекомендаций, когда использовать полную, а когда сокращенную запись нет – со временем сами поймете, как вам удобнее.

Операции сравнения

В Java над числами можно не только производить арифметические операции, но также их сравнивать и получать результат сравнения типа boolean (true и false) – т.е. производить операции сравнения.

Создадим новый проект, назовем его ConditionOperations, и в метод main добавим следующий код:

```
int a = 1;
int b = 2;
boolean compare1 = a <b;
System.out.println ("Compare1: " + compare1);
```

В этом коде происходит создание двух переменных **a** и **b**, а потом их значение сравниваются и *результат сравнения* присваивается в переменную **compare1**, а потом он выводится на экран:

Compare1: true

Это означает, **2** (переменная **a**) больше **1** (переменная **b**) это правдивое утверждение и результат **true** (истина).

Добавим еще пару строк:

```
boolean compare2 = a> b;
System.out.println ("Compare2: " + compare2);
```

Теперь выведется такой результат:

Compare2: false

Другими словами, **1** (переменная **a**) больше **2** (переменная **b**) это ложное утверждение и результат **false** (ложь).

Всего существует шесть операций сравнения:

<(меньше),> (больше),
== (равно),!= (не равно),
<= (меньше или равно),>= (больше или равно)

Все, что они делают – это сравнивают значение левой стороны со значением правой стороны (или еще говорят: сравнивают левый operand с правым operandом). Обычно просто так, такие сравнения не применяются, основное применение – это сравнения внутри условных операторов, – то есть **когда нужно изменить поведение программы в зависимости от каких-то результатов**. Например, человек вводит значение суммы денег, которые он хочет снять с банковского счета, программа анализирует ввод, если значение равно нулю или меньше нуля – то программа просит повторить ввод.

Логические операции

Логические операции, это операции, которые можно производить исключительно над значениями типа boolean. Они нужны для возможности реализации так называемой булевой алгебры (алгебры логики) – это раздел математики, изучающий высказывания, рассматриваемые со стороны их логических значений (истинности или ложности) и логических операций над ними. В Java (да, как и в других языках программирования), в первую очередь логические операции используются для задания сложных проверок, состоящих из нескольких условий.

Создадим новый проект, назовем его LogicOperations, и в метод main добавим следующий код:

```
boolean a = true;
boolean b = false;
boolean c = true;
boolean d = false;
boolean result1 = a && b;
System.out.println ("result1:" + result1); // результат: false
boolean result2 = a && c;
System.out.println ("result2:" + result2); // результат: true
```

Здесь используется операция **&&** – логическое И (логическое умножение). Принцип ее работы – возвращать **true** (истину), когда И левый операнд И правый операнд равны **true**, во всех других случаях – возвращать **false**. Например, в жизни это условие: если надо купить яблоки И есть достаточно для этого денег, то можно идти в магазин. Если одно из этих условий не соблюдается, то в магазин идти уже не надо.

Следующая логическая операция – логическое ИЛИ (логическое сложение), обозначается как **||**. Ее результат будет равен **true**, если ИЛИ левый операнд ИЛИ правый операнд равен **true** (еще можно сказать если хотя бы один из operandов равен **true**). В примере с походом в магазин, яблоки может нам и не нужны, но если есть деньги, то уже можно идти в магазин. Добавим еще кода:

```
boolean result3 = a || b;
System.out.println ("result3:" + result3); // результат: true
boolean result4 = d || b;
System.out.println ("result4:" + result4); // результат: false
boolean result5 = b || d || a;
System.out.println ("result5:" + result5); // результат: true
```

Как мы увидим, что даже когда несколько значений равно **false**, но, если есть хотя бы одно **true** – результат тоже будет **true**.

Еще одна логическая операция – исключающее ИЛИ, обозначается как **^**. Ее результат будет равен **true**, только тогда, когда operandы различные. Пример из жизни: есть мужчины (М) и есть женщины (Ж), дети могут быть у МЖ и ЖМ, но у ММ и ЖЖ – детей быть не может. Добавим еще кода:

```
boolean result6 = a ^ b;
System.out.println ("result6:" + result6); // результат: true
boolean result7 = a ^ c;
System.out.println ("result7:" + result7); // результат: false
```

И еще одна логическая операция – логическое отрицание, или логическое НЕ, или инверсия, обозначается как **!**. Производится над одним operandом (еще говорят унарная опе-

рация). Ее результат будет равен **true**, когда операнд равен **false** и наоборот: если операнд равен **false** – результат будет **true**. Добавим еще две строки кода:

```
boolean result8 =!a;
System.out.println (<>result8:" + result8); // результат: false
boolean result9 =!b;
System.out.println (<>result9:" + result9); // результат: true
```

Вот весь код для проверки:

```
Main.java
1 package com.company;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         boolean a = true;
7         boolean b = false;
8         boolean c = true;
9         boolean d = false;
10        boolean result1 = a && b;
11        System.out.println("result1:" + result1);
12        boolean result2 = a && c;
13        System.out.println("result2:" + result2);
14
15        boolean result3 = a || b;
16        System.out.println("result3:" + result3); // результат: true
17        boolean result4 = d || b;
18        System.out.println("result4:" + result4); // результат: false
19        boolean result5 = b || d || a;
20        System.out.println("result5:" + result5); // результат: true
21
22        boolean result6 = a ^ b;
23        System.out.println("result6:" + result6); // результат: true
24        boolean result7 = a ^ c;
25        System.out.println("result7:" + result7); // результат: false
26
27        boolean result8 = !a;
28        System.out.println("result8:" + result8); // результат: false
29        boolean result9 = !b;
30        System.out.println("result9:" + result9); // результат: true
31    }
32 }
```

Задания

Это первая глава, с которой появляются задания и упражнения для закрепления прочитанного. Рекомендуется их выполнять, даже если они кажутся очень простыми и «вам все понятно». Из-за специфической строгости Java, иногда примитивные вещи могут вызвать удивление.

1. Создайте программу, которая выводит на экран:

```
*****
*   *
*   *
*   *
*****
```

2. Создайте программу, в которой объявлены 2 переменные типа **int** и с ними выполняются арифметические операции: сложение, вычитание, умножение и деление. Результаты выведите на экран.

3. Создайте программу, в которой объявлены 2 переменные типа **int** и **double**. Выполните с ними арифметические операции: сложение, вычитание, умножение и деление, – результат должен быть типа **double**. Результаты выведите на экран.

4. Создайте программу, в которой объявлены 2 переменные типа **int** и необходимо найти остаток от деления одного числа на другое, не используя встроенную функцию нахождения модуля в Java. Результаты выведите на экран.

5. Создайте программу, в которой объявлены 3 переменных типа **boolean**, со значениями `false`, `true`, `false`. Создайте с ними следующие выражения:

- а. Выражение обязательно должно содержать `&&` и результат должен быть `true`.
- б. Выражение обязательно должно содержать `||` и результат должен быть `false`.
- в. Выражение обязательно должно содержать `^` и результат должен быть `true`. (подсказка: почитайте про приоритеты логических операций в Java)

Глава 4. Управление выполнением программы. Условия

В Java, равно как и во всех остальных языках программирования, существуют специальные операторы (команды) для управления и изменения порядка выполнения частей кода: условия и циклы. Условия – это когда программа выбирает какую часть кода выполнить, в зависимости от некоторого логического значения. Циклы – это циклическое повторение части кода, в зависимости от условия или какого-то счетчика. Условия и циклы позволяют реализовывать сложные алгоритмы вплоть до искусственного интеллекта.

Условный оператор if

Условный оператор **if** позволяет вам выполнить какие-то команды если результат условного выражения равен **true** (условие соблюдается). Пример:

```
int a = 5;
int b = 10;
if (a < b) {
    System.out.println («Результат:»);
    System.out.println («а меньше b»);
}
```

Если мы запустим программу с таким кодом, ты получим вывод:

Результат:

а меньше b

Это произойдет потому что условие, определенное для условного оператора **if** равно **true**. Если поменять значение переменной **a**, например, на 15 то программа не выведет сообщений. Также обратите внимание, участок кода, который должен выполниться если *условие выполняется* оформлен внутри двух парных фигурных скобок – очень важно за этим следить, потому что допускается использование вложенных условий. Пример немного измененной и дополненной программы:

```
int a = 5;
int b = 10;
int c = 20;
if (b < c) {
    System.out.println («b меньше c»);
    if (b > a) {
        System.out.println («но b больше a»);
    }
}
```

Если мы запустим программу с таким кодом, ты получим вывод:

b меньше c
но b больше a

Как уже говорилось: код выполняется если логическое значение, передаваемое в оператор **if** равно **true**. В предыдущих примерах использовался только один оператор сравнения, но ничто не мешает использовать их несколько сразу, например, вот так:

```
if (a < b && b < c) {
    System.out.println («b находится между a и c»);
```

}

То есть, если **a < b** и в тоже время **b < c** – условие выполнится и программа выведет:
b находится между a и c

Условный оператор else

А что делать если условие не соблюдается? Можно конечно пытаться создать другое условие для такого случая:

```
int a = 15;
int b = 10;
if (a <b) {
    System.out.println («Результат: a меньше b»);
}
if (a> b) {
    System.out.println («Результат: a больше b»);
}
if (a == b) {
    System.out.println («Результат: a равно b»);
}
```

Но это выглядит избыточно и как минимум затрудняет понимание логики программы. К тому же иногда нужно описать случай *когда-ни-одно-условие-не-верно*. Вот для такого и существует оператор **else**, он может работать только вместе с **if**. Вот переписанный предыдущий пример:

```
int a = 15;
int b = 10;
if (a <b) {
    System.out.println («Результат: a меньше b»);
}
else {
    System.out.println («Результат: a больше b»);
}
```

Внимательный читатель может заметить, что случай, когда **a == b** пропущен. Добавим его вот таким образом:

```
int a = 15;
int b = 10;
if (a <b) {
    System.out.println («Результат: a меньше b»);
}
else if (a> b) {
    System.out.println («Результат: a больше b»);
}
else {
    System.out.println («Результат: a равно b»);
}
```

Тут мы видим еще один вариант использования: **if-else-if**. Таким образом все условия, которые должны быть проверены, помещаются в цепочку, если не одно условие не сработало, то выполняется код в последнем **else**.

Стоит заметить, если код можно разместить в одной строке, то парные фигурные скобки можно не писать. С одной стороны, это уменьшает количество кода, с другой – для новичков может выглядеть запутанно в больших программах, поэтому выбирайте сами как вам удобнее. Без фигурных скобок наш пример выглядит вот так:

```
int a = 15;  
int b = 10;  
if (a < b)  
    System.out.println («Результат: а меньше b»);  
else if (a > b)  
    System.out.println («Результат: а больше b»);  
else  
    System.out.println («Результат: а равно b»);
```

Тернарный оператор

Затрагивая тему сокращения кода обязательно надо рассказать о тернарном (тройном) условном операторе. Вот пример его использования:

```
int a = 15;  
int b = 10;  
int maxNumber = a < b? b: a;  
System.out.println («Большее число это: » + maxNumber);
```

Это работает так: если условие соблюдается, то берется **левое** значение от двоеточия, иначе берется **правое** значение от двоеточия. В нашем случае условие **a < b** верно, поэтому возьмется значение переменной **b** и присвоится переменной **maxNumber**.

Более сложный пример нахождения максимального числа из трех (a,b,c):

```
maxNumber = a < b? (b < c? c: b): (a < c? c: a);
```

Сначала сравниваем переменные **a** и **b**, а потом ту что больше с третьей переменной **c**.

Оператор выбора switch-case

В Java, есть еще один условный оператор – это **switch**. Но в отличие от **if**, он проверяет только на равенство, с другой стороны, его использование выглядит более читаемо в коде и именно поэтому иногда лучше использовать **switch**.

Создадим проект в IntelliJ IDEA с названием NumberTalking. И добавим следующий код:

The switch block is highlighted with a red rectangle.

Программа с помощью **switch** сравнивает переменную **number** с уже готовыми значениями, описанными с помощью **case**, и в случае совпадения исполняет определенный для этого код. **default** используется, если ни одно значение в **case** не подошло (как простой **else**). **break** нужен для того, чтобы после выполнения кода не было сравнивания с другими **case**. Попробуйте разные значения переменной **number**. Как вы увидите: она «может считать только до трех». Важно отметить что параметром для **switch** могут выступать переменные только нескольких типов: **int**, **byte**, **short**, **char**, **String**.

Задания

Использование условных операторов **if-else** уже дает свободу для написания более сложных программ, поэтому и задания теперь будут более сложные и объемные.

1. Напишите программу-анализатор времени года по номеру месяца. В ней должна задаваться переменная **month** типа **int**. Значение этой переменной программа анализирует и выдает время года, например, если **month** равно 1 – выведет «А сейчас-то зима!», если равно 6 – «Лето красное!» и т.д., если же **month** больше 12 или меньше 1 выводит например «Вы с другой планеты?».

2. Напишите программу, которая вычисляет площадь фигуры. Если переменные **a** и **b** больше ноля, программа считает площадь прямоугольника. Если одна из переменных **a** или **b** равна нолю, то считается площадь круга, причем значение радиуса берется из второй переменной, значение которой больше ноля.

3. Напишите программу которая, определяет високосный год в переменной **year** или нет. Високосные годы делятся нацело на 4, но столетия, которые не делятся нацело на 400 високосными годами не являются. Для облегчения расчетов вы можете использовать встроенный оператор Java для нахождения остатка от деления **%** (например, $7 \% 3 = 1$).

4. Напишите программу калькулятор. В ней задается три переменных: **a** (тип **int**, хранит первое значение), **b** (тип **int**, хранит второе значение) и **op** (тип **int** хранит код операции: 1 – сложение, 2 – вычитание, 3 – умножение, 4 – деление). Программа должна совершить операцию с двумя переменными **a** и **b**, в зависимости от кода операции. Например, $a=10$; $b=20$; $op=3$ – в результате программа выведет 200. Поиграйтесь с различными значениями. Обязательно добавьте проверки на неправильные значения (нулевые, отрицательные), проверку на деление на ноль.

5. Попробуйте программу **NumberTalking** «научить считать» до пяти.

6. Напишите программу, которая выводит название месяца года по его номеру, определенному в переменной **month**, используя **switch-case**.

7. Напишите программу, которая выводит количество дней в месяце по его номеру, определенному в переменной **month**, используя **switch-case**.

8. Напишите программу, которая принимает на вход значение года и номер месяца и выводит количество дней в месяце, с учетом года (обычный или високосный), используя **switch-case**.

Глава 5. Управление выполнением программы. Циклы

Кроме возможности выполнять разные части кода, в зависимости от условий, еще одна важнейшая возможность – это исполнять одни и те же куски кода какое-то количество раз (а иногда и бесконечно). Для этого в Java (и во всех других языках программирования) существуют циклы. Есть циклы с условием, есть циклы со счетчиком. Сейчас мы рассмотрим цикл с условием: **while**.

