

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ  
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»  
ФИЛИАЛ «МИНСКИЙ РАДИОТЕХНИЧЕСКИЙ КОЛЛЕДЖ»

**УТВЕРЖДАЮ**

Директор МРК

\_\_\_\_\_ С.Н. Анкуда

«\_\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

Регистрационный № \_\_\_\_\_

**РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ МОБИЛЬНЫХ УСТРОЙСТВ**

**МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ**

**ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ**

для учащихся специальности:

2-39 03 02 «Программируемые мобильные системы»

Минск 2017 г.

**СОСТАВИТЕЛИ:**

С.А. Апанасевич, преподаватель 1 категории дисциплин специального цикла

**РЕКОМЕНДОВАНЫ К УТВЕРЖДЕНИЮ:**

Цикловой комиссией «Программное обеспечение информационных технологий» филиала БГУИР «Минский радиотехнический колледж»

Протокол № \_\_\_\_\_ от \_\_\_\_\_

Заседанием педагогического совета филиала БГУИР «Минский радиотехнический колледж»

Протокол № \_\_\_\_\_ от \_\_\_\_\_

Методическая экспертиза \_\_\_\_\_  
подпись

\_\_\_\_\_  
ФИО

## Содержание:

Лабораторная работа № 1 .....	6
Тема работы: «Создание, компилирование, отладка и выполнение проектов в интегрированной среде разработки» .....	6
Лабораторная работа № 2 .....	41
Тема работы: «Создание линейных программ» .....	41
Лабораторная работа № 3 .....	57
Тема работы: «Разработка программ с использованием управляющих инструкций» .....	57
Лабораторная работа № 4 .....	85
Тема работы: «Разработка программ с использованием массивов» .....	85
Лабораторная работа № 5 .....	105
Тема работы: «Разработка классов и использование их в программах» ..	105
Лабораторная работа № 6 .....	122
Тема работы: «Разработка методов и использование их в программах» ..	122
Лабораторная работа № 7 .....	129
Тема работы: «Перегрузка методов и передача аргументов» .....	129
Лабораторная работа № 8 .....	135
Тема работы: «Разработка программ, реализующих механизм наследования» .....	135
Лабораторная работа № 9 .....	150
Тема работы: «Разработка программ с использованием наследования и переопределение методов» .....	150
Лабораторная работа № 10 .....	157
Тема работы: «Разработка программ, реализующих и использующих интерфейсы» .....	157
Лабораторная работа № 11 .....	169
Тема работы: «Разработка программ с использованием пакетов» .....	169
Лабораторная работа № 12 .....	175
Тема работы: «Разработка программ обработки символов» .....	175
Лабораторная работа № 13 .....	179
Тема работы: «Разработка программ разработки строк» .....	179
Возвращает Unicode-символ, который предшествует данному индексу. ....	182
Лабораторная работа № 14 .....	197
Тема работы: «Обработка исключительных ситуаций» .....	197

Лабораторная работа № 15.....	212
Тема работы: «Поточная модель».....	212
Лабораторная работа № 16.....	223
Тема работы: «Разработка многопоточных приложений» .....	223
Лабораторная работа № 17.....	234
Тема работы: «Разработка программ создания файлов».....	234
Лабораторная работа № 18.....	244
Тема работы: «Разработка программ обработки файлов».....	244

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Создание, компилирование, отладка и выполнение проектов в  
интегрированной среде разработки»

Минск  
2018

## **Лабораторная работа № 1**

### **Тема работы: «Создание, компилирование, отладка и выполнение проектов в интегрированной среде разработки»**

#### **1. Цель работы**

Научить создавать, и компилировать приложение в интегрированной среде разработки.

#### **2. Задание**

Необходимо вывести на экран сообщение по варианту.

Номер варианта соответствует вашему номеру по списку.

1. Двадцать седьмого февраля 1815 года дозорный Нотр-Дам де-ла-Гард дал знать о приближении трехмачтового корабля «Фараон», идущего из Смирны, Триеста и Неаполя.
2. Как всегда, портовый лоцман тотчас же отбыл из гавани, миновал замок Иф и пристал к кораблю между мысом Моржион и островом Рион.
3. Тотчас же, по обыкновению, площадка форта Св. Иоанна наполнилась любопытными, ибо в Марселе прибытие корабля всегда большое событие, особенно если этот корабль, как «Фараон», выстроен, оснащен, гружен на верфях древней Фокеи и принадлежит местному арматору.
4. Между тем корабль приближался; он благополучно прошел пролив, который вулканическое сотрясение некогда образовало между островами Каласарень и Жарос, обогнул Помег и приближался под тремя марсельями, кливером и контрбизанью, но так медленно и скорбно, что любопытные, невольно почуяв несчастье, спрашивали себя, что бы такое могло с ним случиться.
5. Однако знатоки дела видели ясно, что если что и случилось, то не с самим кораблем, ибо он шел, как полагается хорошо управляемому судну: якорь был готов к отдаче, ватербакштаги отданы, а рядом с лоцманом, который готовился ввести «Фараон» узким входом в марсельскую гавань, стоял молодой человек, проворный и зоркий, наблюдавший за каждым движением корабля и повторявший каждую команду лоцмана.
6. Безотчетная тревога, витавшая над толпою, с особой силой охватила одного из зрителей, так что он не стал дожидаться, пока корабль войдет в порт; он бросился в лодку и приказал грести навстречу «Фараону», с которым и поравнялся напротив бухты Резерв.
7. Это был юноша лет восемнадцати – двадцати, высокий, стройный, с красивыми черными глазами и черными, как смоль, волосами; весь

- его облик дышал тем спокойствием и решимостью, какие свойственны людям, с детства привыкшим бороться с опасностью.
8. На следующий день утро выдалось теплое и ясное. Солнце встало яркое и сверкающее, и его первые пурпурные лучи расцветили рубинами пенистые гребни волн.
  9. Пир был приготовлен во втором этаже того самого «Резерва», с беседкой которого мы уже знакомы. Это была большая зала, в шесть окон, и над каждым окном (бог весть почему) было начертано имя одного из крупнейших французских городов.
  10. Вдоль этих окон шла галерея, деревянная, как и все здание. Хотя обед назначен был только в полдень, однако уже с одиннадцати часов по галерее прогуливались нетерпеливые гости. То были моряки с «Фараона» и несколько солдат, приятелей Дантеса. Все они из уважения к жениху и невесте нарядились в парадное платье.
  11. Среди гостей пронесся слух, что свадебный пир почтят своим присутствием хозяева «Фараона», но это была такая честь для Дантеса, что никто не решался этому поверить. Однако Данглар, придя вместе с Кадруссом, в свою очередь, подтвердил это известие. Утром он сам видел г-на Морреля, и г-н Моррель сказал ему, что будет обедать в «Резерве».
  12. И в самом деле через несколько минут в залу вошел Моррель. Матросы приветствовали его дружными рукоплесканиями. Присутствие арматора служило для них подтверждением уже распространившегося слуха, что Дантес будет назначен капитаном. Они очень любили Дантеса и выражали благодарность своему хозяину за то, что хоть раз его выбор совпал с их желаниями.
  13. Едва г-н Моррель вошел, как, по единодушному требованию, Данглара и Кадрусса послали к жениху с поручением известить его о прибытии арматора, появление которого возбудило всеобщую радость, и сказать ему, чтобы он поторопился.
  14. Данглар и Кадрусс пустились бегом, но не пробежали и ста шагов, как встретили жениха и невесту. Четыре каталанки, подруги Мерседес, провожали невесту; Эдмон вел ее под руку. Рядом с невестой шел старик Дантес, а сзади Фернан. Злобная улыбка кривила его губы.
  15. Ни Мерседес, ни Эдмон не замечали этой улыбки. Они были так счастливы, что видели только себя и безоблачное небо, которое, казалось, благословляло их. Данглар и Кадрусс исполнили возложенное на них поручение; потом, крепко и дружески пожав руку Эдмону, заняли свои места – Данглар рядом с Фернаном, а Кадрусс рядом со стариком Дантесом, предметом всеобщего внимания.
  16. Старик надел свой шелковый кафтан с гранеными стальными пуговицами. Его худые, но мускулистые ноги красовались в

великолепных бумажных чулках с мушками, которые за версту отдавали английской контрабандой. На треугольной шляпе висел пук белых и голубых лент. Он опирался на витую палку, загнутую наверху, как античный посох. Словом, он ничем не отличался от щеголей 1796 года, прохаживавшихся во вновь открытых садах Люксембургского и Тюильрийского дворцов.

17. К нему, как мы уже сказали, присоединился Кадрусс, Кадрусс, которого надежда на хороший обед окончательно примирила с Дантесами, Кадрусс, у которого в уме осталось смутное воспоминание о том, что происходило накануне, как бывает, когда, проснувшись утром, сохраняешь в памяти тень сна, виденного ночью.
18. Данглар, подойдя к Фернану, пристально взглянул на обиженного поклонника. Фернан, шагая за будущими супругами, совершенно забытый Мерседес, которая в упоении юной любви, ничего не видела, кроме своего Эдмона, – то бледнел, то краснел. Время от времени он посматривал в сторону Марсея и при этом всякий раз невольно вздрагивал. Казалось, Фернан ожидал или по крайней мере предвидел какое-то важное событие.
19. Дантес был одет просто. Служа в торговом флоте, он носил форму, среднюю между военным мундиром и штатским платьем, и его открытое лицо, просветленное радостью, было очень красиво. Мерседес была хороша, как кипрская или хиосская гречанка, с черными глазами и коралловыми губками.
20. Она шла шагом вольным и свободным, как ходят арлезианки и андалузки. Городская девушка попыталась бы, может быть, скрыть свою радость под вуалью или по крайней мере под бархатом ресниц, но Мерседес улыбалась и смотрела на всех окружавших, и ее улыбка и взгляд говорили так же откровенно, как могли бы сказать уста: «Если вы друзья мне, то радуйтесь со мною, потому что я поистине очень счастлива!»
21. Когда жених, невеста и провожатые подошли к «Резерву», г-н Моррель пошел к ним навстречу, окруженный матросами и солдатами, которым он повторил обещание, данное Дантесу, что он будет назначен капитаном на место покойного Леклера. Увидев его, Дантес выпустил руку Мерседес и уступил место г-ну Моррелю. Арматор и невеста, подавая пример гостям, взошли по лестнице в столовую, и еще добрых пять минут деревянные ступени скрипели под тяжелыми шагами гостей.
22. Губы его посинели, и видно было, как под загорелой кожей вся кровь, приливая к сердцу, отхлынула от лица. Дантес возле себя посадил г-на Морреля и Данглара: первого по правую, второго по левую сторону; потом сделал знак рукой, приглашая остальных рассаживаться, как им угодно.



23. Уже путешествовали вокруг стола румяные и пахучие аральские колбасы, лангусты в ослепительных латах, венерки с розовой раковиной, морские ежи, напоминающие каштаны с их колючей оболочкой, кловиссы, с успехом заменяющие южным гастрономам северные устрицы; словом, все те изысканные лакомства, которые волна выносит на песчаный берег и которые благодарные рыбаки называют общим именем «морские плоды».
24. В тот же самый день, в тот же самый час на улице Гран-Кур, против фонтана Медуз, в одном из старых аристократических домов, выстроенных архитектором Пюже, тоже праздновали обручение. Но герои этого праздника были не простые люди, не матросы и солдаты, они принадлежали к высшему марсельскому обществу.
25. Это были старые сановники, вышедшие в отставку при узурпаторе; военные, бежавшие из французской армии в армию Конде; молодые люди, которых родители – все еще не уверенные в их безопасности, хотя уже поставили за них по четыре или по пять рекрутов, – воспитали в ненависти к тому, кого пять лет изгнания должны были превратить в мученика, а пятнадцать лет Реставрации – в бога.
26. Все сидели за столом, и разговор кипел всеми страстями того времени, страстями особенно неистовыми и ожесточенными, потому что на юге Франции уже пятьсот лет политическая вражда усугубляется враждой религиозной.
27. Император, ставший королем острова Эльба, после того как он был властителем целого материка, и правящий населением в пять-шесть тысяч душ, после того как сто двадцать миллионов подданных на десяти языках кричали ему: «Да здравствует Наполеон!» – казался всем участникам пира человеком, навсегда потерянным для Франции и престола.
28. Сановники вспоминали его политические ошибки, военные рассуждали о Москве и Лейпциге, женщины – о разводе с Жозефиной. Этому роялистскому сборищу, которое радовалось – не падению человека, а уничтожению принципа, – казалось, что для него начинается новая жизнь, что оно очнулось от мучительного кошмара.
29. Осанистый старик, с орденом св. Людовика на груди, встал и предложил своим гостям выпить за короля Людовика XVIII. То был маркиз де Сен-Меран.
30. Этот тост в честь гартвельского изгнанника и короля – умиротворителя Франции был встречен громкими кликами; по английскому обычаю, все подняли бокалы; женщины откололи свои букеты и усеяли ими скатерть. В этом едином порыве была почти поэзия.

### **3. Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### **4. Основные теоретические сведения**

Современные информационные технологии ставят своей целью автоматизировать различные аспекты человеческой деятельности. Для того чтобы машины могли оказывать человеку реальную помощь, они должны разбираться в тонкостях и деталях нашего мира. Только в этом случае компьютеры смогут принимать правильные решения. Для этого ЭВМ должны иметь модель нашего мира, отражающую все многообразие реально существующих взаимодействующих объектов. На сегодняшнем уровне развития информационных технологий только объектно-ориентированное программирование (ООП) способно создавать модели, не имеющие семантического разрыва с реальным миром. Жизнь становится невыносимой без обмена информационными потоками через Интернет. В этом нам помогают сетевые технологии, в частности технология Java. Границы Java все время расширяются. Сначала Java (официальный день рождения технологии Java — 23 мая 1995 г.) предназначалась для программирования бытовых электронных устройств, таких, как телефоны. Потом Java стала применяться для создания активных гипертекстовых страниц — появились апплеты. Затем оказалось, что на Java можно создавать полноценные приложения. Их графические элементы стали оформлять в виде компонентов — появились JavaBeans, с которыми Java вошла в мир распределенных систем и промежуточного программного обеспечения, тесно связавшись с технологией CORBA. Остался один шаг до программирования серверов — этот шаг был сделан — появились сервлеты и EJB (Enterprise JavaBeans). Серверы должны взаимодействовать с системами управления базами данных (СУБД) — появились драйверы JDBC (Java DataBase Connection). Многие СУБД и даже операционные системы (ОС) включили Java в свое ядро, например, Oracle, Linux. Сегодня можно сказать, что Java — технология, которая быстро охватывает все новые и новые области.