Цикл while

Как следует из его английского значения (while в переводе означает «пока что»), он будет повторять в цикле операции, пока результат определенного для него условия равен **true**. Создадим новый проект в IntelliJ IDEA с названием NumberCounter. И в классе Main введем следующий код:

```
>Main.java
1 package com.company;
2
3 public class Main {
4     public static void main(String[] args) {
5         int i = 1;
6         while(i <= 5)
7         {
8             System.out.println("Считаю "+i);
9             i++;
10        }
11    }
12 }
13
```

И получим:

```
Run: Main
1 "C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.1\jbr\bin\java.exe" "
2 Считаю 1
3 Считаю 2
4 Считаю 3
5 Считаю 4
6 Считаю 5
7
8 Process finished with exit code 0
```

Разберемся что происходит. Объявляется переменная **i** типа **int** и ей присваивается значение 1. Это так называемый счетчик. Далее идет оператор **while**, которому передают логическое выражение (проверку), и пока значение переменной **i** меньше или равно 5, будут выполняться команды внутри тела цикла (внутри парных фигурных скобок). Одно выполнение тела цикла называется *итерацией*. Обратите внимание, что в теле цикла последняя команда – это инкрементирование значения переменной **i**. Если так не сделать, то возникнет *бесконечный цикл* и программа будет бесконечно выводить «Считаю 1». Теперь добавим код, чтобы программа отсчитала назад до единицы:

```
1 package com.company;
2
3 public class Main {
4     public static void main(String[] args) {
5         int i = 1;
6         while(i <= 5)
7         {
8             System.out.println("Считаю "+i);
9             i++;
10        }
11    }
12    while(i-- > 1)
13    {
14        System.out.println("Считаю назад "+i);
15    }
16 }
17 }
```

И теперь полный вывод программы будет:

```
"C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.1\jbr\bin\java.exe"
Считаю 1
Считаю 2
Считаю 3
Считаю 4
Считаю 5
Считаю назад 5
Считаю назад 4
Считаю назад 3
Считаю назад 2
Считаю назад 1
Считаю назад 0

Process finished with exit code 0
```

Из интересного: обратите внимание на декремент, сначала происходит проверка текущего значения и только потом происходит декремент. Если бы было написано:

while (-i > 0)

то сначала бы происходила декрементация и только потом сравнение. Тоже самое действительно и для операции инкремента. Как вы можете помнить это особенность использования **операций инкремента и декремента – постфиксный и префиксный инкремент\декремент**.

Но обычно для циклов, которые должны отработать определенное количество раз используется другое оператор цикла **for**, мы его рассмотрим ниже. Основное предназначение цикла **while** – это периодически выполнять код до тех пор, пока в теле цикла что-то не изменит условие цикла или вообще цикл не будет прерван с помощью оператора **break**. Например, это может быть часть программы, которая взаимодействует с пользователем: она будет бесконечно задавать вопросы, пока пользователь не введет команду выхода.

Цикл do-while

Цикл **do-while** это такой же цикл, как и **while**, только с единственной разницей, что тело цикла выполнится хотя бы один раз прежде чем произойдет проверка в **while**:

```
int t = -1;
do {
    System.out.println («Да-да я работаю!»);
} while (t> 0);
```

выведет:

Да-да я работаю!

хотя логическое выражение вернет false. И именно потому, что одна итерация цикла выполнится в любом случае, цикл do-while очень-очень редко используется.

Цикл со счетчиком `for`

Цикл **for** – улучшенный вариант цикла **while** со счетчиком. Он был создан специально для этого. Вот как выглядит код нашей программы с ним, которая считает от 1 до 5 и обратно:

```
for (int i = 1; i <= 5; i++)  
{  
    System.out.println («Считаю »+i);  
}  
for (int i = 5; i> 0; i – )  
{  
    System.out.println («Считаю назад »+i);  
}
```

Как вы видите цикл `for` состоит из трех частей (обычно): 1) объявление переменной-счетчика; 2) объявление условия, которое должно соблюдаться, чтобы исполнялось тело цикла; 3) операция изменения счетчика, в нашем случае обычный инкремент (а можно было написать `i+=1` или даже `i=i+1`). Все эти три части можно разнообразить или вообще опустить. Вот несколько примеров:

бесконечный цикл

```
for (;;) { }
```

или вот так (переменная была объявлена ранее):

```
int i = 0;  
for (; i < 10; i++)
```

или вот так (счетчик инкрементируется в теле цикла):

```
for (int i = 0; i < 10;) {i++;}
```

а можно даже несколько счетчиков:

```
int a,b;  
for (a = 1, b = 5; a <b; a++, b - ) { }
```

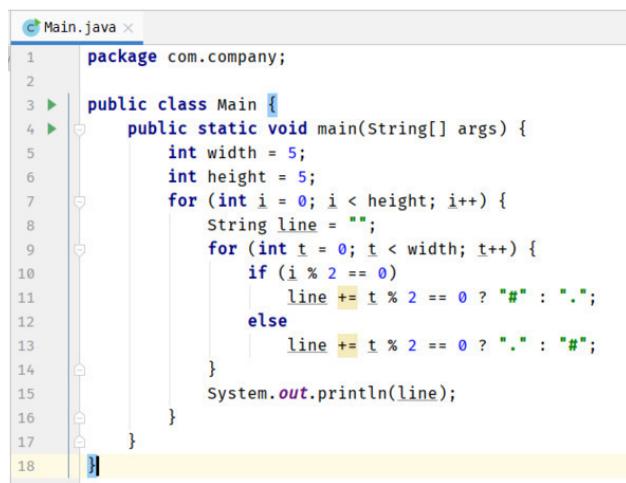
Цикл for – это чемпион по частоте использования, когда программисту надо реализовать проход по какому-то массиву данных, коллекции, проверить большое количество данных, сделать выборку данных по каким-то условиям.

Как и условия, циклы тоже могут быть вложены друг в друга. Обычно это необходимо, когда производится проверка одного набора данных на совпадение с другим набором данных (пример: нахождение пересечение множеств) или, когда реализуется какой-то алгоритм простой сортировки.

В качестве примера мы сейчас сделаем программу, которая распечатывает шахматную доску, вот в таком виде:

#.#.#
.#.#. #.
#.#. #
.#.#. #
#

Это доска размером 5 на 5, мы сделаем программу, которая сможет выводить на экран шахматную доску с разной размерностью. Создадим новый проект в IntelliJ IDEA и дадим ему название ChessmateBoard. Напишем следующий код:

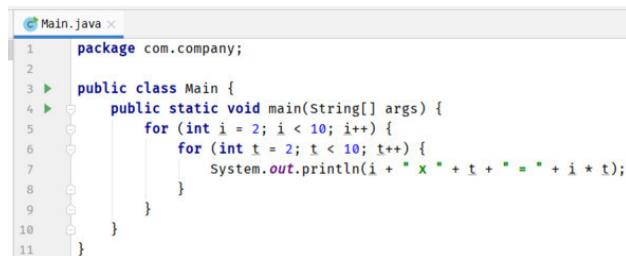


```

1 package com.company;
2
3 public class Main {
4     public static void main(String[] args) {
5         int width = 5;
6         int height = 5;
7         for (int i = 0; i < height; i++) {
8             String line = "";
9             for (int t = 0; t < width; t++) {
10                if (i % 2 == 0)
11                    line += t % 2 == 0 ? "#" : ".";
12                else
13                    line += t % 2 == 0 ? "." : "#";
14            }
15            System.out.println(line);
16        }
17    }
18 }
```

В строках 5 и 6 объявляются и инициализируются две переменные `width` и `height` – это размеры нашей доски. В строке 7 задается внешний цикл, который будет считать количество рядов доски. В строке 8 объявляется и инициализируется пустой строкой переменная `line`. В нее программа будет «набирать» клетки текущего ряда доски. В строке 9 задается внутренний цикл, в теле которого будут добавляться «клеточки» в текущий ряд. Далее реализуется логика отрисовки шахматного порядка – когда один ряд начинается с темной клетки, а следующий – со светлой. Т.е. в строке 10 мы проверяем если номер текущего ряда **четный** (делится на число 2 без остатка), то начинаем с символа `#`, и соответственно если номер ряда **нечетный**, начинаем с символа точки. Так как в каждом ряду клетки доски чередуются, то именно поэтому в строках 11 и 13 тоже идет проверка на четность, чтобы добавлять поочередно то символ решетки, то символ точки. Самое прекрасное, это то, что можно задавать сколь угодно разные значения ширины и высоты доски и все равно она будет правильно отрисована.

Еще один пример работы вложенных циклов будет программа, которая распечатывает всем нами любимую таблицу умножения. Создадим новый проект в IntelliJ IDEA и дадим ему название MultiplicationTable. Вот код программы:



```

1 package com.company;
2
3 public class Main {
4     public static void main(String[] args) {
5         for (int i = 2; i < 10; i++) {
6             for (int t = 2; t < 10; t++) {
7                 System.out.println(i + " x " + t + " = " + i * t);
8             }
9         }
10    }
11 }
```

В строках 5 и 6 определяются внешний и внутренний циклы. Внешний проходит по значениям, **которые умножают**, от 2 до 9 включительно. Внутренний цикл проходит по значениям, **на которые умножают**, и тоже от 2 до 9 включительно. В строке 7 выводится строка, которая формируется *на лету*: берется значение переменной `i` его конкатенируют (или говоря по-простому, но не совсем корректно «складывают») со строкой " `x` ", потом полученную таким образом строку еще конкатенируют со значением переменной `t`, потом результат конкатенируют со строкой " `=` " и конкатенируют со значением произведения переменных `i` и `t`. В общем,

объяснение звучит сложнее, чем-то что написано в коде. В результате программа выводит длинную таблицу в один столбик:

2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
.....

Стоит быть очень внимательным, когда работаете с циклами, особенно если тело цикла большое и имеет большое количество вычислений – это может существенно замедлять работу программы. Поэтому, когда речь идет о обработке большим массивов данных, например, сортировке, важно уметь выбрать подходящий алгоритм.

Задания

Так как, мы уже познакомились и с циклами и условиями, и поэтому некоторые задания будут на умение их использовать вместе.

1. Напишите программу которая выводит числа от 10 до 20.
2. Напишите программу которая выводит числа от 10 до -10.
3. Измените код программы, выводящей таблицу умножения, так, чтобы выводилась таблица умножения от 2 x 1 до 11x 10.
4. Напишите программу которая выводит прямоугольник, стороны которого состоят из «решеток», причем можно задавать разную ширину и высоту прямоугольника. Например, прямоугольник 5 на 3 будет выглядеть так:

```
#####
#....#
#####
```

5. Напишите программу, которая считает факториал значения, заданного в переменной F. Факториал числа, это произведение натуральных чисел от 1 до самого числа, включая данное число (в математике обозначается восклицательным знаком). Например, $1! = 1$, $2! = 2$, $3! = 1*2*3 = 6$, $4! = 1*2*3*4 = 24$.

Глава 6. Управление выполнением программы. Операторы перехода

Как мы видели из предыдущих глав, нам понадобилось всего 5 ключевых слов (`if`, `else`, `while`, `do`, `for`) чтобы программы которые могу делать выбор и повторять действия. Магия состоит в том, что, используя всего эти пять слов программисты пишут большие проекты. Выполнение условия может порождать целую цепочку действий, тело цикла может содержать в себе десятки проверок, внутренних циклов. И чем дольше «живет» проект, тем больше усложняются его определенные участки. Представьте, что программа «проводит опрос» пользователей: пользователь должен ответить на десяток вопросов. Но есть такие вопросы, отрицательные ответы на которые уже будут означать, что продолжать дальше опрашивать пользователя бессмысленно. В таком случае имеет смысл прекратить опрос сразу же после получения такого ответа. В программном коде это будет иметь вид экстренного выхода из цикла. Для этого в Java существует несколько *операторов перехода*: `break`, `continue`, `return`.

Оператор перехода `break`

Оператор `break` используется в трех случаях. Первый случай – совместное использование с оператором условия `switch`. Там он прерывал прохождение проверок с помощью `case`. Рассмотрим следующий код:

```
int a = 2;
switch (a)
{
    case 1:
        System.out.println («Проверяем 1»);
    case 2:
        System.out.println («Проверяем 2»);
    case 3:
        System.out.println («Проверяем 3»);
    default:
        System.out.println («default»);
}
```

Как вы думаете, что он выведет в результате? Для некоторых будет сюрпризом, но он выведет:

```
Проверяем 2
Проверяем 3
default
```

Именно так работает `switch-case`: при попадании в `case` где значение совпадает (в нашем случае 2) он будет выполнять все следующие инструкции уже игнорируя какие-либо `case` и `default`. А для того, чтобы получить корректно работающий код нам надо расставить везде `break`, кроме случая с `default`, потому что `default` это «последний случай»:

```
int a = 2;
switch (a)
{
    case 1:
        System.out.println («Проверяем 1»);
```

```
break;
case 2:
System.out.println («Проверяем 2»);
break;
case 3:
System.out.println («Проверяем 3»);
break;
default:
System.out.println («default»);
}
```

Теперь если мы запустим, то получим изначально ожидаемый вывод:

Проверяем 2

Второй случай, который более важен и чаще используется – выход из цикла. Например, программа в цикле считает среднюю оценку учеников 1-х классов. И вдруг встречается оценка, которая ну точно не может быть (малолетний хакер взломал систему, но ошибся и поставил себе 99 баллов). Использование таких данных однозначно приведено к некорректному финальному результату. Единственно правильный выход – это *выход*, т.е. написать, что используются неверные данные и выйти из цикла.

Вот простая иллюстрация:

```
for (int i = 0; i < 100; i++) {
if (i == 3)
break;
System.out.println («i: » + i);
}
System.out.println («Конец цикла.»);
```

Как мы видим, цикл определен так, чтобы выполнять 100 итераций. Но по факту, выполниться всего 3 итерации и как только счетчик **i** достигнет значения 3, сработает проверка if (**i == 3**) и вызовется оператор **break**. Все, что распечатает этот код:

i: 0
i: 1
i: 2
Конец цикла

Но **break** выполняет выход только из текущего цикла. Вот еще один пример кода:

```
for (int i = 0; i < 2; i++) {
System.out.print («Итерация » + i + «:»);
for (int j = 0; j < 100; j++) {
if (j == 3)
break;
System.out.print (j + «»);
}
System.out.println ();
}
System.out.println («Конец цикла.»);
```

Этот код выведет:

Итерация 0: 0 1 2
Итерация 1: 0 1 2

Цикл завершен.

Т.е. внутренний цикл прерывается, а внешний цикл делает все свои итерации. Обратите внимание что в коде используется метод **print**, в отличии от **println**, он продолжает печатать в текущей строке, а не переходит сначала на новую строку.

Третий случай, когда применяется **break**, переход к строке кода с меткой. Тоже довольно редкий случай, потому что большое количество таких переходов могут образовывать, так называемый, спагетти-код – неприятную мешанину нечитаемой логики. Но, тем не менее, стоит знать о такой возможности. Вот пример кода:

```
hello:  
{  
    for (int i = 0; i < 2; i++) {  
        System.out.print («Итерация » + i + «:»);  
        for (int j = 0; j < 100; j++) {  
            if (j == 3) {  
                break hello;  
            }  
            System.out.print (j + «»);  
        }  
        System.out.println ();  
    }  
    System.out.println («Конец цикла.»);  
}
```

Сначала все обрамляется в блок кода, с помощью фигурных скобок и помечается меткой **hello**. А дальше, код, такой же, как и предыдущий, за одним исключением: `break hello;` – это как раз и позволяет выйти не из текущего цикла, а целиком из помеченного блока кода. В результате код выведет:

Итерация 0: 0 1 2

Т.е. даже не появится надписи Конец хода, потому что она тоже входила в помеченный блок кода.

Оператор перехода **continue**

Оператор перехода **continue**, служит для пропуска текущей итерации цикла и сразу перехода к следующей. Например, необходимо распечатать только четные числа в диапазоне от 1 до 20:

```
for (int i = 1; i <= 20; i++) {  
    if (i%2!= 0) continue;  
    System.out.print (» " + i);  
}
```

Создается цикл от 1 до 20 включительно. В следующей строке выясняется текущее значение счетчика является четным числом или нет (четное число имеет нулевой остаток от деления на 2) и если нечетное -вызывается оператор перехода **continue**, и как следствие следующая строка игнорируется и сразу же начинается следующая итерация.

Оператор перехода **return**

Оператор перехода **return** сразу прекращает выполнение текущего метода, выходит из него. Так как мы еще не рассматривали детально что такое метод и не использовали методы, то можно сказать что оператор **return** просто прекращает выполнение программы. Это может пригодиться, когда, например, пользователь ввел неверные данные и дальнейшая работа программы не имеет смысла.

```
for (int i = 1; i <= 20; i++) {  
    System.out.print (» " + i);  
    if (i == 1)  
        return;  
    }  
    System.out.println («Test»);
```

Такой код только выведет единицу (слово Test напечатано не будет), и программа завершится.

Глава 7. Массивы

До сих пор мы создавали программы, которые работали с одной, двумя, тремя переменными. А что делать если речь идет, например, о написании программы, которая должна находить средний бал по математике всех учеников в школе? А если речь идет о обработке голосов в политическом голосовании? Для решения таких задач, в любом языке программирования есть специальная структура данных, она называется массив и способна хранить огромное количество значений. Если о переменной можно думать, как о некой коробке в которую можно «положить» какое-то значение, то о массиве тогда стоит думать, как о шкафе с полками, в котором хранятся такие коробки и есть кладовщик с отличной памятью, который по **номеру** коробки даст вам ее содержимое. У массивов есть ограничения: простые массивы обычно хранят данные одного типа, т.е., например, массив целочисленных значений, массив строк, массив значений типа float. Второе ограничение: длина массива (или количество значений массива) на самом деле ограничена количеством доступной оперативной памяти, выделяемой операционной системой вашей программе. У вас на компьютере 4 гигабайта памяти, а вы хотите программу, которая имеет массив, занимающий 8 гигабайт – ваша программа вылетит\крашится\упадет\сломается потому что ей памяти столько операционная система не выделит. Особенno часто с этим сталкиваются разработчики программ для мобильных телефонов – там требования по памяти жестче.

Создание одномерного массива

Рассмотрим создание массива значений типа **int**.

```
int [] arr = new int [5];
```

Рассмотрим слева направо. **int []** – квадратные скобки [] это объявление массива, а **int** непосредственно тип массива. **arr** – имя массива. **new** – оператор который запрашивает виртуальную машину Java на выделение памяти, которой будет достаточно для хранения пяти значений типа **int** – **int [5]**. Важно помнить, что массивы в Java имеют неизменяемую длину – вы объявили, что массив может содержать 5 элементов и нет никакой возможности его расширить, кроме как создать массив с большим количеством элементов и скопировать поэлементно туда старый массив. Как мы понимаем это неудобно, зачастую мы можем не знать какую длину должен иметь массив, если задать больше – получиться перерасход памяти, задать меньше – в конечном итоге, что-нибудь не влезет. Для таких случаев в Java используются разного рода **коллекции**, с ними возможны различные манипуляции, вплоть до встроенных сортировок, они могут содержать элементы не только примитивных типов, но и объекты (они будут уже рассмотрены в третьей части).

Итак, мы создали массив из 5 целочисленных значений. При создании все 5 элементов имеют значение ноль. Каждый элемент массива имеет, так называемый индекс – это порядковый номер, по которому можно как присвоить, так взять значение. Индекс массива начинается с нуля (а не с единицы, как мы привыкли считать предметы в жизни.) После объявления массива напишем еще пару строк:

```
arr [2] = 10;  
arr [0] = 5;  
System.out.println (arr [0] + " " + arr [2]);  
// выведет: 5 10
```

Как мы видим из примера, можно задавать значения элементов в каком угодно порядке и также брать значения тоже как душе угодно. Не стоит пытаться обращаться к элементам мас-

сива по индексу большему чем 4 – вы получите `ArrayIndexOutOfBoundsException` ошибку, – человеческим языком запрет на обращение к тому, чего нет.

Все элементы массива можно вывести с помощью цикла:

```
for (int i = 0; i < 3; i++) {  
    System.out.println (arr [i])  
}
```

Выведет:

```
5  
0  
10
```

Как мы видим, верхний предел в цикле задается числом 3. А что случится если мы изменим длину массива? – придется также менять и число в цикле. Чтобы избежать «двойной работы» можно использовать *свойство length* – оно всегда хранит длину массива. Тогда наш код будет выглядеть так:

```
for (int i = 0; i < arr. length; i++) {  
    System.out.println (arr [i])  
}
```

Есть еще один способ создания массива, его удобно использовать, если значений немного:

```
String [] names = {"Петя", "Вася", "Женя"};
```

Таким образом создастся массив `names`, который будет содержать три элемента типа `String`.

Многомерные массивы

Массивы могут быть не только одномерные (линейные), но и многомерные. Одномерные массивы хранят списки элементов, многомерные – хранят списки из одномерных массивов. Одномерный массив – это книга (список из страниц), двумерный массив – это книжная полка, трехмерный массив – это книжный шкаф, четырехмерный массив – это книжный отдел в библиотеке, пятимерный массив – это библиотека и так далее. На практике вам редко придется пользоваться массивами с мерностью больше третьей, потому что для сложных данных используются сложные структуры (объекты), а не наращивание количества списков. Для игр очень популярны двумерные массивы – в них хранят ячейки игральной доски, поле боя, карту острова.

Создадим двухмерный массив типа `int`:

```
int [] [] m = new int [2] [3];  
m [0] [1] = 5;  
m [1] [2] = 10;
```

Как мы видим это очень похоже на создание одномерного массива, только добавились еще одни квадратные скобки, т.е. добавился еще один индекс. Стока `int [] [] m = new int [2] [3];` говорит создать массив `m`, состоящий из 2-х одномерных массивов, которые хранят значения типа `int`, и каждый из них состоит из 3-х элементов. По сути это матрица 2 на 3 (два ряда и три колонки). Дальше в коде в нулевом ряду, по индексу 1 присваивается значение 5; а в первом ряду по индексу 2 присваивается значение 10. Для того чтобы распечатать все значения массива, нам как раз пригодятся вложенные циклы:

```
for (int i = 0; i < 2; i++) {
```

```
System.out.println (<>ряд:" + i);
for (int t = 0; t < 3; t++) {
    System.out.println (m [i] [t]);
}
```

Первый цикл проходит по значениям от 0 до 2 (не включительно), т.е. по рядам. Потом выводиться индекс текущего ряда. Далее второй цикл проходит по элементам каждого ряда – от 0 до 3 (не включительно), – и выводит значение каждого элемента. Вот что мы увидим:

```
ряд:0
0
5
0
ряд:1
0
0
10
```

Также мы можем использовать *свойство* массива **length** и сделать наш код более *универсальным*:

```
for (int i = 0; i < m.length; i++) {
    System.out.println (<>ряд:" + i);
    for (int t = 0; t < m [i].length; t++) {
        System.out.println (m [i] [t]);
    }
}
```

Массив, который мы создали еще называется *прямоугольным* массивом, потому что все его ряды имеют одинаковое количество элементов. В Java возможно создание *непрямоугольных* массивов, например, так:

```
int [] [] N = new int [2] [];
N [0] = new int [2];
N [1] = new int [3];
```

Создается массив N, причем явственно объявляется, что он имеет 2 ряда, и только потом каждому ряду присваивается свой массив, в ряд с нулевым индексом присваивается массив длиной 2, в ряд с индексом один, присваивается массив длиной 3. Таким образом его ряды имеют разную длину. И вот тут как раз очень пригодиться именно использование свойства **length**, потому что, только так мы сможем узнать каждый раз длину ряда.

Задания

1. Создайте массив чисел от 1 до 100. Напишите цикл, который выводит элементы с нечетными индексами.

Модифицируйте программу чтобы она выводила элементы с четными индексами.

2. Задайте массив с тремя элементами. Создайте пустой массив длиной 5. С помощью цикла скопируйте значения элементов первого массива во второй, причем нулевые значения во втором массиве замените на значение 100. Выведите значение второго массива с помощью цикла.

3. Создайте программу которая выводит таблицу умножения на 2.

Модифицируйте программу чтобы выводилась таблица умножения на 5.

Попробуйте модифицировать программу чтобы выводилась полная таблица умножения.

4. Создайте программу которая находит в массиве элемент с наименьшим значением (массив вы должны создать сами, например, с десятью разными значениями).

Измените программу, чтобы она находила элемент с наибольшим значением.

5. Напишите программу которая меняет местами элементы массива – нулевой элемент становится последним, последний – нулевым, первый становится предпоследним, предпоследний – первым и т. д. Не используйте вспомогательный массив для этого.

6. Попробуйте написать программу, которая сортирует массив (располагает элементы массива) в возрастающем или убывающем порядке. Вы можете загуглить «пузырьковая сортировка» – это самый простой и понятный алгоритм сортировки.

7. Создайте программу которая выводит календарь текущего года: каждый месяц с числами и днями недели.

Глава 8. Ввод данных

До этой главы все наши программы работали с данными, заранее определенными в коде, в переменных. Все, что нам не хватает, для написания полноценных программ – это чтобы программа взаимодействовала с человеком (пользователем программы), т.е. воспринимала от него какую-то информацию. Это очень важно, все что мы делаем с компьютерами или мобильными телефонами – вводим информацию и ожидаем получить какой-то осмысленный результат.

В этой главе мы рассмотрим самый простой ввод данных, с помощью клавиатуры (есть еще с помощью мыши, пальцев, поворотов головы и т.д.). С помощью клавиатуры человек может вводить цифры (которые формируют числа) и символы, которые формируют строки. Один из самых простых способов запрограммировать пользовательский ввод – использовать объект класса Scanner. Создадим новый проект IntelliJ IDEA и назовем его SimpleUserInputExample и добавим несколько строк кода:

```
Main.java
1 package com.company;
2
3 import java.util.Scanner;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         Scanner scn = new Scanner(System.in);
9         System.out.println("Введите число:");
10        int n = scn.nextInt();
11        System.out.println("Вы ввели:" + n);
12    }
13 }
```

Если мы запустим программу и сделаем то, что она просит то, получим:

```
Run: Main
C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.1\jbr\bin
Введите число:
10
Вы ввели:10
Process finished with exit code 0
```

Теперь разберем программу строчка за строчкой. Стока 8:

```
Scanner scn = new Scanner(System.in);
```

Здесь объявляется переменная **scn** типа Scanner, или иными словами переменная которая ссылается на *объект* класса Scanner. Ранее уже говорилось, что есть так называемые ссылочные объекты, они занимают какой-то участок памяти, а в переменной хранится ссылка на этот участок памяти. В Java нет нужды думать о таких деталях, поэтому вы можете считать, что переменная просто хранит объект.

new Scanner

Оператор new занимается тем, что говорит виртуальной машине: «эй! выдели память под объект класса Scanner, создай его там и верни ссылку на него». Далее еще интереснее:

```
new Scanner(System.in)
```

Дело в том, что Scanner обрабатывает не только ввод данных с клавиатуры. Это универсальный объект, и он может принимать данные из консоли, из строки, из файла. И поэтому необходимо указать явно источник данных. В нашем случае таким источником данных выступает **System.in** – это неформатированный входной поток байтов (кодов нажатых клавиш на клавиатуре), которые переводятся Scanner'ом в данные, которые нам нужны (в числе с помощью метода **nextInt**). Еще раз строка 8 означает создание объекта Scanner с указанием откуда брать входные данные.

Строка 9 – просто вывод предложения ввести число.

Строка 10:

```
int n = scn.nextInt();
```

Первая часть, это как мы уже знаем объявление переменной типа int. Вторая часть:

```
scn.nextInt();
```

это вызов метода **nextInt ()** объекта Scanner. Таким образом мы просим объект Scanner: «верни нам, то что введут в виде целого числа» и он, повинуясь нашей команде, переводит программу в режим ожидания – программа ждет, пока мы введем *что-то* и нажмем клавишу Enter.

Строка 11 – просто вывод значения переменной **n**.

Вернемся к строке 10. Когда программа ее выполняет и ждет вашего ввода, вы не обязаны вводить число! вы можете ввести что вам угодно. Вы можете ввести, например, букву или слово и нажать Enter, но тогда вы получите ошибку, т.к. мы просили Scanner отдать нам число, а слово в число превратится ну никак не может. Попробуйте сделать так: запустите программу и введите букву A. И вот что выйдет:



Программа выдала ошибку **InputMismatchException** (ошибку несовпадения типов). И все бы ничего, если мы делаем программу для себя, то мы знаем, когда программа просит ввести число, то *надо* ввести число, когда просят ввести слово, то *надо* ввести слово. Но это если этой программой будет пользоваться кто-то еще, скажем так обычный человек, то он может по какой-то причине, намеренной или случайной (по невнимательности) сделать ошибку и тогда программа перестанет работать.

И здесь возникает самая большая проблема разработки программ, причина по которой срываются сроки выпуска программ и игр, причина трата огромных денежных сумм – пользователи очень часто делают не так как задумывали программисты и программисты вынуждены писать огромные куски кода, которые занимаются обработкой различных случаев неправильных, с точки зрения программы, действий пользователя. В каждой серьезной компании по разработке программ есть отдел QA (quality assurance – тестирования качества, или по-простому – тестеры). Тестеры – это самые любимые и одновременно ненавидимые люди для программистов. Любимые, потому что делают огромную работу по выявлению ошибок, ненавидимые, потому что 99.9999% новостей от них – это плохие новости :) Стоит заметить, что профессиональный тестер, это тоже очень востребованный человек, это не «обезьянка бьющая по кнопкам».

кам в надежде сломать программу», это – человек который может грамотно организовать процесс автоматического! тестирования, и для этого ему тоже приходиться программировать.

Так что же делать в нашем случае, когда мы хотим, чтобы наша программа не вываливалась с ошибкой, от каждой нашей опечатки. Для этого **Scanner** обладает тоже соответствующими методами проверки. Прежде чем взять у него число, мы можем спросить «а число ли там у тебя?». Вот как будет выглядеть измененный код:

```

1 package com.company;
2
3 import java.util.Scanner;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         Scanner scn = new Scanner(System.in);
9         System.out.println("Введите число:");
10        if(scn.hasNextInt()) {
11            int n = scn.nextInt();
12            System.out.println("Вы ввели: " + n);
13        }
14        else
15            System.out.println("Ну это определенно не число!");
16    }
17}

```

Самое главное изменение в строке 10:

```
if (scn.hasNextInt ())
```

Тут стоит проверка, которая проверяет что вернет метод **hasNextInt ()** – Scanner спрашивают «а точно там целочисленное значение?» и он вернет *true* если пользователь введет число, и дальше тогда Scanner попросят отдать это число. Если же там не число, то вернется *false* и программа перейдет в строку 15 и выведет уведомление об этом.