Широкое распространение технологии Java связано с тем, что она использует новый, специально созданный язык программирования, который так и называется — язык Java. Этот язык, созданный на базе языков Smalltalk, Pascal, C, оказался удобным для изучения. Программы, написанные на нём, легко читаются и отлаживаются. Язык Java становится языком обучения объектно-ориентированному программированию, так же, как язык Pascal был языком обучения структурному программированию.

### **ВЫПОЛНЕНИЕ JAVA-ПРОГРАММЫ**

Программа, написанная на языке высокого уровня, к которому относится и язык Java, так называемый исходный модуль, не может быть сразу же выполнена. Ее сначала нужно откомпилировать, т. е. перевести в последовательность машинных команд — объектный модуль. Но и он, как правило, не может быть сразу же выполнен: объектный модуль нужно еще скомпоновать с библиотеками использованных в модуле функций и разрешить перекрестные ссылки между секциями объектного модуля, получив в результате загрузочный модуль — полностью готовую к выполнению программу.

Исходный модуль, написанный на Java, не может избежать этих процедур, но здесь проявляется главная особенность технологии Java — программа компилируется сразу в машинные команды, но не команды какого-то конкретного процессора, а в команды так называемой виртуальной машины Java (JVM, Java Virtual Machine). Виртуальная машина Java — это совокупность команд вместе с системой их выполнения. Команды JVM короткие, большинство из них имеет длину 1 байт, отчего команды JVM называют байт^кодами (bytecodes), хотя имеются команды длиной 2 и 3 байта.

Другая особенность Java — все стандартные функции, вызываемые в программе, подключаются к ней только на этапе выполнения, а не включаются в байт-коды. Как говорят специалисты, происходит динамическая компоновка (dynamic binding). Это тоже сильно уменьшает объем откомпилированной программы.

Итак, на первом этапе программа, написанная на языке Java, переводится компилятором в байт-коды. Эта компиляция не зависит от типа какого-либо конкретного процессора и архитектуры некоего конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы. Байт-коды записываются в одном или нескольких файлах, могут храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодами. Затем полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM. При этом не важен ни тип процессора, ни архитектура компьютера.

Интерпретация байт-кодов и динамическая компоновка значительно замедляют выполнение программ. Это не имеет значения в тех ситуациях, когда байт-коды передаются по сети, сеть все равно медленнее любой интерпретации, но в других ситуациях требуется мощный и быстрый компьютер. Поэтому постоянно идет усовершенствование интерпретаторов в сторону увеличения скорости интерпретации. Разработаны JIT-компиляторы (Just-In-Time), запоминающие уже интерпретированные участки кода в машинных командах процессора и просто выполняющие эти участки при повторном обращении, например, в циклах. Это значительно увеличивает скорость повторяющихся вычислений. При выполнении программ Java требуется интерпретация команд JVM в команды конкретного процессора, а значит, нужна программа-

интерпретатор, причем для каждого типа процессоров и для каждой архитектуры компьютера следует написать свой интерпретатор.

Эта задача уже решена практически для всех компьютерных платформ. Кроме реализации JVM для выполнения байт-кодов на компьютере необходим набор функций, вызываемых из байт-кодов и динамически komponующихся с байт-кодами. Этот набор оформляется в виде библиотеки классов Java, состоящей из одного или нескольких пакетов. Каждая функция может быть записана байт-кодами, но, поскольку она будет храниться на конкретном компьютере, ее можно записать прямо в системе команд этого компьютера, избегнув тем самым интерпретации байт-кодов. Такие функции называют "родными" методами (native methods). Применение "родных" методов ускоряет выполнение программы.

Фирма SUN Microsystems — создатель технологии Java — бесплатно распространяет набор необходимых программных инструментов для полного цикла работы с этим языком программирования: компиляции, интерпретации, отладки, включающий и богатую библиотеку классов, под названием JDK (Java Development Kit).

Набор программ и классов JDK содержит:

- ✓ компилятор javac из исходного текста в байт-коды;
- ✓ интерпретатор java, содержащий реализацию JVM;
- ✓ программу просмотра апплетов appletviewer, заменяющую браузер;
- ✓ отладчик jdb;
- ✓ дизассемблер javap;
- ✓ программу архивации и сжатия jar;
- ✓ программу сбора документации javadoc;
- ✓ программу javah генерации заголовочных файлов языка C;
- ✓ программу javakey добавления электронной подписи;
- ✓ программу native2ascii, преобразующую бинарные файлы в текстовые;
- ✓ программы rmic и rmiregistry для работы с удаленными объектами;
- ✓ программу serialver, определяющую номер версии класса;
- ✓ библиотеки и заголовочные файлы "родных" методов;
- ✓ библиотеку классов Java API (Application Programming Interface).

Кроме JDK компания SUN отдельно распространяет еще и набор JRE (Java Runtime Environment) — набор программ и пакетов классов JRE, который содержит все необходимое для выполнения байт-кодов, в том числе интерпретатор java и библиотеку классов. Именно JRE или его аналог других фирм содержится в браузерах, умеющих выполнять программы на Java, ОС и СУБД. Хотя JRE входит в состав JDK, фирма SUN распространяет этот набор и отдельным файлом.

Набор JDK упаковывается в самораспаковывающийся архив. Раздобыв каким-либо образом этот архив: "выкачав" из Internet, с <http://java.sun.com/products/jdk/> или какого-то другого адреса, получив компакт-диск, остается только запустить файл с архивом на выполнение. Откроется окно установки, в

котором среди всего прочего будет предложено выбрать каталог (directory) установки, например, C:\jdk1.3. При согласии с предлагаемым каталогом беспокоиться больше не о чем, иначе необходимо проверить после установки значение переменной PATH, набрав в командной строке окна Command Prompt ОС Windows команду set. Переменная PATH должна содержать полный путь к подкаталогу bin этого каталога. Иначе добавьте путь, например, C:\jdk1.3\bin. Нужно определить и специальную переменную CLASSPATH, содержащую пути к архивным файлам и каталогам с библиотеками классов. Системные библиотеки Java 2 подключаются автоматически, без переменной CLASSPATH. Не следует распаковывать zip- и jar-архивы.

После установки вы получите каталог с названием, например, jdk1.3.1, а в нем подкаталоги:

- ✓ bin, содержащий исполнимые файлы;
- ✓ demo, содержащий примеры программ;
- ✓ docs, содержащий документацию;
- ✓ include, содержащий заголовочные файлы "родных" методов;
- ✓ jre, содержащий набор JRE;
- ✓ old-include, для совместимости со старыми версиями;
- ✓ lib, содержащий библиотеки классов и файлы свойств;
- ✓ src, содержащий исходные тексты программ JDK. В новых версиях вместо каталога имеется упакованный файл src.jar.

Набор JDK содержит исходные тексты большинства своих программ, написанные на Java, что позволяет точно знать, как работает тот или иной метод.

Написать программу на Java можно в любом текстовом редакторе, сохранив файл в текстовом формате с расширением java.

Пусть, например, именем файла будет MyProgram.java, а сам файл сохранен в текущем каталоге.

После создания этого файла из командной строки вызывается компилятор javac и ему передается исходный файл как параметр:

```
javac MyProgram.java
```

Компилятор создает в том же каталоге по одному файлу на каждый класс, описанный в программе, называя каждый файл именем класса с расширением class. Допустим, имеется только один класс, названный MyProgram, тогда получаем файл с именем MyProgram.class, содержащий байт-коды. Если компиляция прошла успешно, то компилятор ничего не сообщит, на экране появится только приглашение ОС. Если же компилятор заметит ошибки, то он выведет на экран сообщения о них. Большое преимущество компилятора JDK в том что он "отлавливает" много ошибок и выдает подробные и понятные сообщения о них.

Далее из командной строки вызывается интерпретатор байт-кодов java, которому передается файл с байт-кодами, причем его имя записывается без расширения:

```
java MyProgram
```

На экране появляется вывод результатов работы программы или сообщения об ошибках времени выполнения.

Сразу же после создания Java, уже в 1996 г., появились интегрированные среды разработки программ для Java, и их число все время возрастает. Некоторые из них являются просто интегрированными оболочками над JDK, вызывающими из одного окна текстовый редактор, компилятор и интерпретатор. Эти интегрированные среды требуют предварительной установки JDK. Другие содержат JDK в себе или имеют собственный компилятор, например, Java Workshop фирмы SUN Microsystems, JBuilder фирмы Inprise, Visual Age for Java фирмы IBM и множество других программных продуктов. Их можно устанавливать, не имея под руками JDK. Большинство интегрированных сред являются средствами визуального программирования и позволяют быстро создавать пользовательский интерфейс, т.е. относятся к классу средств RAD (Rapid Application Development). Выбор какого-либо средства разработки диктуется, во-первых, возможностями компьютера, ведь визуальные среды требуют больших ресурсов, во-вторых, личным вкусом, в-третьих, уже после некоторой практики, преимуществами компилятора, встроенного в программный продукт.

Рекомендуем использовать среду разработки **IntelliJ IDEA**.

### **Знакомство с средой разработки IntelliJ IDEA**

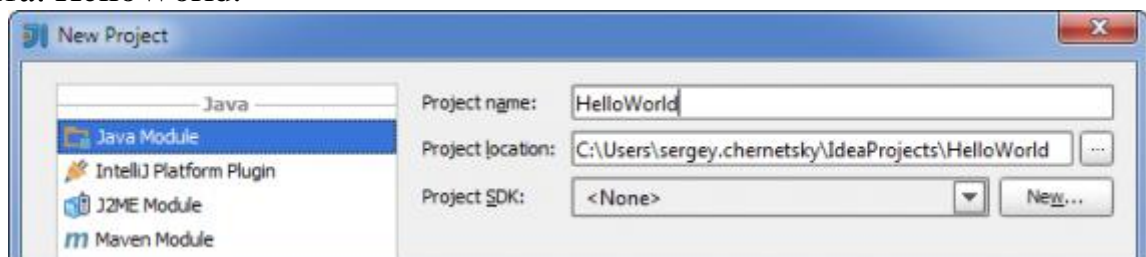
#### **Создание проекта**

Создание любого приложения в IntelliJ IDEA начинается с создания проекта (узнать, почему проект необходим, можно в IntelliJ IDEA Help, по ссылке [Project](#)), так что первым нашим шагом будет создание проекта «Hello, World». Этот проект будет содержать модуль Java для нашего приложения Java.

1. Если ни один проект в данный момент не открыт, нажмите кнопку Create New Project на экране приветствия. В противном случае, выберите пункт New Project в меню File. В результате откроется мастер создания проекта.

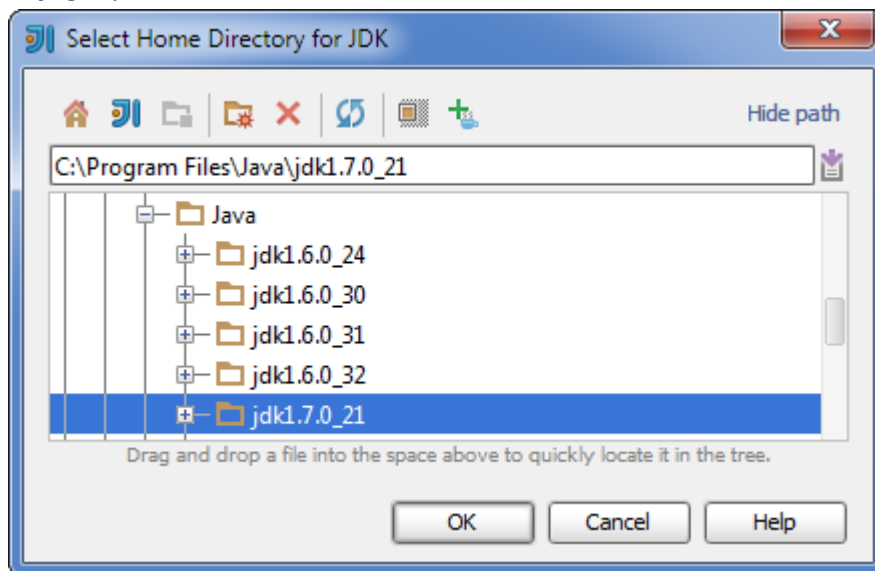
2. В левой панели выберите Java Module.

3. В правой части страницы, в поле Project name, введите название проекта: HelloWorld.

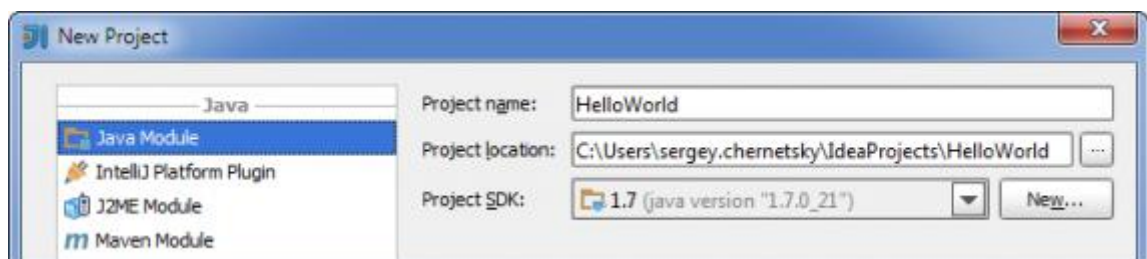


4. Если до этого вы никогда не настраивали JDK в IntelliJ IDEA (в таком случае в поле Project SDK стоит <None>), необходимо сделать это сейчас. Вместо <None> нажмите New и в подменю выберите JDK. В окне Select Home

Directory for JDK выберите директорию, в которую устанавливалось JDK и нажмите OK.