Если вы хотите связать свою профессиональную деятельность с программированием именно на Java, вы должны знать, что большую часть программирования в ней занимает именно обработка ошибок. Например, чтобы написать на Java самый простой код, по открытию файла и считывания информации из него, вам понадобится написать десятка два строк кода, в Python – только одну. Java надежна, безопасная и именно поэтому код многословен.

В нашей программе есть определенное неудобство – если человек ошибся, то ему заново надо запускать программу. А если в программе надо ввести не одно значение, а десять? – он может сделать ошибку и на восьмом, и на девятом и на десятом вводе и тогда он будет терять время на повторный ввод всех значений. Есть простое решение для такой проблемы – использование цикла **while**, который будет повторяться до тех пор, пока пользователь не введет удовлетворительное значение. Причем это касается не только проблемы с несовпадением типов данных, но и, например, соблюдения каких-либо логических ограничений.

Напишем программу, которая проверяет возраст пользователя и дает ему доступ к «взрослой информации», если тому больше 18 лет. Создадим проект в IntelliJ IDEA и назовем его **AdultAgeCheck**. Вот весь код программы:

```

1 package com.company;
2 import java.util.Scanner;
3 public class Main {
4     public static void main(String[] args) {
5         int age = 0;
6         do {
7             System.out.println("Для доступа к разделу введите ваш возраст:");
8             Scanner scn = new Scanner(System.in);
9             if (!scn.hasNextInt()) {
10                 System.out.println("Вам нужно ввести целое число.");
11                 continue;
12             }
13             age = scn.nextInt();
14             if (age < 1)
15                 System.out.println("Вы еще не родились!");
16             if (age > 115)
17                 System.out.println("На планете Земля еще не был зарегистрирован такой долгожитель!");
18         } while (age < 1 || age > 115);
19         if (age < 18)
20             System.out.println("Вам надо еще немного подрасти.");
21         else
22             System.out.println("[Тут может быть смешная взрослая шутка]");
23     }
24 }

```

Рассмотрим код построчно. Объявляем и инициализируем (присваиваем начальное значение) переменную `age`. Строки 6—19 работают цикл **do-while** – это тот самый редкий случай, когда он здесь полезен, потому что *сначала* надо спросить пользователя, и только потом проанализировать его ввод. Стока 7 просто приглашение пользователя ввести что требуется. Стока 8 – создание `Scanner`, для считывания возраста пользователя. Стока 9 – «защита от дурака» (программистский термин, обозначающий такой тип проверок). Если пользователь ввел не целое число, а, например, текст, программа говорит об этом и возвращается в начало тела цикла с помощью `continue`, и вопрос пользователю повторяется. Если же пользователь ввел целое число, то следуют еще две проверки – чтобы игнорировать заведомо неверные значения: отрицательные, нулевые и слишком и вывести предупреждение пользователю. Пользователю надо всегда давать информацию что он сделал неправильно – это одно из правил юзабилити (построения правильных пользовательских интерфейсов). Далее следует оператор `while` тоже с проверкой допустимого значения, если значение не попадает в диапазон от 1 до 114, то цикл повторяется и вопрос задается снова. Если значение удовлетворяет условиям, то производится финальная проверка уже непосредственно для доступа к информации (то, из-за чего все затевалось). Если значение меньше 18 выводится одно сообщение, если больше – другое (ну или был бы, например, уже вызов какой-то функции или переход на другой экран).

На примере такой простой программы вы как раз можете увидеть сколько усилий требуется для того, чтобы избежать ошибок неправильного ввода данных. Конечно использование графического интерфейса и мыши могут облегчить взаимодействие с пользователем, но там тоже есть свои нюансы, особенно когда речь идет о сложных формах ввода. Вы сами можете увидеть это в банках, даже простые банковские операции требуют большого количества действий от человека.

Заключение к первой части книги

Эта глава является последней главой первой части книги. Обычно в книгах по программированию не делают на этом акцент, но в данном случае для этого есть причины.

Первая причина: то, что было рассказано в этих восьми главах **достаточно** для того, чтобы вы смогли писать полноценные программы, начиная от реализации простейших алгоритмов, заканчивая компьютерными играми (пусть в текстовом режиме). По-хорошему вы должны были выполнить все упражнения, чтобы почувствовать нравится ли вам программирование, нравится ли вам программирование на Java. Это очень важные чувства – очень сложно заниматься программированием, когда оно тебе не нравится. Может причина в том, что вы чего-то не поняли до конца и тогда стоит поискать ответы на свои вопросы в интернете, в группах ВК, в телеграмм-чатах – информации море, я по себе знаю, что порой не хватает объяснения, которое написано именно так, чтобы понял именно я.

Программист – это человек, который пишет программы (спасибо кэп), и чем больше вы пишите программ, тем больше вы совершенствуетесь как программист. Конечно, необходимым условием является постепенный и постоянный рост уровня сложности. Попробуйте очень хороший сайт <https://www.codewars.com/>, там огромное количество задач, решения к которым можно писать практически на всех языках программирования. Там стоит начать с 8 kyu (самый легкий уровень) до 5—4 kyu (уровни после которого идут задачи уже олимпиадного программирования и для коммерческого программирования не пригодны) включительно. Стоит также решать там задачи, если вам нужно подготовиться к собеседованиям, обычно спрашивают типовые задачи как на codewars. Еще вы можете попробовать проект Эйлера – это набор математических задач для программистов разной сложности.

Конечно, может случиться так, что вам не понравилась сама Java. Тогда смело можете пробовать любой другой язык – знаний первых 8 глав достаточно для того, чтобы с легкостью погрузиться и попробовать, что вы считете более интересным. К тому же, будет уже с чем сравнить. На самом деле, самый сложный язык программирования – это *первый* язык, – а после того как стал в нем специалистом, остальные языки выучиваются с удвоенной-утроенной скоростью.

Задания

В этот раз задания будут сложнее.

1. Напишите программу, которая принимает от пользователя положительное число, и считает сумму всех чисел от 1 до введенного числа. Например, пользователь ввел 5, программа должна выдать ответ: « $1 + 2 + 3 + 4 + 5 = 15$ »
3. Напишите программу которая «раздает» из карточной колоды в 52 листа 5 карт. Вы должны создать массив, который содержит 52 карты. Каждая карта представлена в виде строки, с форматом значение-масть, например, «Туз бубен», «6 пик», «2 треф», «Валет червей». Потом надо перемешать этот массив и после вывести первые пять элементов массива.
4. Модифицируйте программу по раздаче карт, чтобы она могла «сдавать» карты нескольким игрокам – т.е. по 5 карт несколько раз.
5. Напишите программу «больше-меньше», которая пытается угадать задуманное человеком число. Человек загадывает число, программа выдает свою отгадку и спрашивает угадала ли она, если нет, то спрашивает было ли отгадка больше или меньше задуманного числа, и потом все повторяется заново.
6. Напишите программу которая находит все простые числа (числа которые делятся только на единицу или на самих себя) в заданном диапазоне
7. Напишите программу которая конвертирует обычные десятичные числа в римские. Например, вводиться 17, и программа выдает XVII.
8. Напишите программу которая играет в Крестики-нолики
9. Напишите программу которая выигрывает в игру Орел-решка. Человек загадывает число 0 или 1, а программа должна угадать. Игра ведется с положительным балансом в сторону человека.
10. Напишите программу которая конвертирует римские числа в обычные десятичные числа. Например, вводиться XXIII и программа выдает 23.

Часть II

Глава 9. Объектно-ориентированное программирование

Добро пожаловать всем тем, кто решил продолжать и не бросил книгу, во вторую часть. Все что мы проходили, пробовали и писали в первой части, можно кратко назвать – «мы занимались функциональным программированием». Все наши программы имели простую структуру и исполнялись линейно, и мы решали простейшие задачи, не требующие наличия сложных объектов. В этой же части мы переходим к теории и практике создания программ, основанных на принципах объектно-ориентированного программирования, *де-факто* индустриальному стандарту создания программного обеспечения на текущий момент.

Ранее уже было краткое введение об ООП (объектно-ориентированном программировании), но на самом деле это огромная тема для изучения, потому как именно только с использованием ООП можно создать большие программные комплексы, которые можно развивать и поддерживать годами. Коммерческое программирование – это не математика и алгоритмы, а в первую очередь умение и желание разбираться и понимать, как выстроена архитектура приложения\системы. По ООП написаны сотни книг про то, как правильно строить архитектуру проекта, какие паттерны проектирования (т.е. уже проверенный опытным путем схемы взаимодействия объектов) использовать, как проводить рефакторинг (оптимизацию архитектуры проекта). Тема очень большая, поэтому в этой книге только приводиться объяснение реализации базовых принципов ООП средствами языка Java.

Главная идея использования ООП: программа – это совокупность некоторых объектов, которые общаются между собой. ООП это эволюционный шаг в понимании как надо писать программы для компьютеров. Первый шаг был написание программ практически на языке команд микропроцессора – язык ассемблер, в котором программисту надо было думать на уровне ячеек памяти и атомарных операций по копированию из одной ячейки в другую. Потом появилось функциональное программирование – программа состояла из набора функций и библиотек функций, которые писал и которыми оперировал программист. А потом появилось ООП, наиболее подходящая абстракция для написания программ: в реальном мире мы манипулируем объектами и используем их свойства и способности. Более того программы с ООП проще читаются, проще понимаются, проще исправляются; и это очень весомые аргументы, которые более важны, чем, например, скорость исполнения программ. Программа, написанная на ассемблере будет работать в несколько раз быстрее, но на ассемблере не пишут программы для управления сервером, срок разработки такой программы увеличился бы в десятки раз.

Классы и объекты – основные понятия

Классы в Java – это фундамент на котором строится весь язык. Класс – это шаблон (форма, заготовка) по которому создаются объекты. В классе определяются данные и код, которые выполняют действия над этими данными. Данные хранятся в атрибутах класса, код – в методах класса. Объекты – это экземпляры классов. Таким образом класс, это фактически описание (спецификация), в соответствии с которым, должны создаваться объекты. Важно понимать, что класс – это логическая абстракция, физическое представление класса в оперативной памяти компьютера возникнет лишь после того, как будет создан объект этого класса.

Нет никаких ограничений на количество атрибутов и количество методов, которые могут быть в классе. Вполне возможны случаи, когда класс содержит только атрибуты или только методы.

Для написания программ в первой части нам было не нужно использовать классы, потому что программы были простые. Все что было нужно – это придумать как решить задачку. Но когда речь идет о сложных приложениях, программист, прежде чем сесть и начать писать код, продумывает какие классы надо запроектировать, какую иерархию отношений между классами надо выстроить. Конечно это звучит устрашающе, но на самом деле нет повода волноваться: это приходит с опытом, и все что вам нужно делать это – ABC, Always Be Coding. В основе архитектуры любого приложения, написанного на Java (и не только), лежат четыре принципа ООП.

Принципы ООП: «Три кита на одной черепахе»

По классике ООП основан на четырех принципах: инкапсуляция, наследование, полиморфизм и абстракция – «три кита на абстрактной черепахе». И о них надо уметь рассказать, даже если вас разбудят среди ночи. Ну а если серьезно, это один из вопросов на собеседованиях на работу (и честно говоря выглядит очень странно если человек, который полгода-год обучаясь Java не может о них рассказать).

Итак, **абстракция** – это отделение важного от несущественного. Абстракция – это выделение в моделируемом предмете важного для решения конкретной задачи. Если вам для проекта нужны часы, которые считают время, тогда при создании класса Часы, вам не нужно думать, как работают шестеренки в механизме часов, а важно только сосредоточиться на реализации счета времени. Но если вы будете делать симулятор механизма часов, тогда вам придется задуматься о том, как работают шестеренки.

Инкапсуляция – это заключение атрибутов и методов объекта в сам объект, таким образом атрибуты и методы имеют контекст и не существуют сами по себе. Это дает возможность программисту определить какие атрибуты и методы доступны для пользователей проектируемого класса, а какие должны быть скрыты, чтобы ничего не сломать.

Наследование – самый «осозаемый» принцип в Java, – принцип, согласно которому, объект может наследовать атрибуты и методы другого объекта, таким образом способствуя повторному использованию кода. Вы описываете объект Car (машина) достаточно детально, а потом наследуетесь от него в Truck (грузовик) и Automobile (легковая машина) и в них уже будут присутствовать все свойства и методы объекта Car и вы сможете заняться уже непосредственной детализацией грузовика и легковушки.

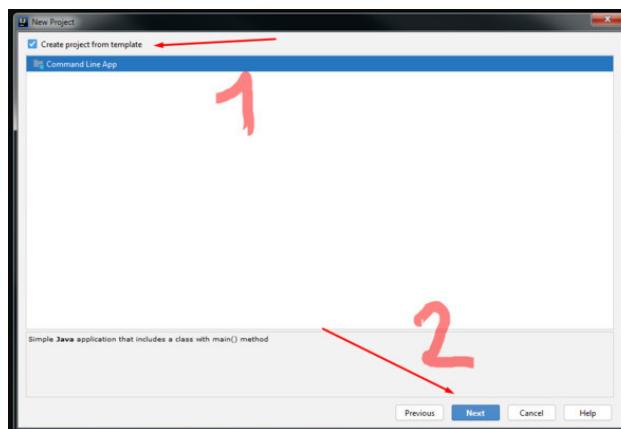
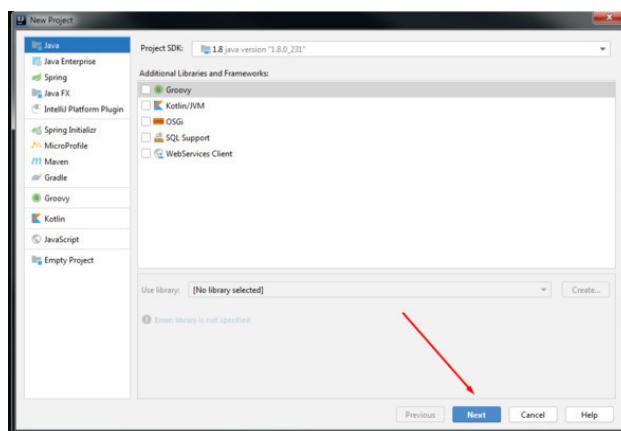
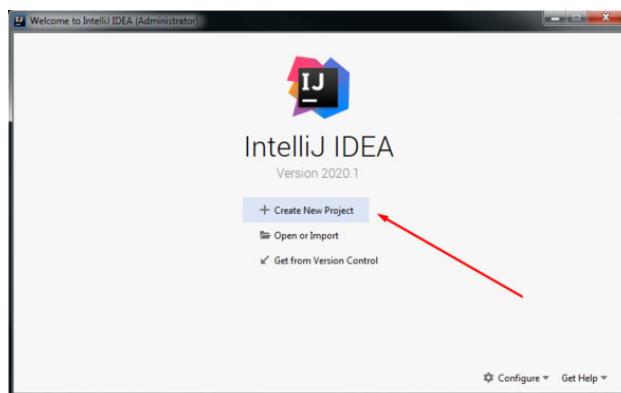
Полиморфизм – значение в переводе «иметь много форм» и от этого мало полезного смысла. На самом деле полиморфизм в программировании, это когда вы можете вызывать одинаковый метод, особо не интересуясь у кого вы его вызываете. И, честно говоря, это тоже объяснение так себе. Поэтому вот пример. У вас есть объект Animal (животное), от которого вы наследуете Dog (собака), Cat (кот) и Cow (корова). В объекте Animal есть метод talk (говорить) и соответственно он «перейдет» наследникам. Но есть проблема собака гавкает, кот мяукает, а корова мычит. И поэтому в каждом объекте метод talk будет переопределен (еще говорят перегружен) чтобы каждый произносил звуки как положено. Далее мы хотим собрать всех этих животных в зоопарк и в какой-то момент заставлять их всех «говорить». Для этого у нас будет массив объектов Animal, пробегаясь по которому, мы будем вызывать метод talk. И вот именно благодаря принципу инкапсуляции, заложенному в языке Java, будет вызван конкретный метод talk в зависимости от объекта.