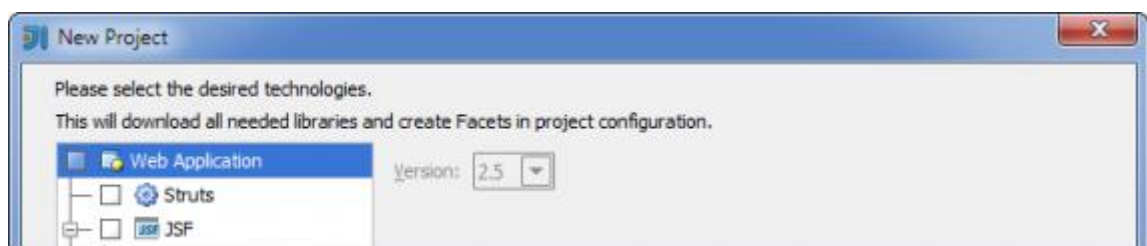
Версия JDK, которую вы выбрали, появится в поле Project SDK.



Кликните Next.

Учтите, что указанная версия JDK будет связана по умолчанию со всеми проектами и Java модулями, которые в дальнейшем будут создаваться.

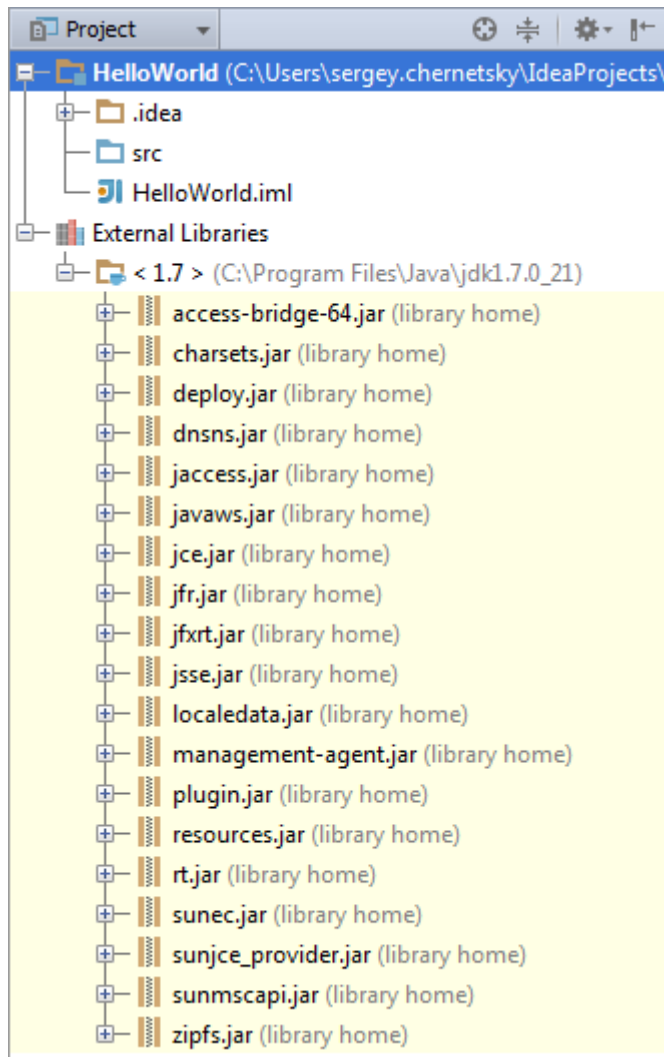
5. На следующей странице осуществляется выбор мастера для указания дополнительных технологий, которые будут поддерживаться в нашем модуле.



Поскольку наше приложение будет «старым добрым приложением Java», мы не нуждаемся в любой из этих технологий. Так что просто нажмите кнопку Finish. Подождите, пока IntelliJ IDEA создает необходимые структуры проекта. Когда этот процесс завершится, вы можете увидеть структуру Вашего нового проекта в окне Project.

## Изучение структуры проекта

Давайте пробежимся взглядом по структуре проекта.



В дереве проекта видим две директории верхнего уровня:

HelloWorld. Это узел, содержащий ваш Java модуль. Папки .idea и файлы внутри директории HelloWorld.iml используются для хранения данных конфигурации вашего проекта и модулей соответственно. SRC папки содержат исходный код.

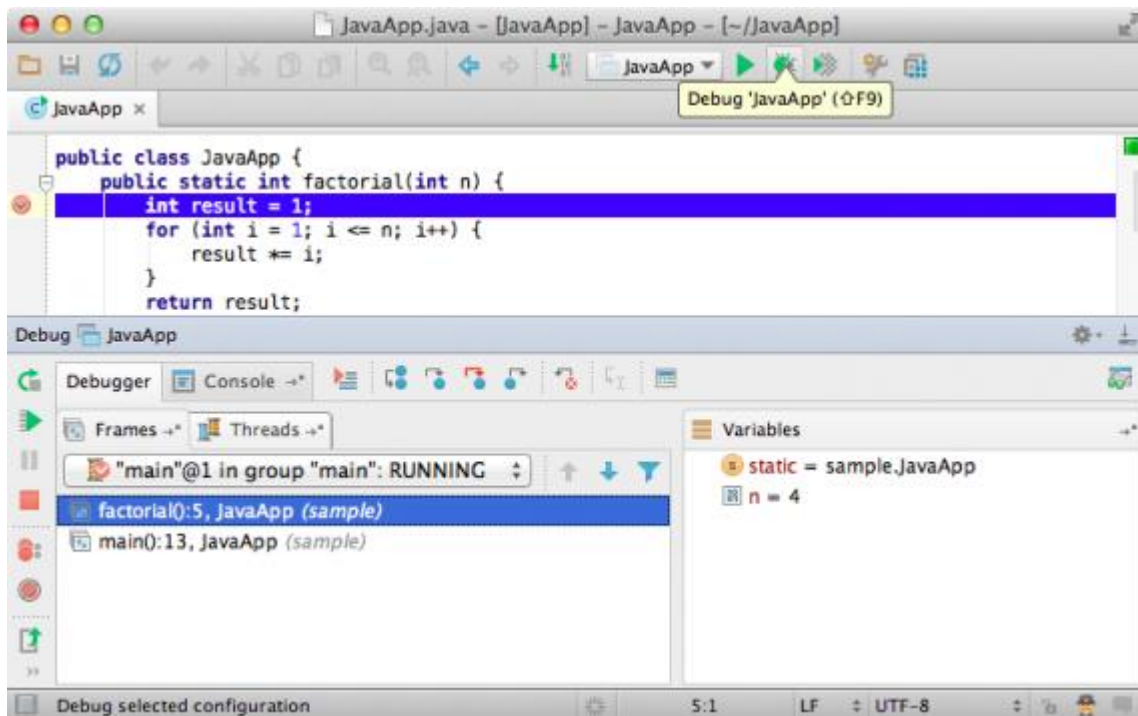
External Libraries (внешние библиотеки). Это категория, которая представляет все «внешние» ресурсы, необходимые для вашего проекта. В настоящее время в этой категории файлы .jar, из выбранного нами JDK.

Из всех упомянутых папок в рассматриваемом примере нам понадобится только SRC. (Для получения дополнительной информации об окнах инструментов в целом и окне Project в частности, см. [IntelliJ IDEA Tool Windows](#)) и [Project Tool Window](#) в IntelliJ IDEA Help)

## Отладчик



Запуск отладчика. После того как вы настроите конфигурацию запуска вашего проекта, вы можете запускать его в режиме отладки, нажав Shift + F9



В окне отладчика вы можете видеть стек вызовов функций и список потоков, с их состояниями, переменными и окнами просмотра состояния. Когда вы выбираете контекст вызова функции, вы можете просмотреть значения переменных соответствующих выбранному контексту.

Полезные клавиатурные сокращения отладчика

Установить/снять точку останова — Ctrl + F8 (Cmd + F8 для Mac)

Возобновить выполнение программы — F9

Перейти к следующей инструкции — F8

Перейти внутрь функции — F7

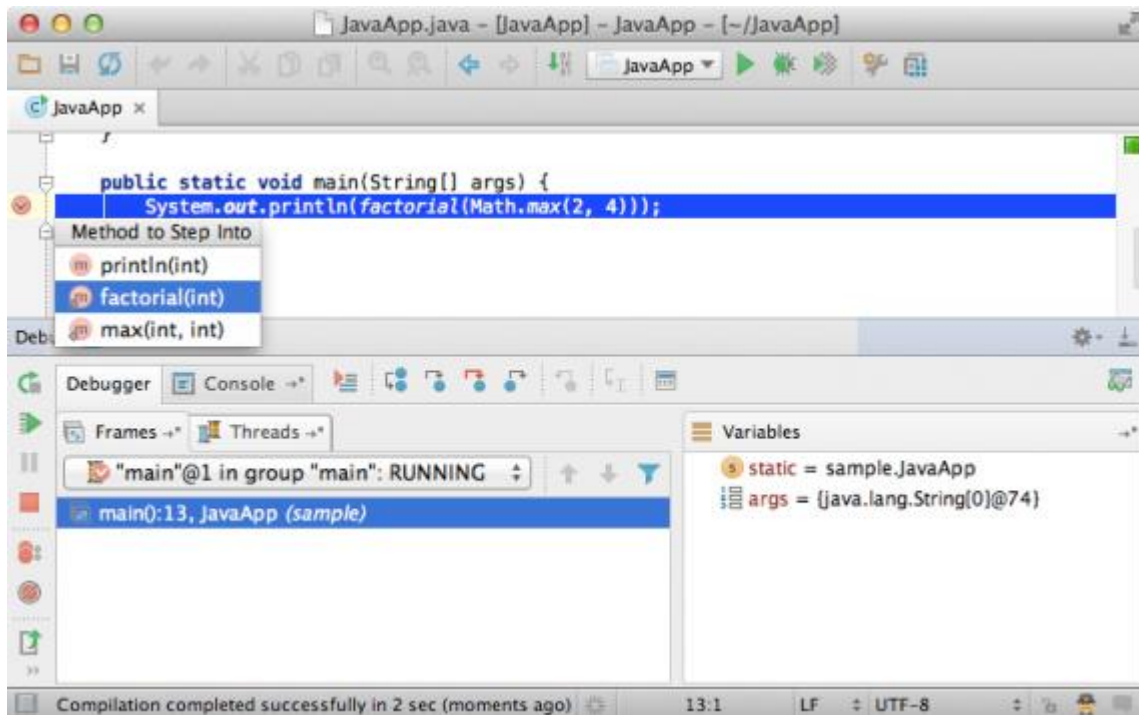
Приостановить выполнение — Ctrl + F2 (Cmd + F2)

Переключить между просмотром списка точек останова и подробной информацией о выбранной точке — Shift + Ctrl + F8 (Shift + Ctrl + F8)

Запустить отладку кода с точки на которой стоит курсор — Shift + Ctrl + F9 (если это внутри метода main())

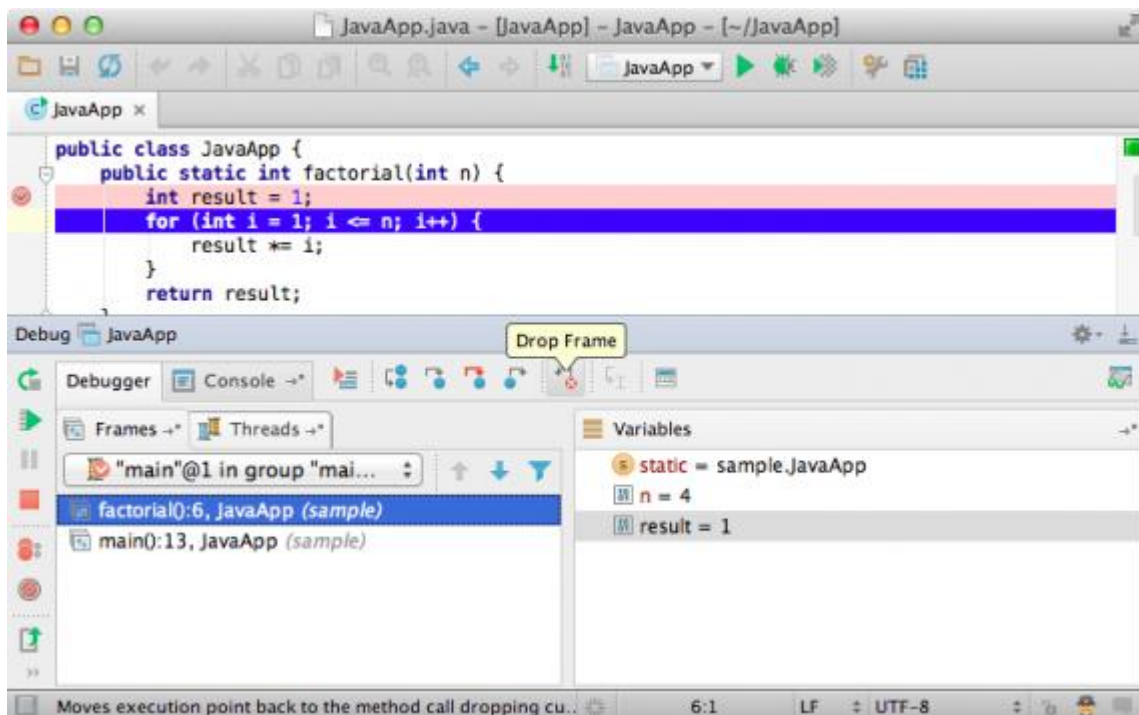
Умный переход внутрь

Иногда вам надо при пошаговой отладке перейти внутрь определенного метода, но не первого который будет вызван. В таком случае вы можете нажать Shift + F7 (Cmd + F7 для Mac) чтобы выбрать из предложенного списка метод который вам нужен. Это может сэкономить вам массу времени.