Да все это звучит пока что сложно, запутанно и абстрактно и именно поэтому в следующей главе мы начнем строить свой собственный зоопарк с животными, над которыми и будем проводить объектно-ориентированные эксперименты.

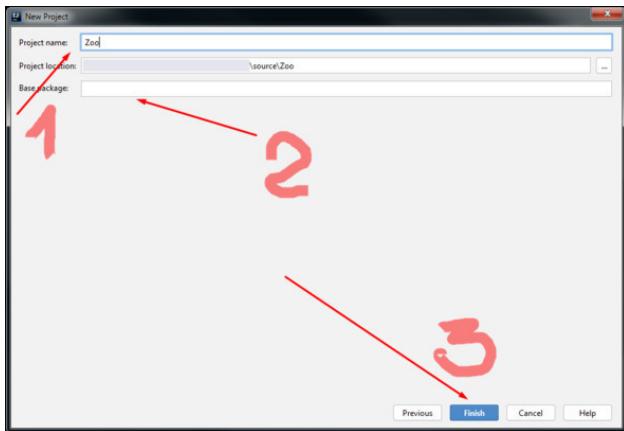
Глава 10. Наш зоопарк

Заранее хочу предупредить, проект Зоопарк, будет самым большим в этой книге, потому что на основе его будут объяснены и показаны на практике все ООП идеи в Java. Очень рекомендуется **внимательно читать и повторять описанные действия**.

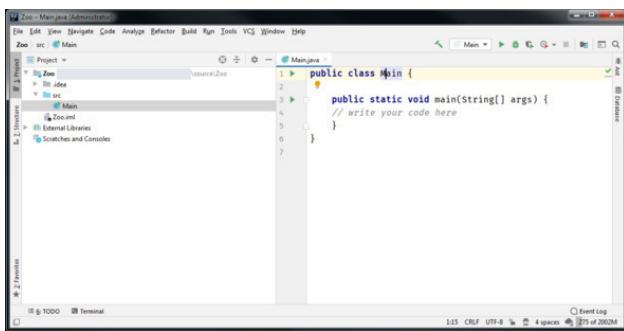
Создадим новый проект в IntelliJ IDEA и назовем его Zoo (зоопарк). Несмотря на то, что в первой части уже было расписано в картинках как создавать проект в IntelliJ IDEA, я позволю себе еще раз это сделать, только уже без объяснений. Повторение – мать учения;).



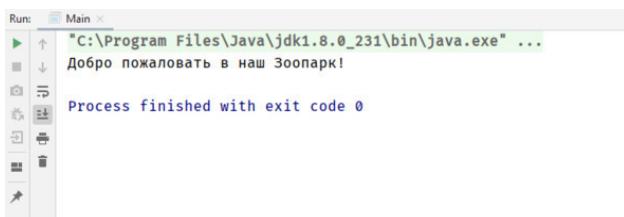
Обратите внимание: надо удалить все в поле **Base package**



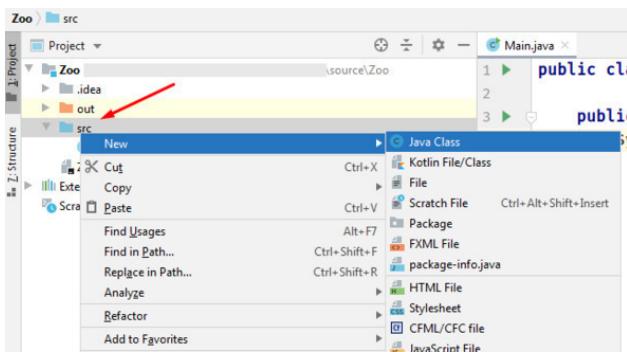
И вот что вы должны получить в финале:



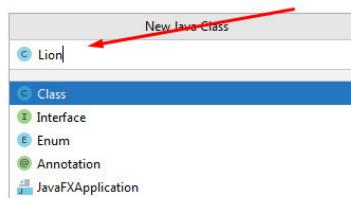
Для красоты добавим внутрь метода main строчку кода:
System.out.println («Добро пожаловать в наш Зоопарк!»);
чтобы все «было красиво» после запуска программы:



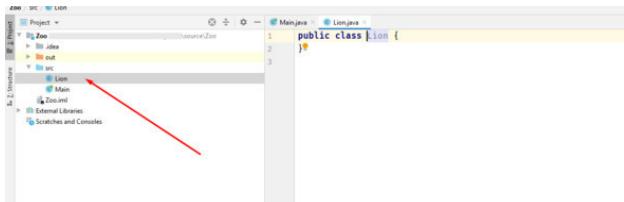
Как всем известно, в зоопарк ходят смотреть животных, и в наш зоопарк надо срочно поселить кого-нибудь. Начнем с «царя зверей» и сделаем то, что еще никогда не делали! – создадим новый класс сами. Для этого в панели **Project**, кликнем правой кнопкой мыши на папке **src**, выберем там пункт меню **New** и в подпункте **Java Class**.



В открывшемся окошке введем **Lion**



и нажмем клавишу Enter. В результате в нашем проекте появится новый класс **Lion**, у которого пока что нет ни атрибутов, ни методов.



Трудно ли быть Творцом?

Итак, у нас есть пустой шаблон – класс **Lion**, который должен стать Львом в нашем Зоопарке. Давайте подумаем, что у нас будет делать Лев. В первую очередь на ум приходит, что Лев прям-таки обязан громко рычать, даже так: **РЫЧАТЬ**. Также наш Лев должен быть большим, маленький котенок ну никак не подойдет на роль «царя зверей». Ну и он конечно должен хорошо кушать, чтобы быть сильным. И он конечно должен иметь массу тела, иначе после создания он улетит, вместо того чтобы жить в нашем зоопарке. Теперь посмотрим, как это все реализуется на языке программирования Java.

Для того, чтобы понимать насколько наш Лев большой, создадим в классе **Lion**, соответствующий атрибут **height** (рост):

```
public float height;
```

И вот уже мы видим новое слово **public**. Это *модификатор доступа*, таким образом определяется видимость атрибута снаружи. **public** мы пишем тогда, когда хотим, чтобы этот атрибут был доступен для использования за пределами класса, в котором он определен. Если бы мы хотели, чтобы никто не знал, как огромен наш Лев, тогда мы написали **private**. Дальше мы видим уже знакомый нам **float**, рост Льва у нас будет задаваться в метрах, и потом уже идет непосредственно название атрибута **height**.

Второй атрибут, который необходимо добавить – масса тела. Тут поступаем аналогично:

```
public float weight;
```

как мы видим изменилось только название атрибута – `weight` (вес), он будет в килограммах. Как можно заметить, я говорю «будет в метрах», «будет в килограммах» – я сам наделяю физическим смыслом значения, а для компилятора Java это всего лишь числа с плавающей запятой.

Теперь перейдем к созданию методов. Наш уже «материальный» Лев, обязан заявить о своем присутствии с помощью рычания:

```
public void growl()
{
    System.out.println ("G-R-R-R!!!");
```

снова *модификатор доступа* **public** и еще одно новое слово **void**: означает что метод ничего не возвращает в процессе своего выполнения. Дальше у нас будут методы, которые нам будут возвращать значения. После **void** идет название метода `growl` (рычать). И после названия две скобки () которые показывают, что не надо ничего *передавать внутрь метода*. Потом идет *тело метода*, оформленное фигурными скобками {}. Тело метода содержит операции, которые вы хотите чтобы происходили. В нашем случае со Львом, мы хотим, чтобы он «рычал» и это реализуется с помощью вывода в консоль строки «`G-R-R-R!!!`».

И второй метод «лев кушает»(eat):

```
public void eat (float food)
{
    weight += food;
```

метод публичный (public), он также не возвращает ничего (void), но уже принимает *параметр* `food` типа **float**. Т.е. передается количество пищи в метод, и уже в теле метода, это количество прибавляется к массе **weight** (у нас теоретический лев, который питается со 100% КПД).

Вот, в итоге, какого Льва мы получили и все что нам остается наконец-то поселить его в нашем Зоопарке.

```
1  public class Lion {
2      public float height;
3      public float weight;
4
5      public void growl()
6      {
7          System.out.println("G-R-R-R!!!");
8      }
9
10     public void eat(float food)
11     {
12         weight += food;
13     }
14
15 }
```

Для этого перейдем в класс Main и в методе main напишем следующие строки, после нашего приветствия:

```
Lion lion = new Lion ();
lion.growl ();
```

```
1 public class Main {
2
3     public static void main(String[] args) {
4         System.out.println("Добро пожаловать в наш Зоопарк!");
5
6         Lion lion = new Lion();
7         lion.growl();
8     }
9 }
10 }
```

В первой добавленной строке Lion lion означает что мы объявляем переменную типа Lion (или можно сказать класса Lion) и сразу же присваиваем этой переменной объект (!) который создаем путем использования ключевого слова **new**. Т.е. запись new Lion () приказывает виртуальной машине Java выделить память под объект, который создается согласно описанию в классе Lion. Когда JVM это сделает, она вернет ссылку на этот участок памяти и эта ссылка присвоится переменной lion. Поэтому любая переменная, которой «присвоен объект» это, на самом деле, переменная ссылочного типа и содержит она ссылку на область памяти, в которой хранится созданный объект.

Вторая добавленная строка lion.growl () – это, как вы понимаете, вызов метода **growl** объявленного в классе Lion. Запустим программу и вот что программа выведет в консоль:

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
Добро пожаловать в наш Зоопарк!
G-R-R-R!!!
Process finished with exit code 0
```

Обратите внимание – обращение к атрибутам и методам объекта происходит через символ».» точка. Просто есть языки программирования, в которых используется другая запись для этого.

Теперь давайте покормим нашего льва, насыплем ему 10 кг еды и посмотрим сколько он стал весить:

```
lion.eat(10);
System.out.println(lion.weight);
```

Вот что получим в консоли:

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
Добро пожаловать в наш Зоопарк!
G-R-R-R!!!
10.0
Process finished with exit code 0
```

Как мы видим наш лев весит всего 10 килограммов, после того как ему дали 10 килограммов, а все потому что изначально его вес не был задан. Исправим это недоразумение и зададим ему вес прежде чем что-либо с ним делать, добавим после создания объекта lion следующую строку:

lion. weight = 50;

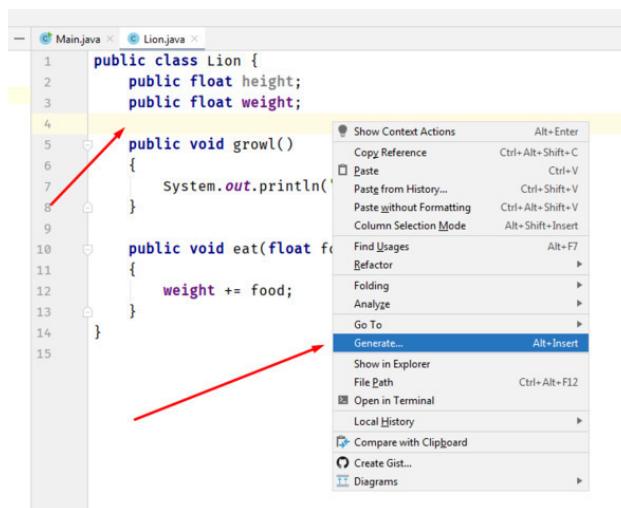
теперь наш лев изначально весит 50 килограмм, и после запуска программы мы уже увидим, что после кормежки он весит 60 килограмм.

Конструктор класса

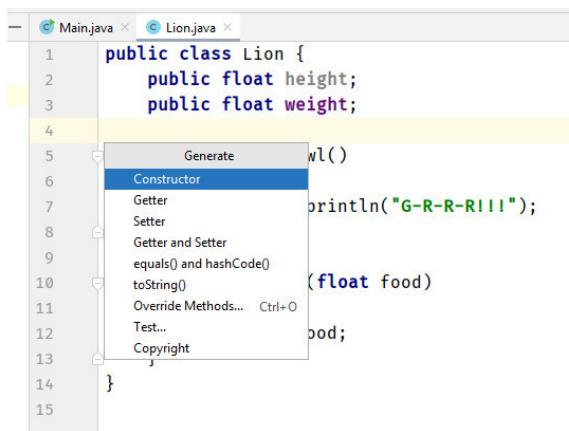
Я думаю вам, как и мне показалось что несколько неудобно задавать вес созданному льву отдельно, после его создания. А если представить, что нужно еще задавать, например, его цвет, рост, возраст, предпочтения в еде... так можно что-нибудь и забыть. И вот как раз для того, чтобы изначально задать необходимые параметры на момент создания объекта, существует **конструктор класса**.

Конструктор класса – это особый метод, но вся его особенность стоит всего лишь только в двух моментах: 1) он имеет имя, совпадающее с именем класса; 2) у него отсутствует начисто тип возвращаемого значения, потому что JVM возвращает ссылку на объект и этому ничто не должно мешать; 3) и самое главное, он вызывается **автоматически (!)** при создании объекта.