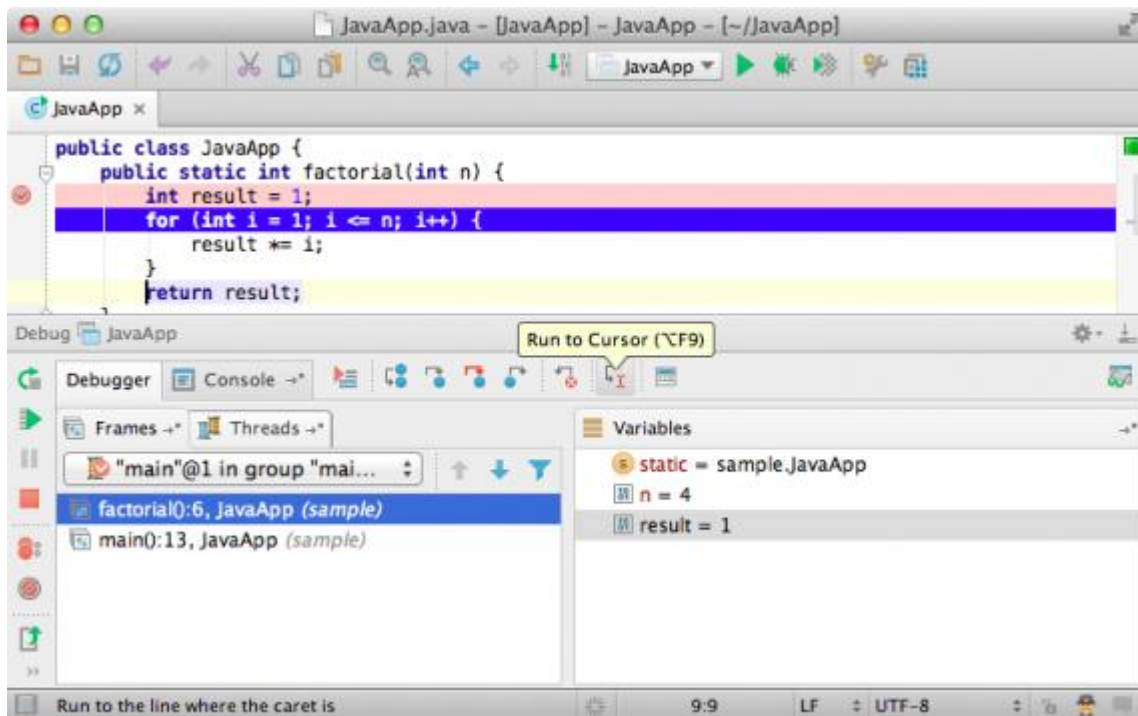


Удалить контекст вызова функции

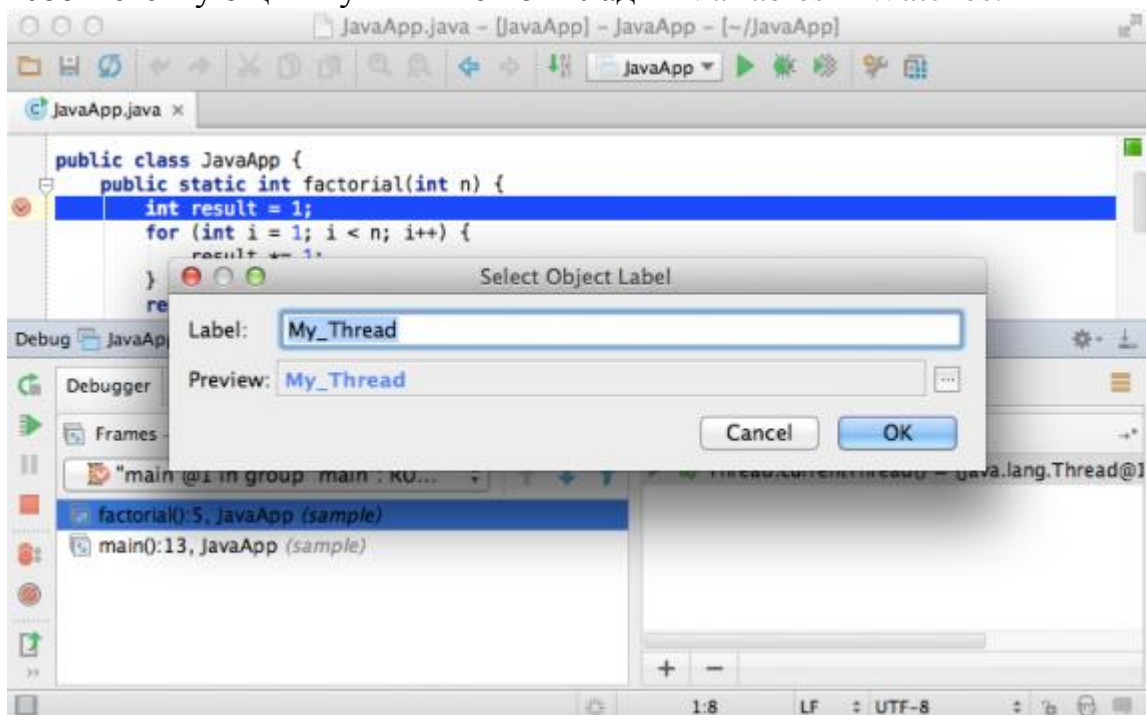
Если вам нужно «вернуться назад во времени» во время отладки, вы можете сделать это удалив контекст вызова функции. Это сильно поможет если вы по ошибке зашли слишком глубоко. Таким образом вы не откатите глобальное состояние выполнения программы, но как минимум вы вернетесь назад по стеку вызовов функций.



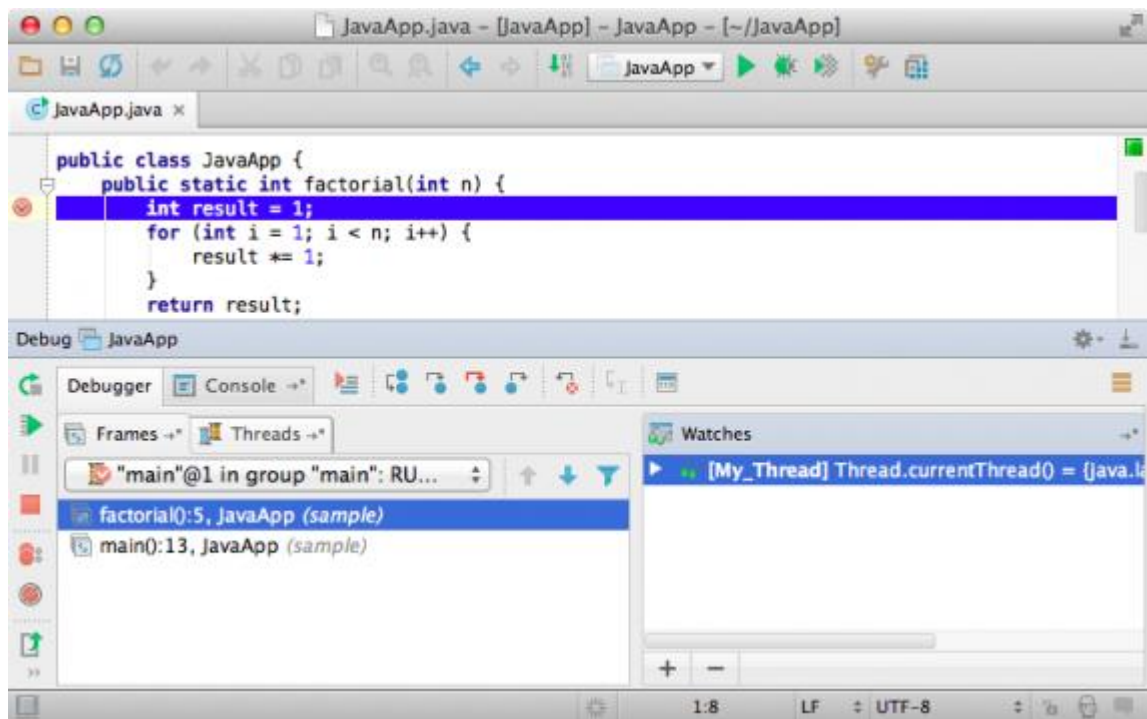
Переход к курсору. Иногда вам надо возобновить выполнение программы и остановиться на какой-то другой строчке кода, не создавая точку останова. Это легко — просто нажмите **Alt + F9**.



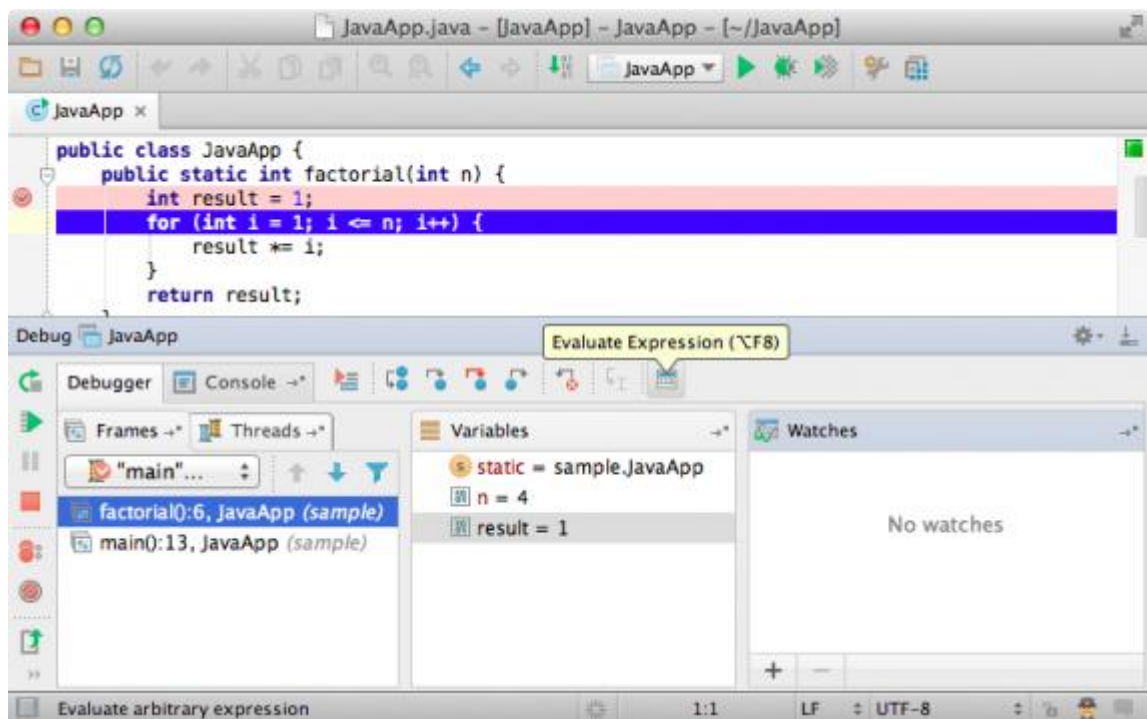
Пометить элемент. Если вы хотите легко видеть какой-то элемент во время отладки, вы можете добавить к нему цветную метку, нажав **F11** или выбрав соответствующий пункт в меню вкладки **Variables** и **Watches**.



Когда этот элемент появится в списке, вы увидите его метку.

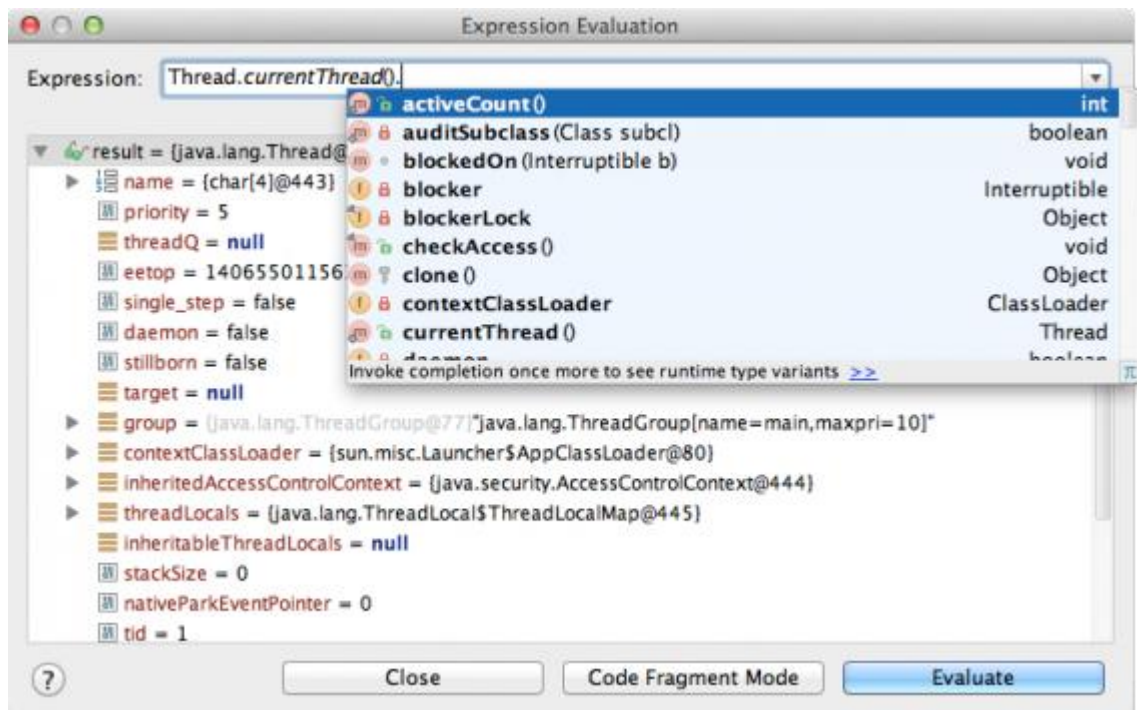


Вычислить выражение. В режиме отладки вы можете вычислить любое выражение, с помощью очень мощного инструмента вызываемого нажатием Alt + F8.

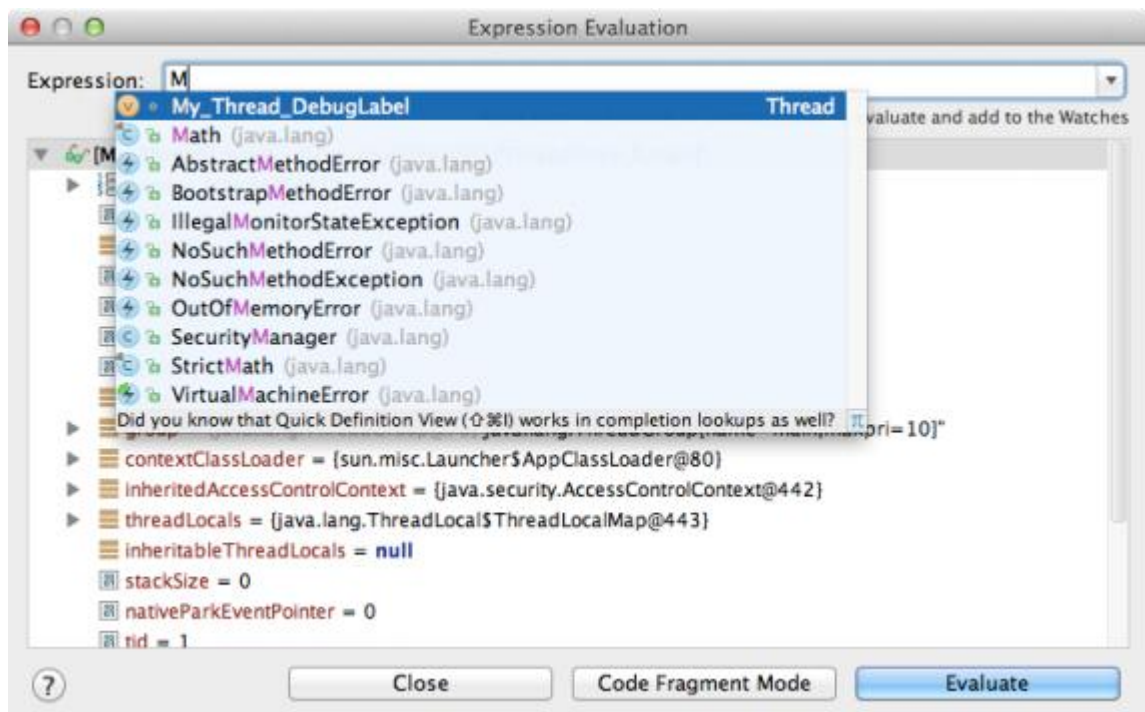


Этот инструмент предоставляет автодополнение кода как и редактор, так что ввести любое выражение будет очень просто.

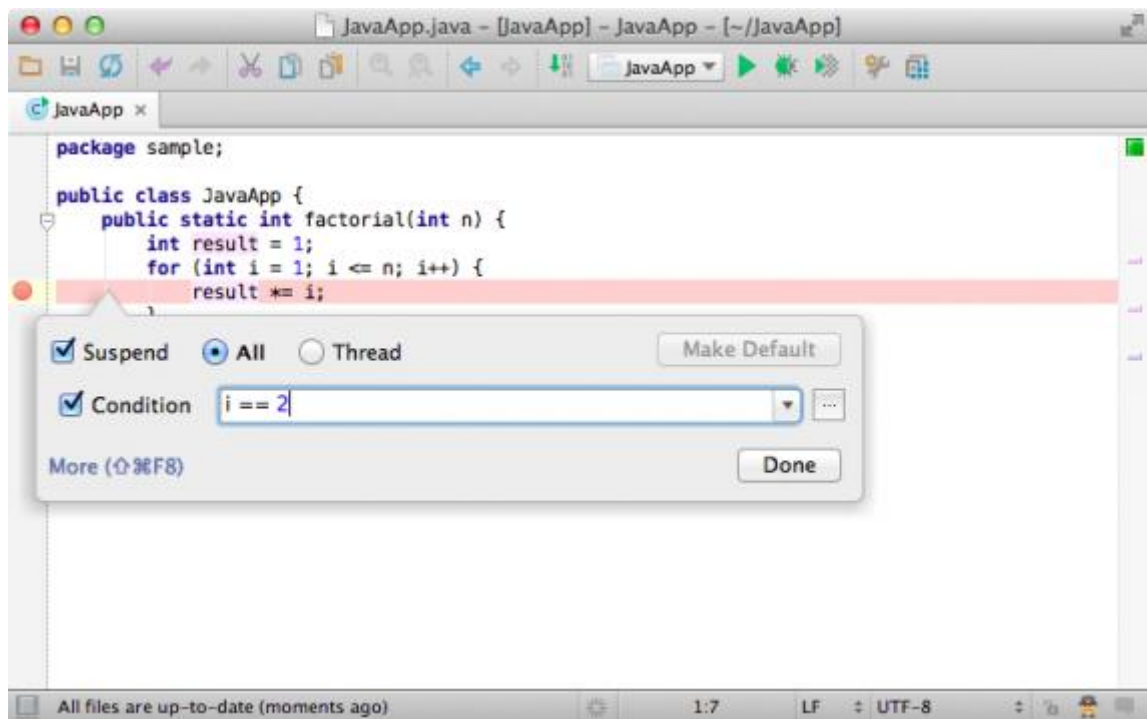




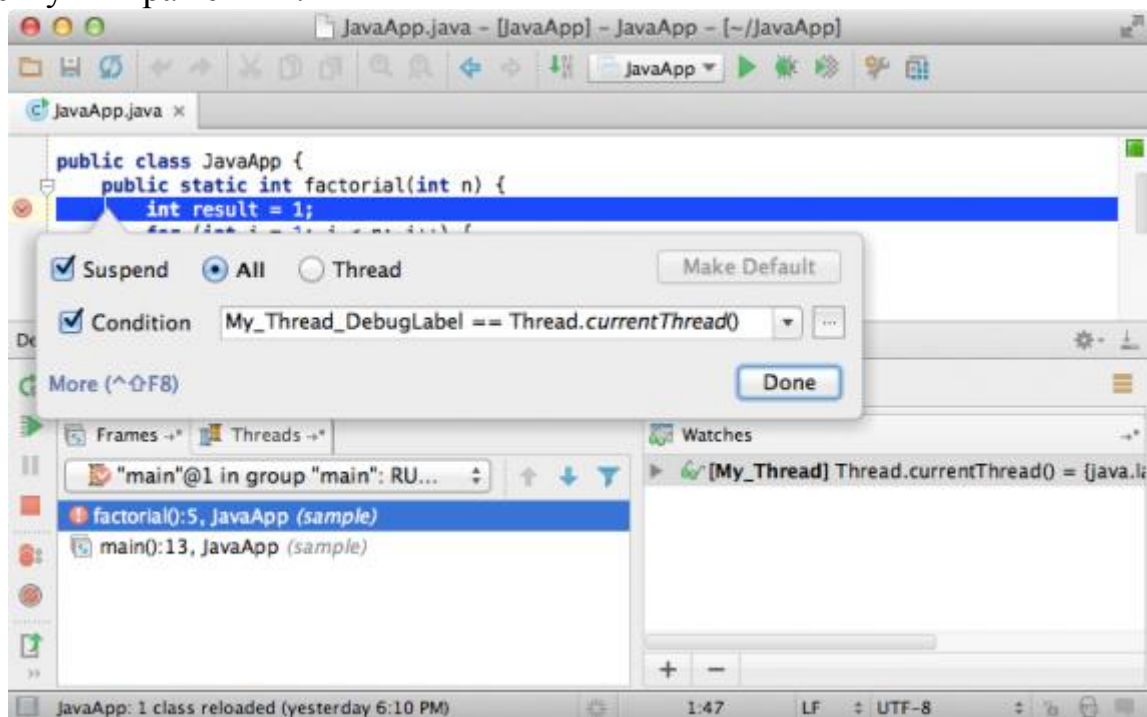
Если у вас есть какие-то элементы с метками, автодополнение кода покажет вам эти метки чтобы вы могли легко найти нужные элементы и вычислить их значения.



Состояние и настройки точки останова  
Если вы хотите поменять какие-то настройки точки останова, вы можете нажать Shift + Ctrl + F8 (Shift + Cmd + F8 для Mac). Во всплывающем окне вы можете ввести нужные вам параметры.



Если у вас какой-то элемент имеет метку, вы также можете использовать эту метку в выражениях.

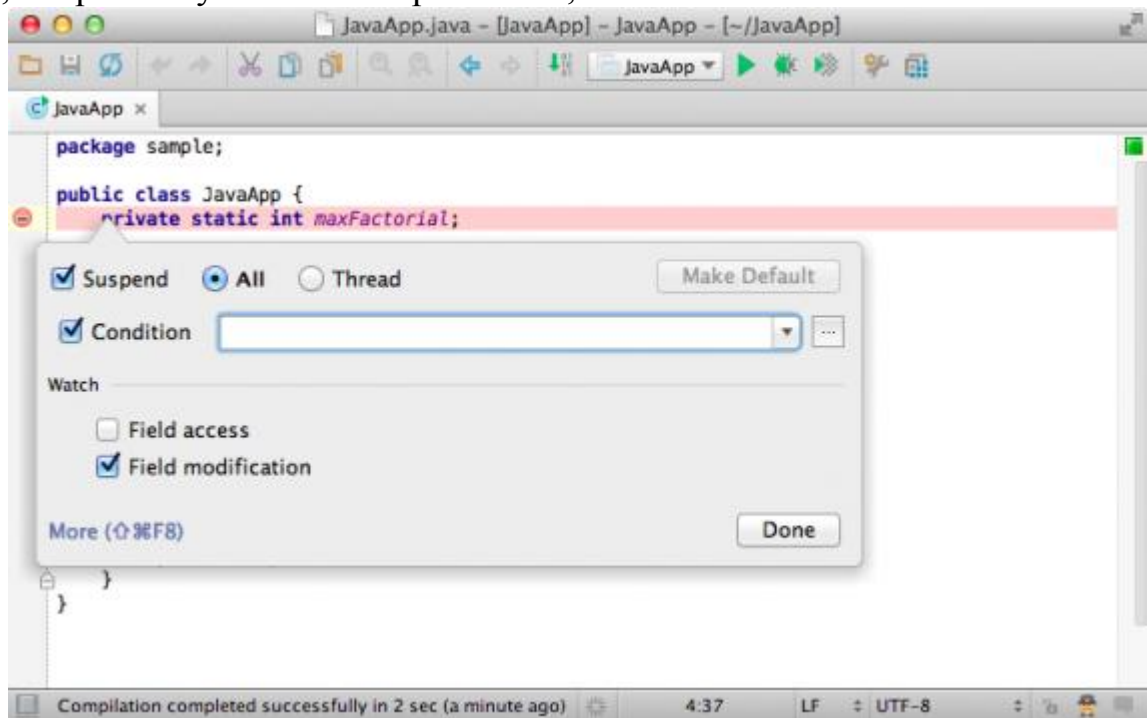


Чтобы получить список всех точек останова в вашем проекте (с расширенными настройками), снова нажмите Shift + Ctrl + F8 (Shift + Cmd + F8 для Mac).

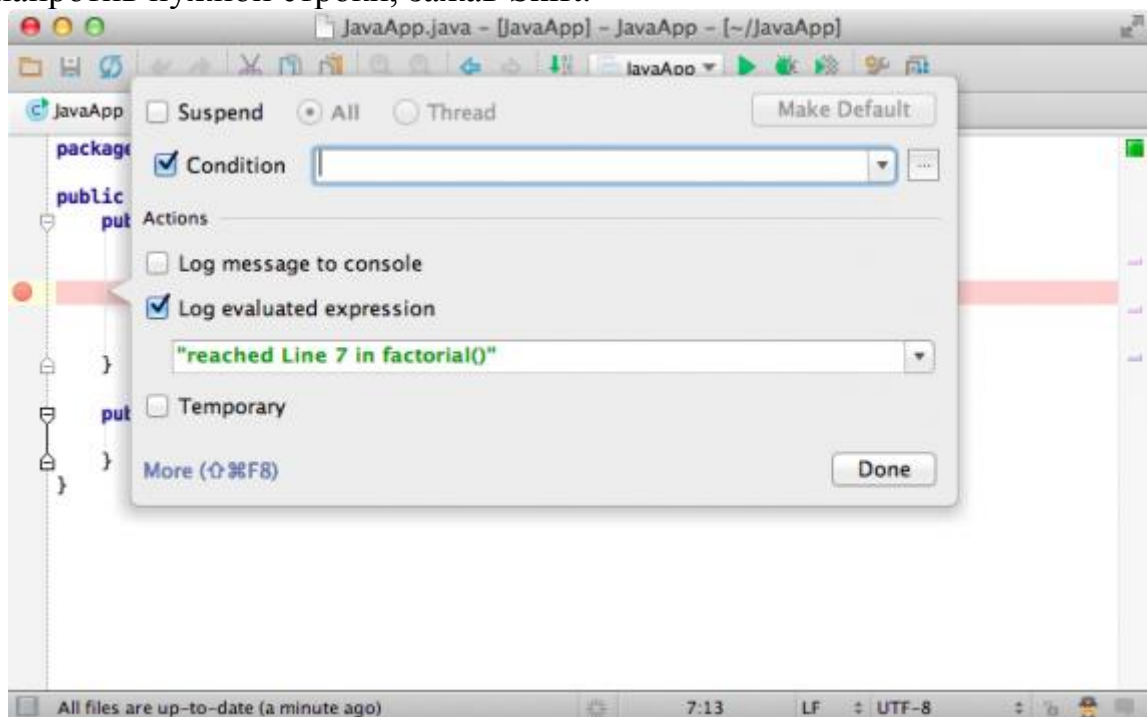
#### Точки останова переменной

В дополнение к условным точкам останова, вы можете также использовать точки останова переменной. Такие точки срабатывают, когда производится чтение или запись в какую-то переменную. Для того чтобы