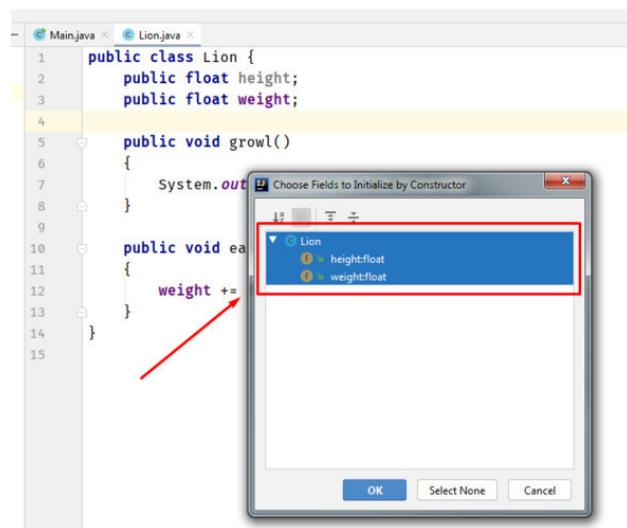
Давайте создадим конструктор для класса Lion. И теперь немного «уличной магии»: мы не будем писать код, а воспользуемся для этого IntelliJ IDEA (именно за это ее любят программисты) – в классе Lion, над объявлением метода growl, кликните правой кнопкой мыши, выберите Generate:



потом в появившемся окошке выберите Constructor:



А потом в еще одном новом оконке выберите все строчки (через клик + Shift):



В итоге будет добавлен следующий код:

```
public Lion (float height, float weight) {  
    this. height = height;  
    this. weight = weight;  
}
```

«Трупрограммеры» (true programmers или настоящие программисты) могут написать этот код руками. В итоге весь код класса Lion приобретет следующий вид:

```
1 public class Lion {  
2     public float height;  
3     public float weight;  
4  
5     public Lion(float height, float weight) {  
6         this.height = height;  
7         this.weight = weight;  
8     }  
9  
10    public void growl()  
11    {  
12        System.out.println("G-R-R-R!!!");  
13    }  
14  
15    public void eat(float food)  
16    {  
17        weight += food;  
18    }  
19}  
20
```

Так как конструктор изменился, то нужно переписать создание объекта в классе Main: сейчас там IntelliJ IDEA показывает ошибку, что теперь в конструктор надо передавать два аргумента:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         System.out.println("Добро пожаловать в наш Зоопарк!");  
5  
6         Lion lion = new Lion();  
7         lion.weight = 50; Expected 2 arguments but found 0  
8         lion.growl();  
9         lion.eat( food: 10 );  
10        System.out.println(lion.weight);  
11    }  
12}
```

Перепишем эту строку:

Lion lion = new Lion (0.4f, 50);

и уберем строку с присвоением **weight**. Теперь наш лев имеет и рост – обратите внимание, я после числа 0.4 еще добавил букву **f** – таким образом говоря JVM что это число с плавающей запятой типа **float**, иначе JVM она будет считать его типом **double**. В итоге код будет выглядеть вот так:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         System.out.println("Добро пожаловать в наш Зоопарк!");  
5  
6         Lion lion = new Lion( height: 0.4f, weight: 50 );  
7         lion.growl();  
8         lion.eat( food: 10 );  
9         System.out.println(lion.weight);  
10    }  
11}
```

Объект this

Все прекрасно в получившемся конструкторе класса Lion. Как мы видим это: публичный метод (модификатор public), отсутствует тип возвращаемого значения (после public сразу идет имя метода, совпадающее с именем класса Lion), дальше передается два аргумента height и weight. Только одно смущает – наличие в методе странного слова **this**.

this, это особая переменная, которой автоматически присваивается ссылка на объект в котором она находится. Т.е. в классе **Main** переменная **lion** содержит ссылку на созданный объект класса **Lion** и это позволяет обратится к объекту везде за пределами объекта. А вот **this** как раз позволяет обратится к объекту внутри объекта. Это удобно в нескольких случаях. Например, в нашем, когда имена передаваемых аргументов совпадают с атрибутами класса, иначе как тогда присвоить атрибуту **height** значение аргумента **height**?

```
height = height
```

так работать не будет, JVM будет думать, что мы хотим присвоить аргументу значение его самого. Второй, более распространённый случай использования **this** – это когда нам надо передать объект в какой-нибудь метод в виде аргумента (мы встретимся с таким случаем позже).

Пора раскрасить льва

Мы живем в разноцветном мире, и львы тоже имеют свой цвет. Поэтому давайте добавим новый атрибут для класса Lion, назовем его **color**, и он будет типа String. Во-первых, добавим атрибут в класс Lion:

```
public String color;
```

во-вторых, расширим конструктор добавлением третьего аргумента String color и изменим тело конструктора добавлением присваивания значения аргумента **color** атрибуту класса **color**:

```
public Lion (float height, float weight, String color) {  
    this. height = height;  
    this. weight = weight;  
    this.color = color;  
}
```

вот так теперь будет выглядеть наш конструктор. И конечно мы помним – раз поменялся конструктор, надо изменить снова в классе Main создание объекта lion:

```
Lion lion = new Lion (0.4f, 50, «yellow»);
```

Теперь у нас будет желтый лев. Теперь мы можем создавать львов с разным окрасом. Но возможна ситуация, когда тот кто будет пользоваться нашим классом Lion, для создания своего зоопарка сделает ошибку и вместо yellow введет yello или ellow и получится не совсем правильный лев. Вот как раз для случаев чтобы застраховаться от таких ошибок есть два способа.

Константы

Первый способ – это объявление констант, т.е. таких значений, которыми можно пользоваться, но нельзя изменить. Создадим несколько констант для раскраски льва в классе Lion:

```
public static final String RED = «red»;  
public static final String YELLOW = «yellow»;
```

```
public static final String WHITE = «white»;
public static final String BLACK = «black»;
```

Здесь стоит обратить внимание на два новых ключевых слова. Первое – static, второе – final.

Используя слово **static**, программист говорит JVM, что он хочет, чтобы объявляемая переменная принадлежала не создаваемому объекту, а классу. Класс – это тоже в своем роде объект, и он тоже может иметь атрибуты и методы. И вот как раз для того чтобы различать атрибуты и методы экземпляра класса, от атрибутов и методов класса используют слово **static**.

Второе слово **final** нужно для того, чтобы сказать JVM, что больше нельзя менять содержимое атрибута – и именно это делает такой атрибут постоянным или *константой*.

Далее после типа String идет название, все буквы которого большие – это просто так принято среди программистов что если подразумевается, что атрибут – это константа, то задавать ее имя большими буквами. JVM же в свою очередь абсолютно все равно какими буквами называется атрибут.

Теперь если мы перейдем в класс Main и захотим в конструкторе задать цвет льву, IntelliJ IDEA нам уже даст на выбор список констант, которые можно использовать. И это определенно помогает нам избежать ошибки в наборе, хотя мы все также имеем возможность ввести какой-то свой цвет.



Я выбрал RED:

```
Lion lion = new Lion (0.4f, 50, Lion.RED);
```

Иногда мы можем не давать такой возможности пользователю нашего класса – вводить свои значения, а хотим только позволить выбрать из предлагаемого списка. Для такого случая в Java существуют enumerations или перечисления.

Enumerations (перечисления)

Перечисление – это тоже своего рода класс со статическими константами и даже с возможностью иметь методы, просто выглядит немного упрощенно. Создадим перечисление для цветов в классе Lion. Для этого в первую очередь удалим все статические константы с цветами (чтобы не было путаницы). Вместо них напишем следующее:

```
public enum Colors
{
    RED,
    YELLOW,
    GREEN,
    WHITE,
    BLACK;
}
```

как видно из кода это просто список названий констант, и мы даже не знаем, что они хранят внутри. Далее изменим тип атрибута **color String** на **Colors**:

```
public Colors color;
```

изменим также тип аргумента **color** в конструкторе:

```
public Lion (float height, float weight, Colors color)
```

и теперь попробуйте в классе Main задать цвет льву, IntelliJ IDEA предложит список:



На этот раз я выбрал белый цвет:

```
Lion lion = new Lion (0.4f, 50, Lion.Colors. WHITE);
```

теперь если вывести в консоль цвет льва, я добавил строчку кода для этого в классе Main:

```
System.out.println(lion.color);
```

мы получим:



Задания

На этот раз задания будут больше похожи на эксперименты для самостоятельного изучения. Не волнуйтесь если у вас что-то не получиться, все будет рассмотрено дальше. Обязательно скопируйте проект с зоопарком, чтобы не сломать его экспериментами.

– Попробуйте добавить какой-нибудь новый атрибут и метод нашему классу Lion. Например, пусть у него будет имя и он будет выполнять какую-нибудь команду – например «идти» или «прыгать».

– Добавьте в класс Lion в метод eat проверку на переедание: если масса льва достигает 90 килограмм, то вместо кормления выводится сообщение «I'm not hungry» («я не голоден»).

– Попробуйте создать несколько десятков львов и потом их всех покормить. Для этого вы можете использовать массив типа Lion и цикл, с помощью которого проходить по каждому элементу массива и кормить льва.

– Изучите метод random класса Math. Попробуйте использовать его для кормления львов, давая случайное количество еды в диапазоне от 3 до 10 килограмм.

– Попробуйте создать класс другого животного собаки, кошки, утки, кого хотите. И наделить этот класс своими атрибутами и методами.

Глава 11. Инкапсуляция – защищаем наших животных

В реальном зоопарке есть проблема: посетители могут сколько угодно кормить животных и чем попало, что конечно же вредно для животных. В нашем виртуальном Зоопарке проблем гораздо больше: любой пользователь класса Lion может в любой момент менять любое свойство созданных объектов-львов. Причем он может задавать любые значения, в том числе и те, которые могут не иметь никакого смысла – например, ничто не помешает ему задать отрицательный вес льву или накормить его тонной еды.

И вот как раз для избегания такого рода небезопасных проблем при построении классов и нужно учитывать один из принципов ООП, а именно инкапсуляцию – т.е. давать доступ к тому, что действительно разрешено менять и контролировать в каких пределах это можно менять.

Защищаем льва

Первое, что надо сделать для защиты от «варваров» – сделать все атрибуты класса Lion приватными, – модификатор доступа private. Вот как это будет выглядеть:

```
Main.java Lion.java
1  public class Lion {
2      public enum Colors
3      {
4          RED,
5          YELLOW,
6          GREEN,
7          WHITE,
8          BLACK;
9      }
10     private float height;
11     private float weight;
12     private Colors color;
13
14     public Lion(float height, float weight, Colors color) {
15         this.height = height;
16         this.weight = weight;
17         this.color = color;
18     }
19
20     public void growl() { System.out.println("G-R-R-RIII"); }
21
22     public void eat(float food) { weight += food; }
23 }
```

Но тогда появляется другая проблема, а что делать если пользователь класса захочет узнать вес, рост, цвет льва? И вот для этого случая, обеспечения контролируемого доступа к атрибутам объекта существует так называемые геттеры и сеттеры (getters и setters).

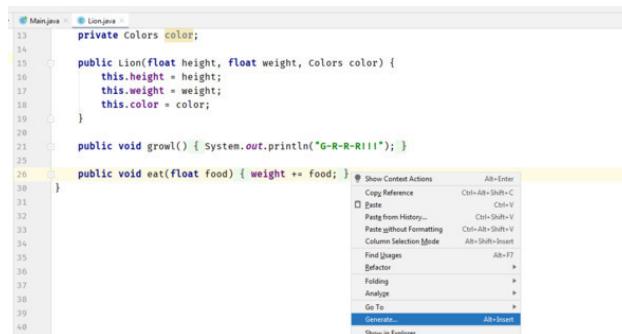
Геттеры и сеттеры (getters и setters)

Геттер – от английского слова get (получать), сеттер – от английского слова set (устанавливать). К сожалению геттеры и сеттеры в Java – это просто по-особому названные обычные методы. В других языках геттеры и сеттеры выглядят и используются в более лаконичной форме. Но как-бы то ни было давайте подумаем, что мы откроем с помощью них пользователю класса.

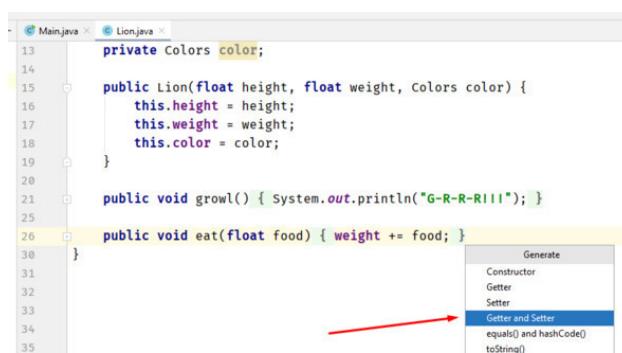
height – нужно ли разрешать снаружи менять рост льва? – я так не думаю, он сам растет. Тоже самое и с его весом – *weight*. Поэтому для них нужно сделать только геттеры.

color – здесь мне кажется вопрос воображения. Представим, что раз у нас зоопарк, туда приходят дети и иногда, по особым праздникам львов красят безопасной и быстро смывающейся краской. Поэтому для *color* сделаем геттер и сеттер.

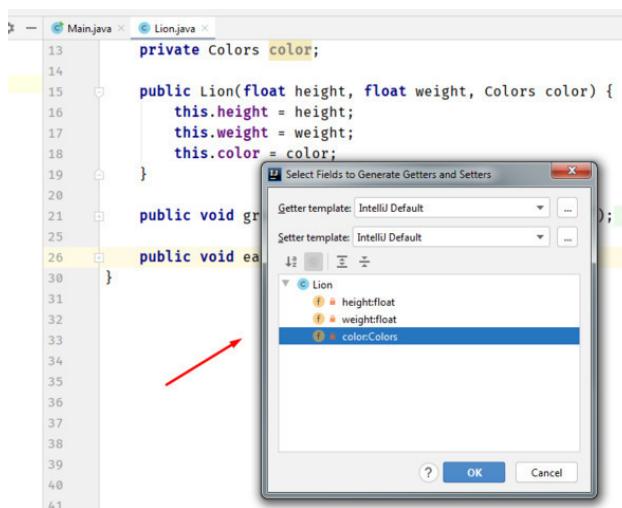
Вот как раз и начнем с *color*, воспользуемся умением IntelliJ IDEA делать это автоматически. Для этого кликнем правой кнопкой мыши после самого последнего метода и выберем Generate:



потом Getter and setter в новом окошке:



и потом color в еще одном окошке и OK:



Если вы все сделали правильно, то в конец класса будут добавлены два метода:

```
1 public class Lion {
2     public enum Colors
3     {
4         RED,
5         YELLOW,
6         GREEN,
7         WHITE,
8         BLACK;
9     }
10
11    private float height;
12    private float weight;
13    private Colors color;
14
15    public Lion(float height, float weight, Colors color) {
16        this.height = height;
17        this.weight = weight;
18        this.color = color;
19    }
20
21    public void growl() { System.out.println("G-R-R-R!!!"); }
22
23    public void eat(float food) { weight += food; }
24
25    public Colors getColor() {
26        return color;
27    }
28
29    public void setColor(Colors color) {
30        this.color = color;
31    }
32
33}
34
35}
36
37}
38
39}
```

Рассмотри их подробнее, первый – геттер:

```
public Colors getColor () {
    return color;
}
```

публичный метод, который возвращает значение типа **Colors**, внутри метода оператор **return** который непосредственно и возвращает значение атрибута **colors**.

Теперь сеттер:

```
public void setColor (Colors color) {
    this.color = color;
}
```

тоже публичный метод, который ничего не возвращает, который принимает аргумент типа **Colors** и присваивает его атрибуту **colors**.

Таким образом мы можем сказать, что это методы-обертки над атрибутами класса, главный смысл которых: обеспечивать доступ снаружи и контролировать присвоение значений.