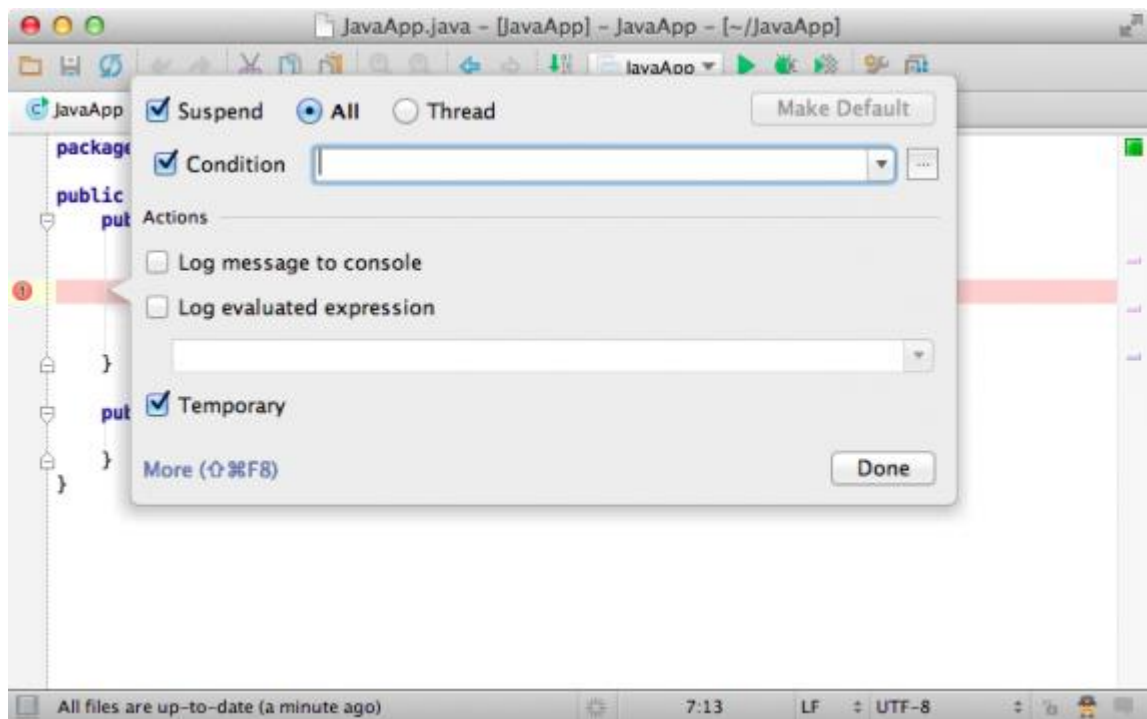
создать такую точку останова, кликните на панель слева от редактируемого текста, напротив нужной вам переменной, зажав Alt.



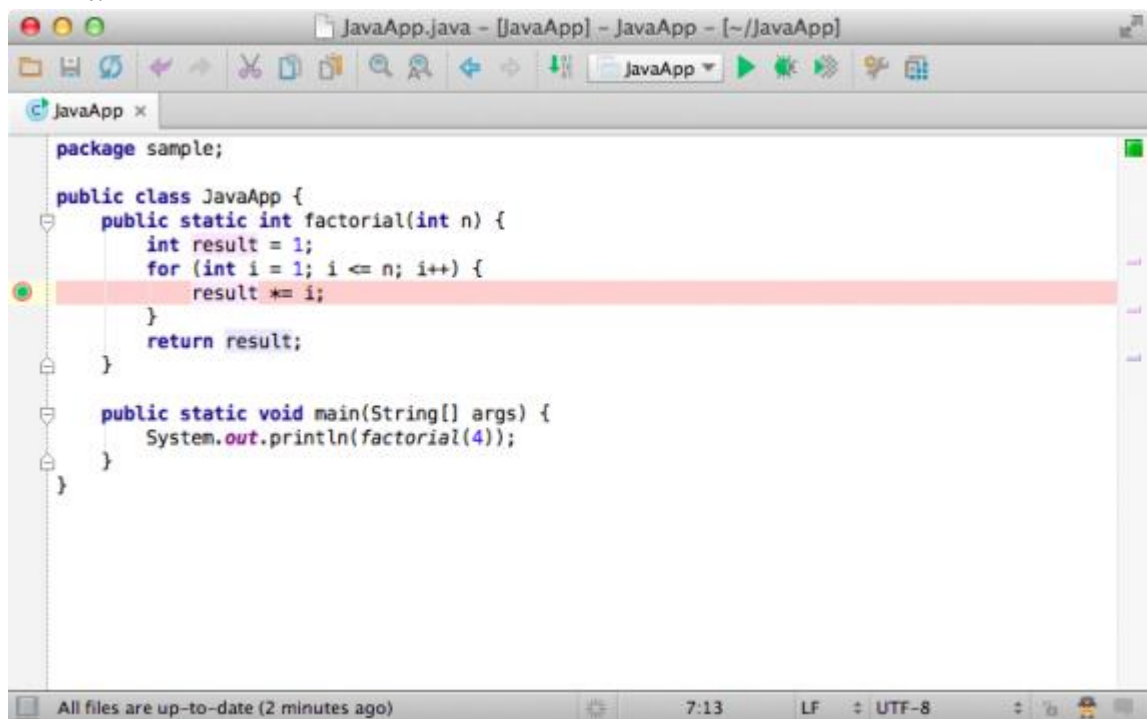
Точки останова по событиям. Еще одна полезная возможность — вычислить определенное выражение в нужной вам строке кода не прерывая выполнение. Для этого вам нужно кликнуть на панель слева от редактируемого кода напротив нужной строки, зажав Shift.



Временные точки останова. Для того чтобы создать точку останова которая сработает только один раз, кликните на панель слева от кода зажав Shift + Alt.

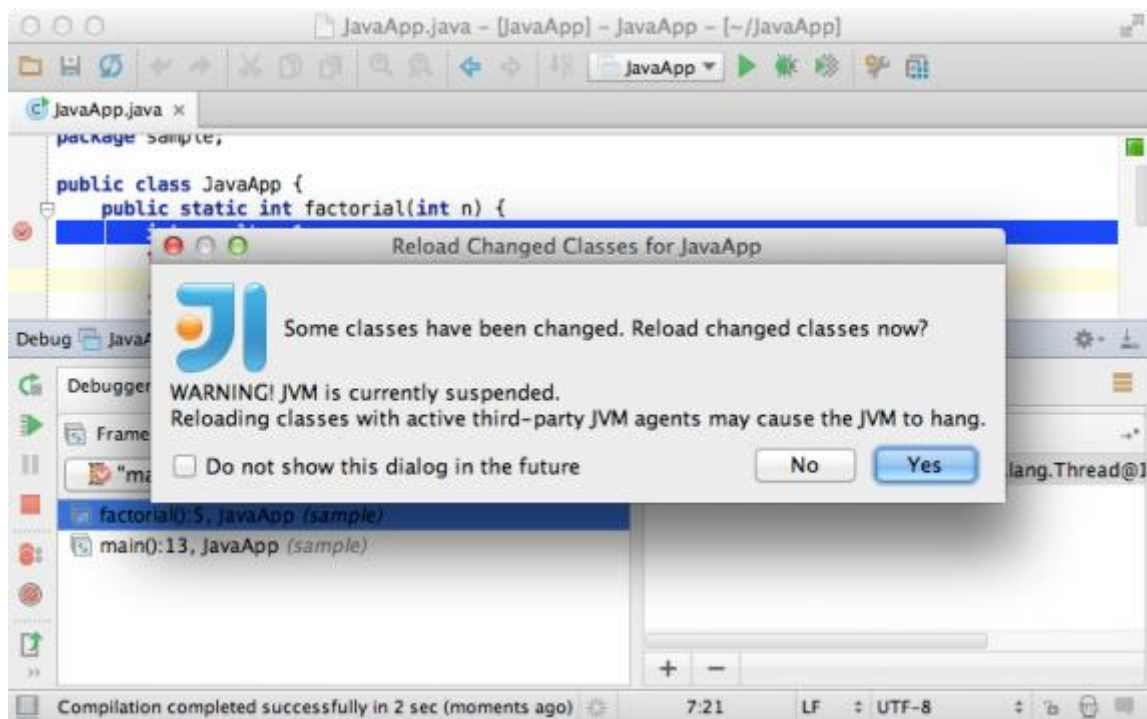


Отключить точки останова. Также очень полезно знать, что любая точка останова может быть быстро отключена по нажатию на панель слева от кода с зажатым Alt.



Загрузка изменений и быстрая замена. Иногда вам нужно внести небольшие изменения в код без прерывания процесса отладки. Так как виртуальная машина Java поддерживает возможность HotSwap, среда разработки в режиме отладки предлагает вам перезагрузить измененные классы когда вы их скомпилируете.

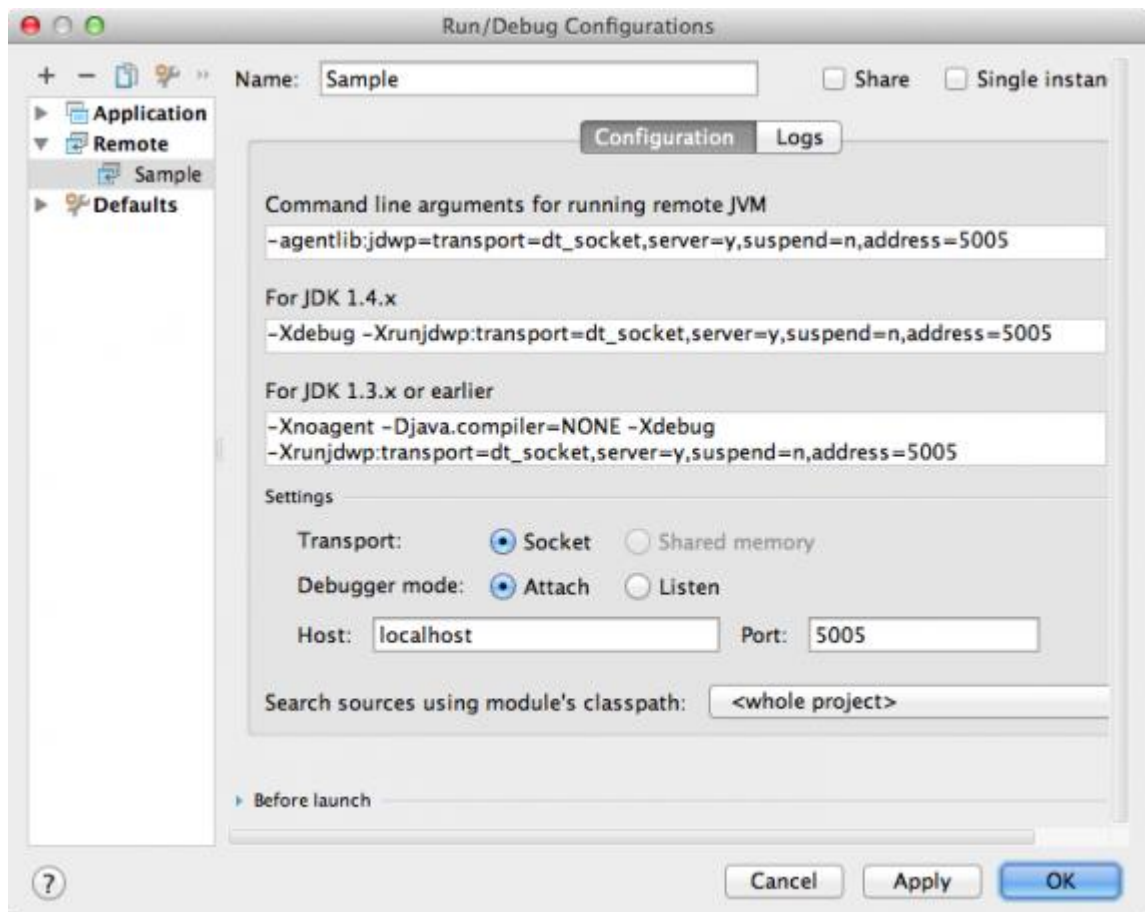




Не забывайте, что функционал HotSwap в Java машине имеет ряд ограничений и не позволяет перезагружать статические поля и методы.

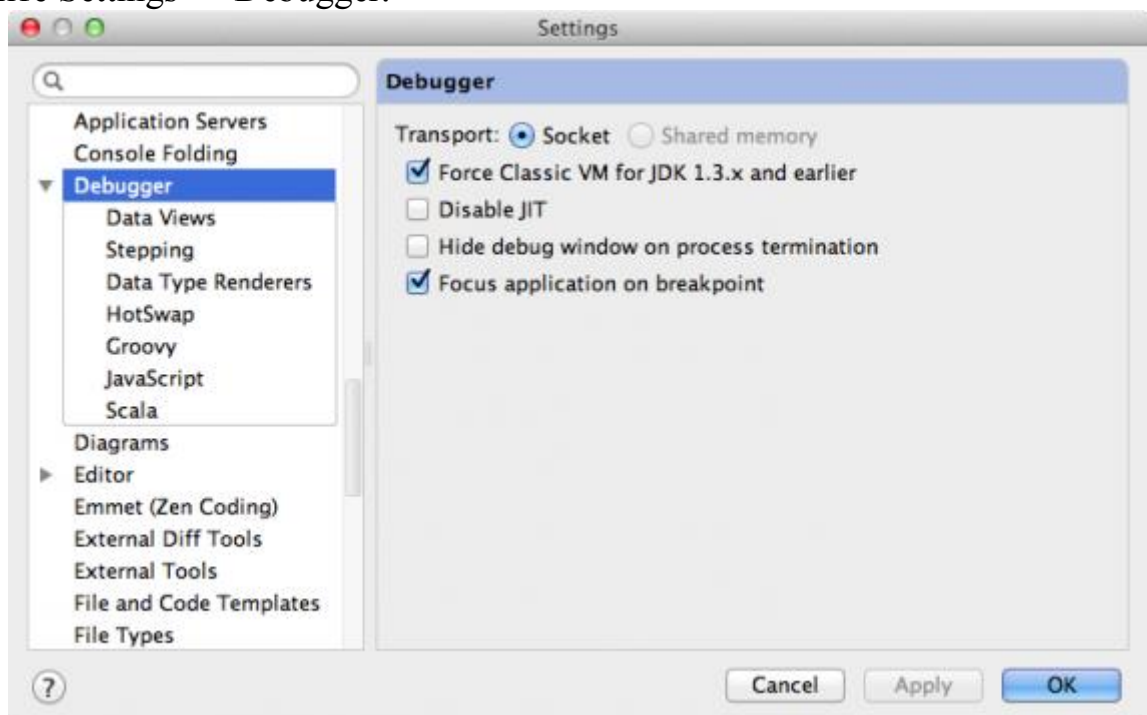
Удаленная отладка. Последняя вещь в IntelliJ IDEA о которой вам точно нужно знать это удаленная отладка. Удаленная отладка — подключение отладчика к уже запущенной у вас или на другом компьютере Java машине по сетевому порту. Таким образом можно подключить отладчик к серверу приложений, запущенному на сервере.