Все что нам осталось – сделать два геттера для *height* и *weight*. Кто хочет пишет руками, кто хочет использует IntelliJ IDEA. Вообще рекомендуется при изучении все писать руками, а только уже по прошествии времени использовать плюшки IDE, в которой работаете. Итак, в итоге должны появиться еще два метода:

```
16     this.height = height;
17     this.weight = weight;
18     this.color = color;
19 }
20
21 public void growl() { System.out.println("G-R-R-R!!!"); }
22
23 public void eat(float food) { weight += food; }
24
25 public Colors getColor() {
26     return color;
27 }
28
29 public void setColor(Colors color) {
30     this.color = color;
31 }
32
33 public float getHeight() {
34     return height;
35 }
36
37 public float getWeight() {
38     return weight;
39 }
40
41 }
```

Теперь над нашими львами нельзя глумиться всем, кому не лень. Насчет цвета не волнуйтесь, по окончании проекта мы также не позволим перекрашивать льва кому угодно, только специально обученному персоналу. Все что остается – это исправить код в классе Main, переделать две строчки кода, которые выводят вес и цвет льва. Вот финальный результат:

```
1 public class Main {
2
3     public static void main(String[] args) {
4         System.out.println("Добро пожаловать в наш зоопарк!");
5
6         Lion lion = new Lion( height: 0.4f, weight: 50, Lion.Colors.WHITE );
7         lion.growl();
8         lion.eat( food: 10 );
9         System.out.println(lion.getWeight());
10        System.out.println(lion.getColor());
11    }
12
13 }
```

Если вы запустите программу – она должна выполниться без ошибок.

Глава 12. Расширяем Зоопарк

Если в нашем Зоопарке будут только львы, то к нам никто не будет ходить и зоопарк обанкротится. Поэтому единственный выход – завести больше животных! – красивых и интересных этим сейчас и займемся.

Я люблю жирафов – они большие и добрые!

Как вы поняли, первый кандидат на поселение – это Жираф. Создадим новый класс Giraffe – правая кнопка мыши на src в панели Project -> New —> Java class... даем имя классу Giraffe.

Кстати, важное замечание, имя класса которое вы даете должно совпадать с именем файла, в котором код этого класса находится. Т.е. для класса Giraffe IntelliJ IDEA создаст файл Giraffe.java и вам категорически не стоит переименовывать этот файл руками. Если вы захотите переименовать сам класс – делайте это внутри IntelliJ IDEA она сама все правильно переименует.

Теперь скопируем все атрибуты и методы из класса Lion в Giraffe – copy-paste, так сказать. Только не надо копировать перечисление Colors. И еще исправьте конструктор класса с Lion на Giraffe. Опять-таки, кто хочет пишет руками – это полезнее и тренирует память и осознанность. В итоге должно получиться:

```
public class Giraffe {
    private float height;
    private float weight;
    private Lion.Colors color;

    public Giraffe(float height, float weight, Lion.Colors color) {
        this.height = height;
        this.weight = weight;
        this.color = color;
    }

    public void growl() { System.out.println("G-R-R-R!!"); }

    public void eat(float food) { weight += food; }

    public Lion.Colors getColor() {
        return color;
    }

    public void setColor(Lion.Colors color) {
        this.color = color;
    }

    public float getHeight() {
        return height;
    }

    public float getWeight() {
        return weight;
    }
}
```

Внимательный читатель заметит одно недоразумение – жирафы не рычат! – они, как подсказывает Yandex, могут свистеть, мычать, реветь и издавать звуки, напоминающие флейту. В нашем Зоопарке они будут красиво «флейтировать» – соответственно метод growl мы заменим на fluting:

```
public void fluting()
{
    System.out.println («pheew pheew»);
}
```

Лебеди – красивые и летают

Как уже всем понятно создаем новый класс Swan (лебедь). Потом копипастим атрибуты и методы из класса Lion. Не забываем переименовать конструктор класса на Swan. Лебеди не рычат, они шипят, поэтому надо заменить метод growl на hiss. И нововведение: надо добавить новый метод fly, потому что лебеди летают, сам метод пока оставим пустым. Вот что получится в итоге:

```
public class Swan {
    private float height;
    private float weight;
    private Lion.Colors color;

    public Swan(float height, float weight, Lion.Colors color) {
        this.height = height;
        this.weight = weight;
        this.color = color;
    }

    public void hiss() { System.out.println("ssssssssssll!"); }

    public void eat(float food) { weight += food; }

    public void fly() {}

    public Lion.Colors getColor() {
        return color;
    }

    public void setColor(Lion.Colors color) {
        this.color = color;
    }

    public float getHeight() {
        return height;
    }

    public float getWeight() {
        return weight;
    }
}
```

Ну и теперь тестов ради полностью изменим метод main в классе Main. Сначала сотрем там все что было про создание и вывод информации о льве. Потом создадим одного льва, жирафа и лебедя, ну и заставим их произнести звуки, которые они обычно произносят.

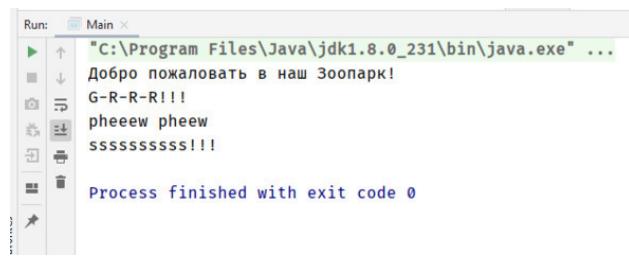
Вот так в итоге будет выглядеть класс Main:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("добра пожаловать в наш Зоопарк!");

        Lion lion = new Lion( height: 0.5f, weight: 55, Lion.Colors.YELLOW);
        Giraffe giraffe = new Giraffe( height: 4.7f, weight: 120, Lion.Colors.YELLOW);
        Swan swan = new Swan( height: 1.2f, weight: 21, Lion.Colors.WHITE);

        lion.growl();
        giraffe.fluting();
        swan.hiss();
    }
}
```

И вот что выведется в консоль при запуске:



Время обеда – пора кормить животных.

Теперь давайте расширим наш Зоопарк тем, что в нем поселим по 5 особей каждого вида: 5 львов, 5 жирафов, 5 лебедей. А потом нам надо будет их всех накормить. Для всего этого воспользуемся массивами и циклами. Опять очистим метод main класса Main, оставив только приветствие. И в первую очередь наплодим львов. Для дальнейшего использования нам надо где-то хранить созданных львов и для этого подойдет обычный массив с типом Lion, вот так мы его объявим:

```
Lion [] lions = new Lion [5];
```

теперь создадим 5 львов в цикле:

```
for (int i = 0; i < 5; i++)
{
    lions [i] = new Lion (0.6f,55, Lion.Colors. YELLOW);
```

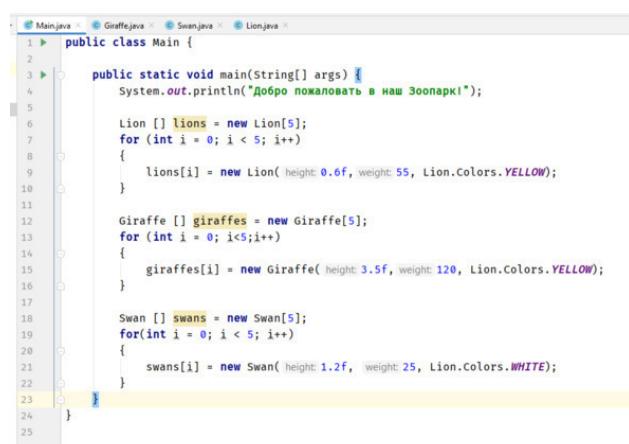
Тоже самое проделаем для жирафов и пингвинов:

```
Giraffe [] giraffes = new Giraffe [5];
for (int i = 0; i < 5; i++)
{
    giraffes [i] = new Giraffe (3.5f,120, Lion.Colors. YELLOW);
```

```
Swan [] swans = new Swan [5];
```

```
for (int i = 0; i < 5; i++)
{
    swans [i] = new Swan (1.2f, 25, Lion.Colors. WHITE);
```

В итоге метод main будет выглядеть так:



А теперь осталось дело за малым: их всех накормить:

```
for (int i = 0; i < 5; i++)
    lions [i].eat (10);
for (int i = 0; i < 5; i++)
    giraffes [i].eat (10);
for (int i = 0; i < 5; i++)
    swans [i].eat (10);
```

При запуске программа выдаст только приветствие, и это понятно наши животные «спасибо» за кормление не говорят.

Я думаю, что большинство уже увидело глобальную проблему, а если кто не увидел – задайте себе вопрос «Почему я устал\устала писать столько кода?» Ответ будет в следующей главе, не спешите перелистывать страницу – подумайте....

Глава 13. Улучшаем менеджмент в Зоопарке – абстракция и наследование

Что же за глобальная проблема в организации кода, который мы написали в прошлой главе? У нас всего только три вида животных, а нам уже приходиться заводить под каждый вид отдельный массив, отдельно создавать, отдельно кормить. А как мы знаем в реальном зоопарке могут быть даже не десятки, а сотни видов животных, только представьте сколько кода придется писать, чтобы накормить 100 видов животных!

Но именно для решения таких случаев мы используем еще два принципа ООП – абстракцию и наследование.

Абстракция и наследование

Абстракция и наследование очень тесно связаны друг с другом. Потому что абстракция – это теоретическая часть, а наследование – практическая часть одной идеи: идеи объединения общих атрибутов и методов и вынесение их в отдельный класс, который будет *предком* для других классов.

Для наших животных мы вынесем практически все атрибуты и методы в *родительский* класс Animal. В каждом классе есть уникальные методы, которые не могут быть вынесены в родительский класс без изменений: это методы как животное воспроизводит звуки и метод **fly** у Swan. В этом случае мы применим силу абстрагирования (!) и в родительском методе определим методы **makeSound** и **move** – т.е. более общие методы, которые имеют тот же смысл.

Создадим новый класс Animal, и в него скопиастим атрибуты и методы из класса Lion. Поправим конструктор класса: переименуем Lion в Animal и уберем Lion в аргументах конструктора:

```
public Animal (float height, float weight, Colors color)
```

Теперь обратите внимание, у нас есть такая строка:

```
private Lion.Colors color;
```

надо ее **исправить на**:

```
private Colors color;
```

т.к. теперь перечисление Colors будет присутствовать в классе Animal. Также надо исправить геттер getColor на:

```
public Colors getColor ()
```

и сеттер setColor исправить на:

```
public void setColor (Colors color)
```

Как вы видите мы таким образом везде избавляемся от упоминания о перечислении Colors, объявленном в классе Lion.

Теперь уберем метод **growl** и добавим два новых метода:

```
public void makeSound () {  
    System.out.println ("...");  
}  
public void move () {  
}
```

Метод **makeSound** просто выводит три точки, потому что он будет *переопределён* в дочерних классах (наследниках). Метод **move** вообще пустой, мы позже с ним разберемся. В итоге вот какой класс Animal у нас получится:

```
1 public class Animal {
2     public enum Colors
3     {
4         RED,
5         YELLOW,
6         GREEN,
7         WHITE,
8         BLACK;
9     }
10    private float height;
11    private float weight;
12    private Colors color;
13
14    public Animal(float height, float weight, Colors color)
15    {
16        this.height = height;
17        this.weight = weight;
18        this.color = color;
19    }
20    public void makeSound()
21    {
22        System.out.println("....");
23    }
24    public void move()
25    {
26    }
27    public void eat(float food)
28    {
29        weight += food;
30    }
31    public Colors getColor()
32    {
33        return color;
34    }
35    public void setColor(Colors color)
36    {
37        this.color = color;
38    }
39    public float getHeight()
40    {
41        return height;
42    }
43    public float getWeight()
44    {
45        return weight;
46    }
47}
```

И вот теперь мы готовы к тому, чтобы применить наследование! Начнем с класса Lion. Чтобы указать что класс Lion является наследником класса Animal надо изменить объявление класса Lion:

```
public class Lion {
```

заменить на:

```
public class Lion extends Animal {
```

здесь используется ключевое слово **extends**, которое указывает от кого *наследуется* класс.

Мы видим, что у класса Animal и у класса Lion повторяются атрибуты: height, weight и color. А наследование как раз и нужно для того, чтобы избегать подобных вещей. Но в классе Animal эти атрибуты объявлены как приватные, т.е. закрыты для доступа снаружи и закрыты для наследования – это второй смысл модификатора доступа private. Как же их открыть для наследования? Если мы заменим private на public, то мы сделаем их не только доступными для наследования, но и доступными для доступа снаружи кому угодно! Как раз ля этого есть третий модификатор доступа – **protected**, – он делает доступным атрибут\метод для наследников, но закрытым для использования для всех остальных. Итак, меняем в Animal везде private на protected. В классе Lion удаляем все атрибуты: height, weight и color.

Т.к. в классе Animal и в классе Lion повторяются геттеры и сеттеры (мы же скопировали), удаляем их всех в классе Lion – они также наследуются из класса Animal теперь.

Метод **eat** в классе **Lion** также можно удалить. Остается метод **growl** – все что нужно, это переименовать его в **makeSound**. Почему это нужно? – в методе **Animal** выводится в консоль в три точки, а наш Лев «звучит» по-другому и громко – таким образом мы **переопределяем** метод. И это обычный способ, когда нам надо поменять поведение метода, который был определен в родительском классе.

В конструкторе класса **Lion** также надо сделать изменение. Он делает тоже, что и конструктор класса **Animal**: присваивает аргументы атрибутам. Его необходимо переписать вот так:

```
public Lion (float height, float weight, Colors color) {
    super (height, weight, color);
}
```

тут мы видим еще одно новое слово **super** – это обращение к конструктору родительского класса, и как мы видим, туда передаются аргументы. После всех изменений, класс **Lion** будет выглядеть так:

```
public class Lion extends Animal{
    public Lion(float height, float weight, Colors color) {
        super(height, weight, color);
    }
    public void makeSound() {
        System.out.println("G-R-R-R!!!");
    }
}
```

обратите внимание рядом с объявлением метода **makeSound** IntelliJ IDEA показывает синий кружок и красную стрелку вверх – таким образом она подсказывает что родительский метод переопределяется. Если вы кликнете на него то попадете в этот родительский метод.

Проделаем тоже самое с классами **Swan** и **Giraffe**:

- **extends Animal**
- удаляем атрибуты, геттеры, сеттеры и метод **eat**
- переименовываем методы **hiss** и **fluting** в **makeSound**
- удаляем метод **fly** у **Swan**. Т.к. есть метод **move** в **Animal** и если что, его можно переопределить
- меняем тело конструктора на использование **super**.

Вот что мы получим:

```
public class Giraffe extends Animal{
    public Giraffe(float height, float weight, Colors color) {
        super(height, weight, color);
    }
    public void makeSound() { System.out.println("pheew pheew"); }
}
```

```
public class Swan extends Animal{
    public Swan(float height, float weight, Colors color) { super(height, weight, color); }
    public void makeSound() { System.out.println("ssssssssss!!"); }
}
```

Чтобы быть уверенными, что нигде вы не допустили ошибку, вы можете запустить программу – программа должна показать приветствие и закончить свой работу.

Время ужина

Теперь мы будем избавляться от избыточного кода в методе main класса Main. Давайте опять применим нашу суперспособность – Абстрагирование! Что мы видим? мы видим, что мы создаем животных и потом их кормим, но в связи с тем, что они все были разные, приходилось писать для каждого вида отдельный код. Теперь перепишем код следующим образом:

- создадим массив типа Animal, который будет содержать 15 элементов
- напишем цикл, в котором сразу будем создавать Льва, Жирафа и Лебедя каждую итерацию (в каждом шаге цикла)
- и наконец-то магия – можно написать цикл, который вызывает метод eat независимо от вида животного!

```
1 > Main.java < Giraffe.java < Swan.java < Lion.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Добро пожаловать в наш Зоопарк!");

        Animal[] animals = new Animal[15];
        int counter = 0;
        for (int i = 0; i < 5; i++) {
            animals[counter] = new Lion( height: 0.6f, weight: 55, Lion.Colors.YELLOW);
            counter++;
            animals[counter] = new Giraffe( height: 3.5f, weight: 120, Lion.Colors.YELLOW);
            counter++;
            animals[counter] = new Swan( height: 1.2f, weight: 25, Lion.Colors.WHITE);
            counter++;
        }

        // feeding animals
        for(int i = 0; i<15;i++)
            animals[i].eat(food: 10);
    }
}
```

Есть несколько замечаний по коду. Первое и самое важное: почему массив типа Animal, а мы в него можем клать объекты другого типа? Ответ: потому что это объекты-наследники класса Animal и таким образом они тоже как-бы типа Animal. Почему в цикле где мы кормим животных мы просто можем вызвать метод eat и не думать, как раньше у кого мы его вызываем? Ответ: тоже потому, что эти объекты – наследники Animal, – у которого определен этот метод eat и соответственно любой наследник может к нему обратиться.

Второе замечание: зачем нужен этот счетчик counter в цикле где мы создаем животных? Тут все очень примитивно – примитивный массив в Java обделен методом, который может просто «заталкивать» значения в массив. В других языках это обычно метод push. Но можно не волноваться: скоро мы будем использовать продвинутый вариант массива.

И третье замечание. Оно не сразу бросается в глаза. Теперь, когда мы кормим животных, они получают одинаковую порцию еды. Вам не кажется, что лебедя может разорвать от десяти килограмм? Защитники животных могут быть спокойны – Java-программисты могут решить эту проблему безболезненно, и никто не пострадает и не останется голодным!

Глава 14. Заботимся о каждом – полиморфизм

В это главе мы научимся использовать четвертый принцип ООП – полиморфизм. Это поможет нам более внимательно отнестись к нашим животным в зоопарке, учесть особенность строения их организма. Но прежде чем мы ими займемся, мы познакомимся с продвинутым вариантом массива.

Объектно-ориентированный массив: ArrayList

Обычный массив в Java (array) имеет существенные ограничения: при объявлении мы можем задать его фиксированную длину, мы не можем его расширить при необходимости, у него отсутствуют много методов для работы с элементами. Стоит заметить, что Java, один из немногих языков программирования, который имеет расширенную иерархию коллекций – классов которые работают с наборами данных и обладают специальными методами, упрощающими жизнь разработчика, например, автоматической сортировкой или формированием массива уникальных элементов. Мы их рассмотрим в третьей части книги.

А пока для наших целей будет достаточно узнать, что отличной заменой обычному массиву в Java является класс ArrayList. Он не имеет фиксированной длины и обладает полезными методами для работы: **add** – позволяет добавлять элементы в конец списка (правильнее говорить списка, потому что list с английского значит *список*), а также добавлять элемент в нужную позицию, сдвигая при этом остальные (!); **addAll** – работает также как **add** только уже добавляет несколько элементов; **contains** – позволяет проверить присутствие элемента в списке; **get** – получить элемент по индексу; **indexOf** – узнать индекс элемента в списке; **remove** – удалить элемент в списке по индексу; **set** – заменить\установить элемент в списке по индексу; **toArray** – вернуть обычный массив. Сами видите сколько полезного здесь есть (и сколько отсутствует у обычного массива)

Теперь перепишем метод main в классе Main. В итоге:

```

Main.java  Animal.java  Giraffe.java  Swan.java  Lion.java
1 import java.util.ArrayList;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Добро пожаловать в наш Зоопарк!");
7
8         ArrayList<Animal> animals = new ArrayList();
9
10        for (int i = 0; i < 5; i++) {
11            animals.add(new Lion( height: 0.6f, weight: 55, Animal.Colors.YELLOW));
12            animals.add(new Giraffe( height: 3.5f, weight: 120, Animal.Colors.YELLOW));
13            animals.add(new Swan( height: 1.2f, weight: 25, Animal.Colors.WHITE));
14        }
15
16        for (Animal a:animals)
17            a.eat( food: 10);
18    }
19
20 }

```

Теперь разберем построчно.

`ArrayList <Animal> animals = new ArrayList();`

это объявление ArrayList который будет содержать элементы типа Animal. И выглядит как просто создание обычного объекта класса ArrayList. Следующее: в теле цикла используется метод **add**, чтобы добавить созданное животное.

И самое интересное: ArrayList это не просто массив – это *коллекция* элементов. В Java любая коллекция позволяет *перебирать* элементы, которые она содержит (такой перебор элементов еще называют *итерированием*). В нашем коде итерирование выглядит вот так:

```
for (Animal a: animals)
```

«расшифровка» звучит как: исполняй цикл, пока коллекция не закончилась, а очередной элемент присвой в переменную **a**. Это удобно, потому что не надо писать дополнительно ни проверку на длину коллекции, ни счетчик, который надо инкрементировать.

Как мы видим, код стал более *читаем*. И вообще зачастую правильно используемый класс коллекций способен сильно упростить разработку функционала программы, но для этого надо знать особенности из использования.

Животные опять накормлены, но «проблема переедания» все также осталась. И сейчас мы будем ее решать.

На раз-два отчитайся

В первую очередь, пора уже прекратить безотчетно всех кормить – сейчас, после создания животного, мы не получаем никакой информации, что с ним происходит. Давайте попробуем выводить информацию о виде, которому он принадлежит и текущем весе. Мы будем выводить всю эту информацию после того как животное было накормлено.

Первое что приходит в голову это распечатать объект с помощью **System.out.println**. Закомментируем весь код в методе **main**. И напишем следующие строки:

```
Animal some = new Lion (0.4f, 55, Animal.Colors.GREEN);
System.out.println (some);
```

Запустим программу и результат будет выглядеть несколько странным:

```
<C:\Program Files\Java\jdk1.8.0_231\bin\java. exe>...
Lion@1b6d3586
Process finished with exit code 0
```

Но все верно, Java вывела класса от которого был создан объект (**Lion**) и указала ячейку памяти, на которую ссылается переменная **some**. Правда такая информация нам практически бесполезна. Нам нужно немного погрузится в теорию, чтобы сделать как нам хочется.

Дело в том, что кода мы пытаемся распечатать объект JVM «за кадром» вызывает у объекта метод **toString ()**. Вы можете спросить: но как же так, ни в одном из наших классов не определено такого метода?! Правильно, в мире Java, абсолютно любой класс первым классом-предком (такой себе супер-предок) имеет класс **Object**. Это тоже скрыто «за кадром», но как раз **Object** обладает методом **toString ()**, который потом переходит «по наследству» всем остальным классам и объектам. Больше о нем вы можете прочитать здесь <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>. Хорошая новость состоит в том, что мы можем переопределить метод **toString ()**, чтобы он выводил желаемую нам информацию.

Откроем класс **Animal** и в самом конце напишем метод:

```
public String toString ()
{
    return <I am " + this.getClass().getName () +> (height:" + height +", weight:
" + weight +", color:" + color +")>;
}
```

Главное – не запутаться в кавычках. Все что делает этот метод: возвращает строку. Мы «собираем» эту строку из чередуя текст и информацию. **this.getClass().getName ()** – так мы получаем имя класса, дальше мы просто добавляем свойства. После запуска программы результат будет гораздо более информативен:

```
<C:\Program Files\Java\jdk1.8.0_231\bin\java. exe>...
```

```
I am Lion (height:0.4, weight: 55.0, color: GREEN)
Process finished with exit code 0
```

Теперь можно вернуться к аккуратному кормлению животных.
С каждого по способностям, каждому по потребностям

Для того чтобы контролировать количество пищи, которое потребляет животное в первую очередь мы сделаем метод eat в классе Animal защищенным – protected, – т.е. доступным только для наследников класса, чтобы никто не смог дать больше пищи. А потом создадим новый метод в классе Animal: feed (кормить) – он не будет принимать никаких аргументов:

```
public void feed () {}
```

и как вы видите, он не содержит никакого функционала внутри. Зачем же он нужен, если он ничего не делает?! А он нужен для того чтобы его переопределили в классах-наследниках, что мы и сделаем.

В классе Lion добавим:

```
public void feed ()
{
    eat (10);
}
```

В классе Giraffe добавим:

```
public void feed ()
{
    eat (3);
}
```

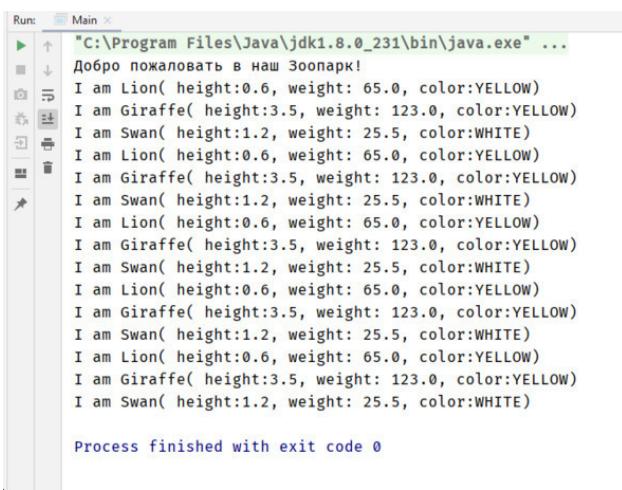
В классе Swan добавим:

```
public void feed ()
{
    eat (0.5);
}
```

Уберем тестовый код в методе main и поменяем цикл:

```
1 import java.util.ArrayList;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         System.out.println("Добро пожаловать в наш Зоопарк!");
8
9         ArrayList<Animal> animals = new ArrayList();
10
11        for (int i = 0; i < 5; i++) {
12            animals.add(new Lion( height: 0.6f, weight: 55, Animal.Colors.YELLOW));
13            animals.add(new Giraffe( height: 3.5f, weight: 120, Animal.Colors.YELLOW));
14            animals.add(new Swan( height: 1.2f, weight: 25, Animal.Colors.WHITE));
15        }
16
17        for (Animal a:animals) {
18            a.feed();
19            System.out.println(a);
20        }
21    }
22
23 }
```

После запуска программа выведет:



The screenshot shows a Java application window titled "Main". The console output displays a series of messages from three objects: Lion, Giraffe, and Swan. Each message includes the object's name, its height, weight, and color. The output is as follows:

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
Добро пожаловать в наш Зоопарк!
I am Lion( height:0.6, weight: 65.0, color:YELLOW)
I am Giraffe( height:3.5, weight: 123.0, color:YELLOW)
I am Swan( height:1.2, weight: 25.5, color:WHITE)
I am Lion( height:0.6, weight: 65.0, color:YELLOW)
I am Giraffe( height:3.5, weight: 123.0, color:YELLOW)
I am Swan( height:1.2, weight: 25.5, color:WHITE)
I am Lion( height:0.6, weight: 65.0, color:YELLOW)
I am Giraffe( height:3.5, weight: 123.0, color:YELLOW)
I am Swan( height:1.2, weight: 25.5, color:WHITE)
I am Lion( height:0.6, weight: 65.0, color:YELLOW)
I am Giraffe( height:3.5, weight: 123.0, color:YELLOW)
I am Swan( height:1.2, weight: 25.5, color:WHITE)
I am Lion( height:0.6, weight: 65.0, color:YELLOW)
I am Giraffe( height:3.5, weight: 123.0, color:YELLOW)
I am Swan( height:1.2, weight: 25.5, color:WHITE)

Process finished with exit code 0
```

А вот он наш полиморфизм

Вот наконец-то мы дошли до иллюстрации четвертого (и последнего) принципа ООП. Его объяснение звучит запутано, зато на примере выглядит очень просто.

Раньше мы вызывали метод **eat**, который был определен в базовом классе Animal, и из-за того, что все животные – наследники Animal, то метод **eat** был доступен для них. Теперь ситуация прямо противоположная, мы перебираем список объектов с типом Animal, и вызываем метод **feed**. По идеи, должен был бы вызываться метод **feed** класса Animal, но Java поддерживает полиморфизм и творит чудо: она понимает, что раз это наследник класса Animal и у него есть свой собственный метод **feed**, то вызывать необходимо именно его. Полиморфизм – это вызов методов, соответствующих контексту, если бы метод отсутствовал в наследнике, вот тогда Java вызвала бы метод в базовом классе Animal.

This is the end?

Ну вот и подошло к концу наше совместное погружение в основы программирования с помощью Java. Но любое окончание – это одновременно и начало чего-то нового. И нового у вас у будет предостаточно, если вы решите продолжить ваше путешествие в мир программирования.

«А много еще учить?»

Как я раньше говорил – эта книга всего лишь краткое введение в программирование и в язык программирования Java. Поэтому еще очень много тем ждут вашего внимания:

- классы-оболочки (Integer, Float,...)
- абстрактные классы и интерфейсы
- анонимные классы
- работа со строками
- работа с датой
- работа с вводом-выводом (чтение-запись файлов)
- исключения и их обработка
- многопоточность
- работа с графическим интерфейсом
- фреймворк Spring

Темы большие, но существует огромное количество книг, плейлистов на YouTube по каждой теме.

«Как мотивировать себя на изучение?»

На самом деле никак. Мотивация сейчас рассматривается как волшебная пиллюя, которую, во-первых, надо где-то взять, а во-вторых, как только нашел сразу использовать. На самом деле, это очередная тема на которой пытаются заработать всякие бизнес-тренеры.

Правда состоит в том, что вам или интересно программировать и вы это делаете, или вы только «пытаетесь себя заставить» – тогда может просто найти себе другое применение, а не вестись на тренды?

Но даже если у вас есть неподдельный интерес и желание программировать, попробуйте ввести дисциплину в процесс обучения. Пишите каждый день маленькие программы или решайте задачки с codewars, обязательно узнавайте что-то новое из теории. Если вы столкнулись с чем-то непонятным – гуглите!

Еще один вариант ввести дисциплину в ваше изучение – запишитесь на платные курсы. Сейчас есть варианты даже с оплатой после (!) того как вы устроились на работу. Хорошие платные курсы – это когда присутствует четкая программа обучения, видео-уроки, которые были записаны преподавателями именно этих курсов (а не из YouTube), проверка домашних заданий, менторинг, несколько проектов которые потом можно без стыда положить в портфолио.

«Будет ли продолжение?»

Для меня написание книги – это был абсолютно новый опыт, который мне понравился. Буду ли я писать «улучшенную книгу про Java»? – не уверен. А вот новая книга про программирование устройств на базе ОС Android – это идея которая мне нравится.

Поэтому благодарю всех кто потратил свое время и прочитал книгу, еще больше буду благодарен за замечания и предложения которые вы можете написать в ВК группу https://vk.com/pablo_shkodit в обсуждении «Книга Java 2021: Легкий старт». Также там вы можете задать вопросы программирование в целом, и программирование на Java в частности.

Весь исходный код вы можете скачать с <https://github.com/shaman4d/Java2021EasyStartBook>