Чтобы создать конфигурацию для удаленного запуска, перейдите к редактированию конфигураций (Edit configurations) и нажмите «добавить конфигурацию удаленного запуска» (Remote). Убедитесь что вы указали правильное имя компьютера и порт прежде чем запустить эту конфигурацию.



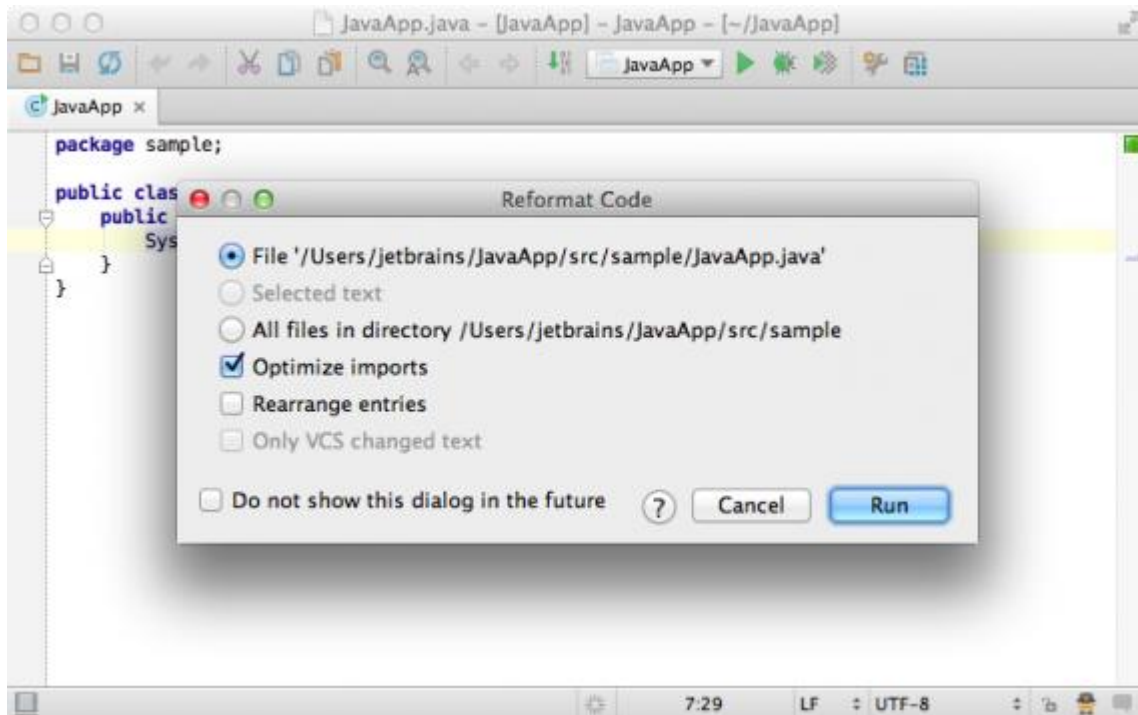
## Настройки

Если вы хотите поменять настройки отладчика по умолчанию, нажмите Settings → Debugger.

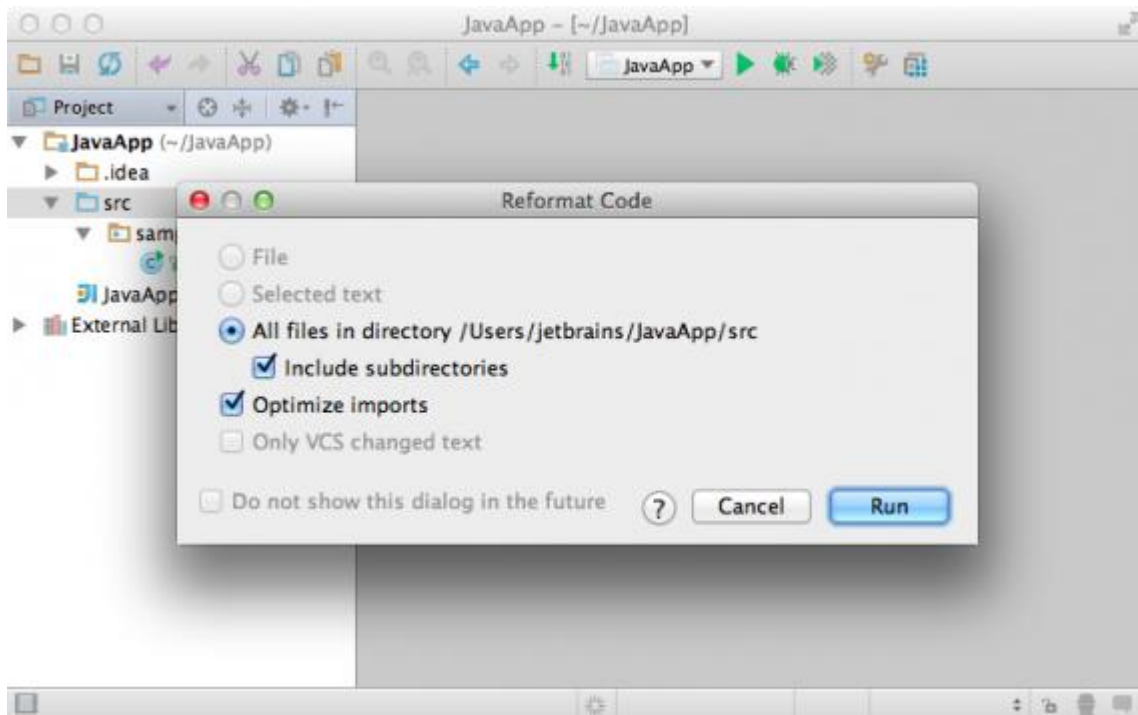


## Стиль и форматирование кода.

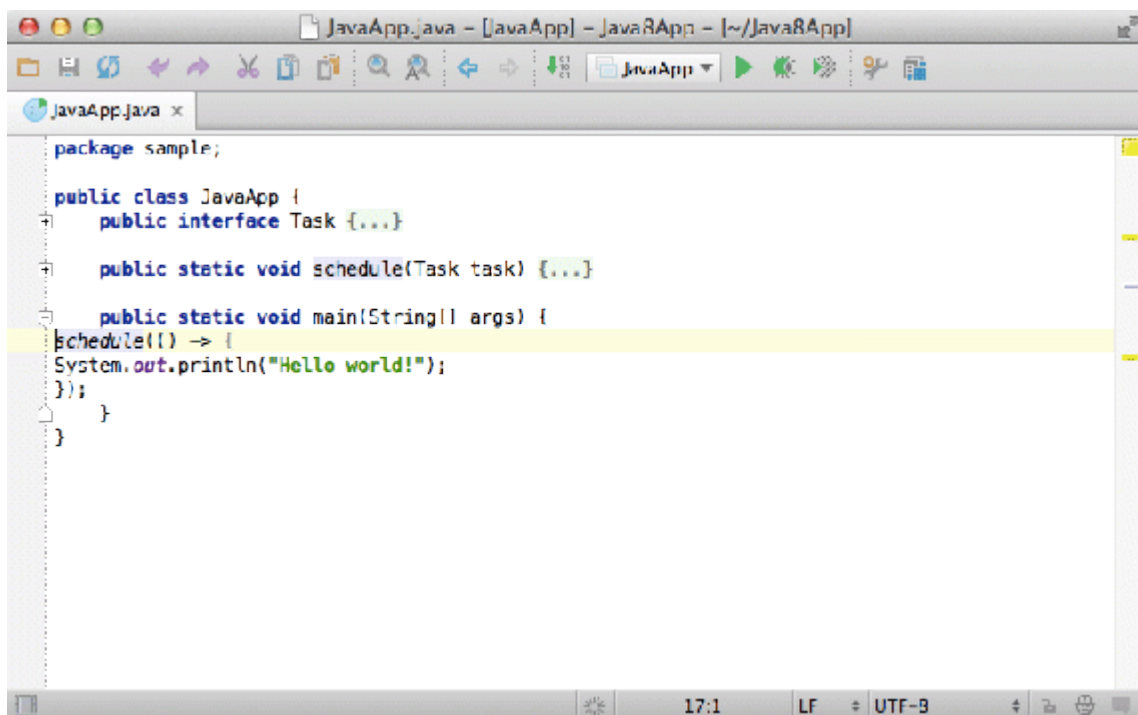
**Форматирование кода.** *IntelliJ IDEA* автоматически применяет стиль кода, настроенный при редактировании, и в большинстве случаев вам не нужно вызывать *Reformat Code* (Форматирование кода) явно. Однако, вы можете сделать это в любое время как для всего файла, так и просто для выбранного куска кода, или даже для всего каталога, просто нажав *Alt + Ctrl + L* (*Alt + Cmd + L* для *Mac*.)



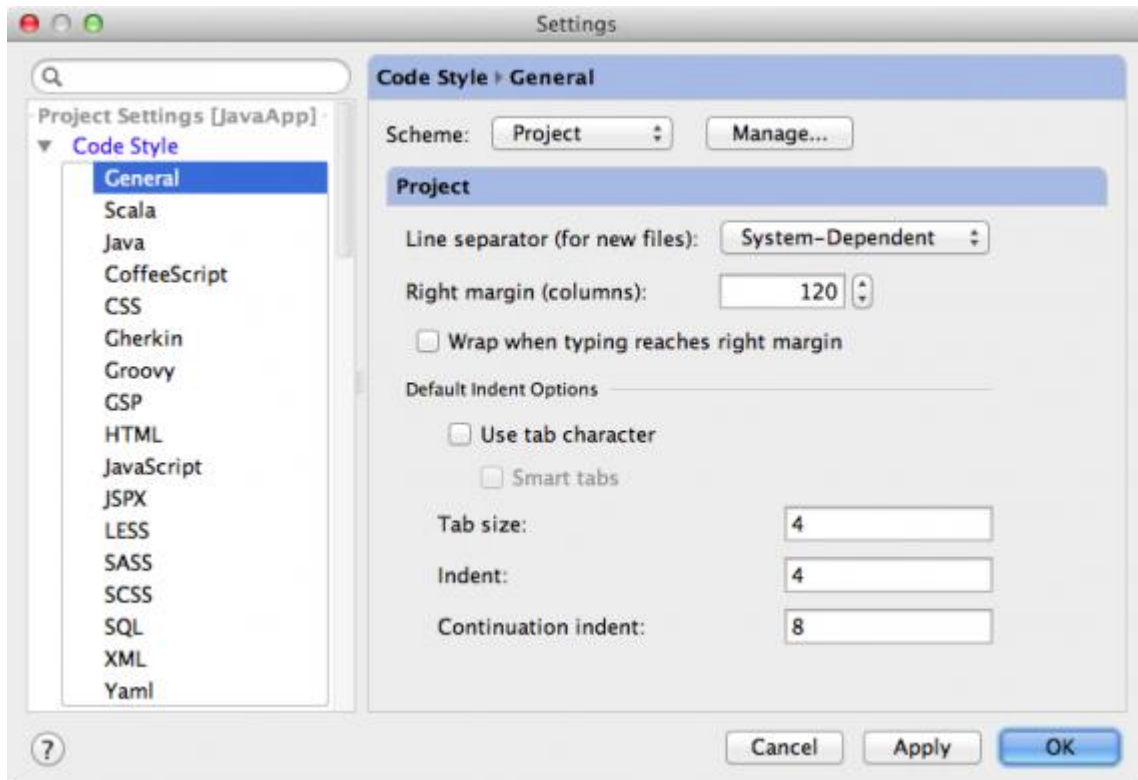
Если вы хотите применить *Reformat Code* для всех файлов в директории, используйте ту же кнопку на панели инструментов проекта.



**Автоматическое выравнивание строк.** Еще одна полезная возможность *Auto-Indent Lines*, которая помогает вам исправить отступ строки. Это действие иногда является лучшим вариантом, чем *Reformat Code*, потому что он не требует от вас выбрать что-нибудь. Просто нажмите **Alt + Ctrl + I** (**Alt + Cmd + I** для Mac), и отступы для текущей строки будут быстро приведены в порядок.



**Настройки.** *IntelliJ IDEA* позволяет вам настроить параметры стилей кода для каждого из поддерживаемых языков, либо для проекта либо для среды разработки в целом с помощью *Settings* → *Code Style*



Схемы стилей для проекта могут быть распространены среди других членов вашей команды разработчиков, с помощью системы контроля версий.

**Символ табуляции.** Последняя, но не менее важная настройка стоящая вашего внимания это *Use tab character* (использование символа табуляции). По умолчанию эта настройка выключена, и *IntelliJ IDEA* использует обычные пробелы для выравнивания строк вместо символов табуляции. Если в ваших файлах используется слишком много пробелов для выравнивания кода, вы можете оптимизировать их размер включив эту настройку (4 пробела будут заменены на 1 символ табуляции).

### Создание первой программы

Необходимо создать проект с среде *IntelliJ IDEA* (смотри выше) с главным классом **HelloWorld**.

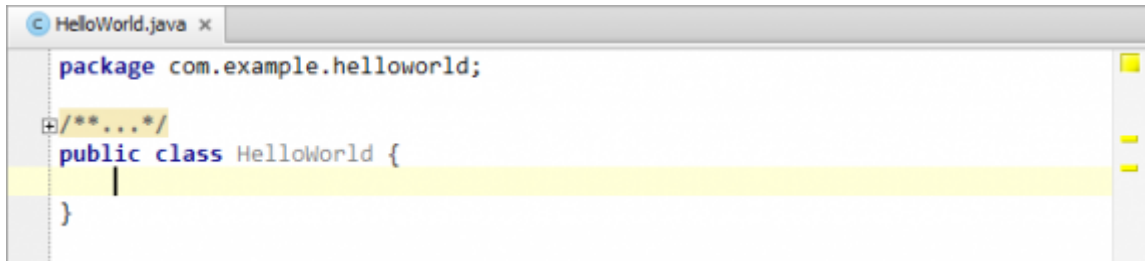
Код в конечном состоянии (как вы, наверное, знаете) будет выглядеть следующим образом:

```
package com.example.helloworld;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Объявление пакета и класса уже есть, теперь добавим недостающие пару строк.

Поместите курсор в конец текущей строки, после знака {, и нажмите ENTER, чтобы начать новую строку (На самом деле, можно сделать проще: независимо от позиции курсора, нажатие клавиш SHIFT + ENTER начинает новую строку, сохраняя предыдущие строки без изменений).



```
package com.example.helloworld;

/**...*/
public class HelloWorld {

```

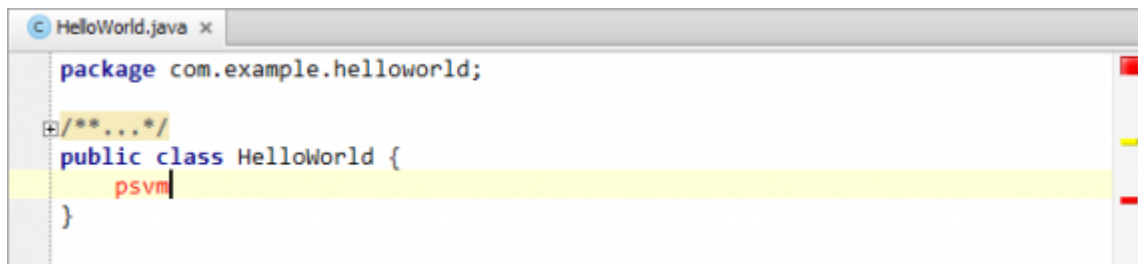
### Использование активного шаблона для метода Main().

Строку

```
public static void main(String[] args) {}
```

вполне можно и просто напечатать. Однако рекомендовал бы вам другой метод. Печатаем:

```
psvm
```

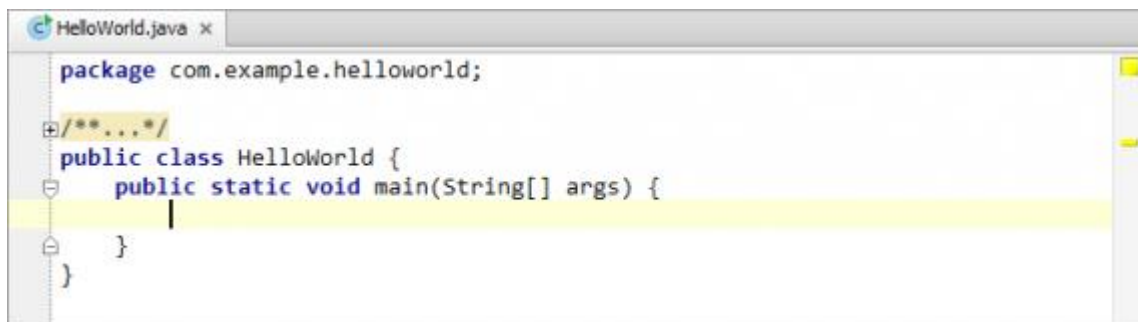


```
package com.example.helloworld;

/**...*/
public class HelloWorld {
    psvm

```

и нажимаем TAB. В результате получаем:



```
package com.example.helloworld;

/**...*/
public class HelloWorld {
    public static void main(String[] args) {

```

В данном случае мы использовали активный шаблон генерации кода объекта. Активный шаблон имеет аббревиатуру- строку, определяющую шаблон (PSVM = public static void main в этом примере) и клавишу для вставки фрагмента в код



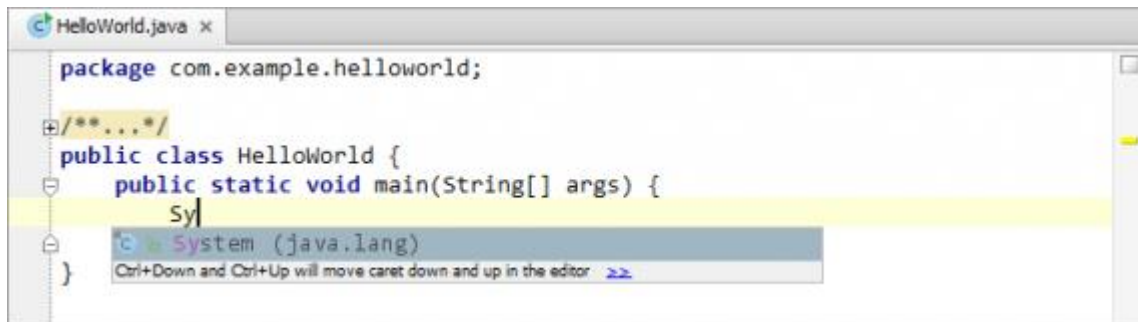
(TAB в данном случае). Дополнительную информацию можно найти в разделе Live Templates в IntelliJ IDEA Help.

### Использование автоматического завершения кода.

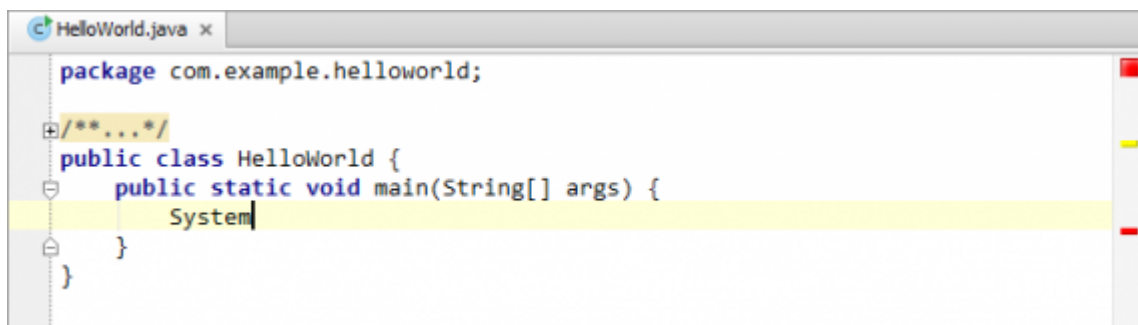
Теперь пришло время добавить оставшиеся строки кода (System.out.println («Hello, World!»);). Мы сделаем это с помощью операции автоматического завершения кода в IntelliJ IDEA. Печатаем:

Sy

Автоматическое завершение кода предлагает нам варианты:



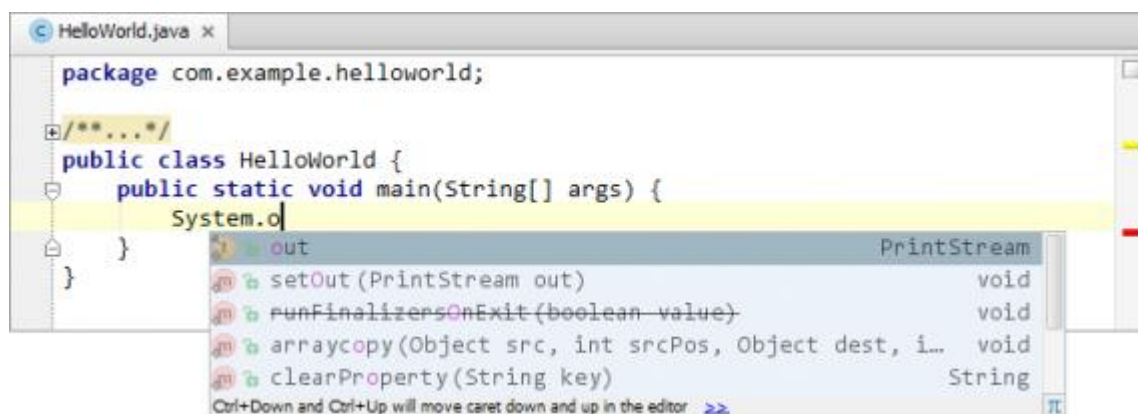
В данном случае вариант только один: **System (java.lang)**. Нажимаем ENTER, чтобы выбрать его.



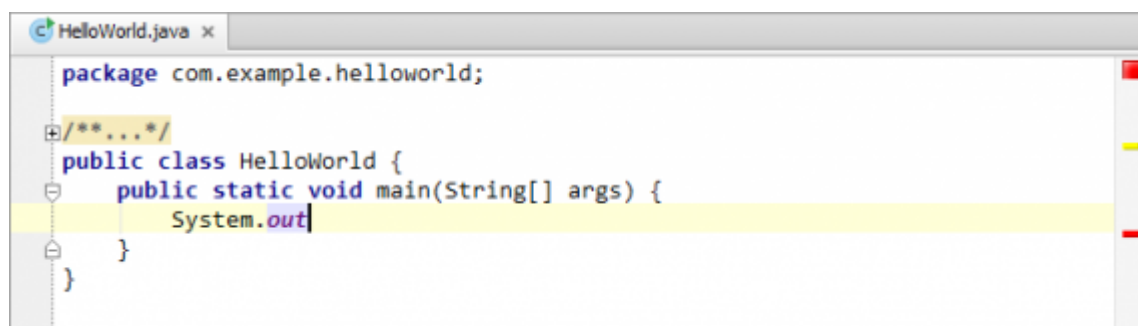
Печатаем точку и букву «о»:

.o

Функция автоматического завершения кода снова предлагает нам варианты:



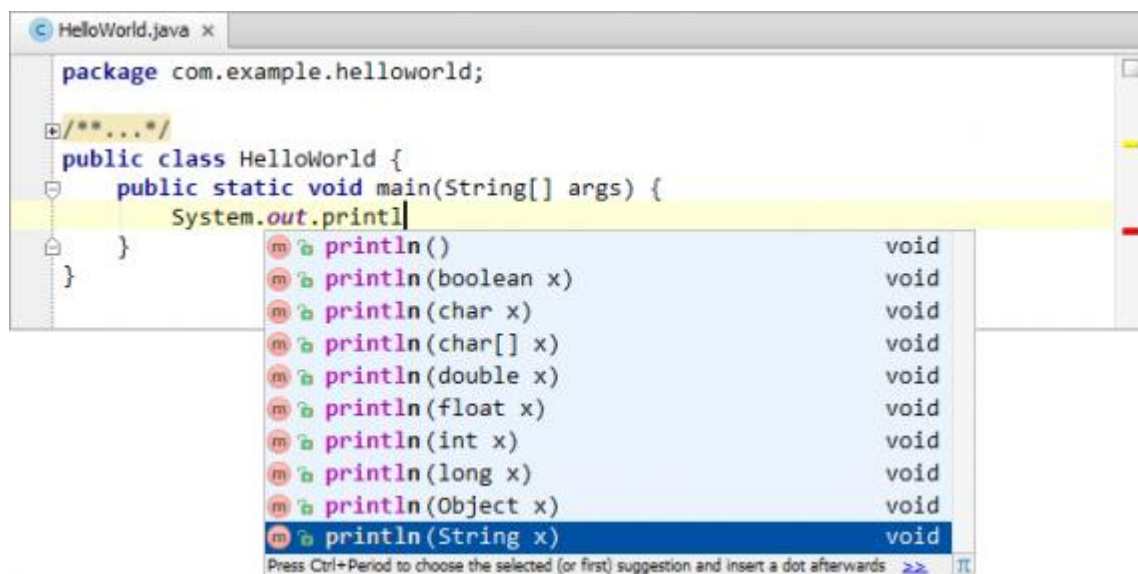
Нажимаем ENTER, чтобы выбрать out.



Печатаем:

```
.println
```

Обратите внимание, как изменяется список вариантов в процессе ввода. Метод, который мы ищем — **Println (String x)**.



Выбираем **println(String x)**. Код принимает следующий вид:



```
>HelloWorld.java x
package com.example.helloworld;

/**...*/
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println();
    }
}
```

Печатаем кавычки:

```
"
```

Как видите, вторые кавычки появляются автоматически, а курсор перемещается в место, где должен быть наш текст. Печатаем:

```
Hello, World!
```

```
HelloWorld.java x
package com.example.helloworld;

/**...*/
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Этап написания кода завершен.

### Использование активного шаблона для Println ()

К слову, мы могли бы выполнить вызов Println () с помощью активного шаблона. Аббревиатура для соответствующего шаблона- Sout. а клавиша активации- TAB. Вы можете попробовать использовать этот шаблон в качестве дополнительного упражнения. (Если вы думаете, что с вас достаточно активных шаблоны, перейдите по созданию проекта).

Удалите строку

```
System.out.println("Hello, World!");
```

Печатаем

```
sout
```

и нажимаем TAB. Строка

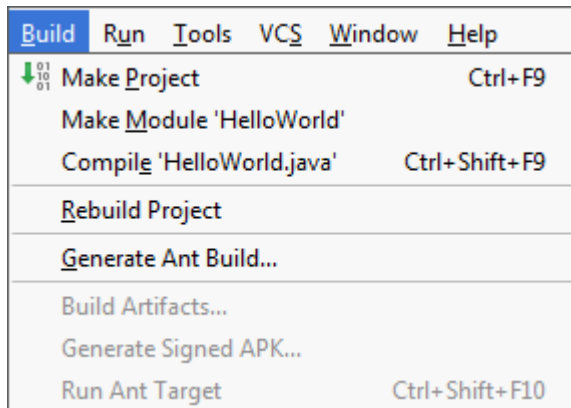
```
System.out.println();
```

добавляется автоматически, и курсор оказывается в скобках. Нам остается напечатать

```
Hello, World!
```

## Строительство проекта

Опции построения проекта или его части доступны в меню **Build**.



Многие из этих опций доступны также в контекстном меню в окне **Project** и в редакторе для HelloWorld.java.

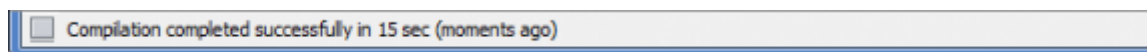
Также есть значок на панели инструментов, которая соответствует команде **Make Project** (📁).

Теперь давайте построим проект.

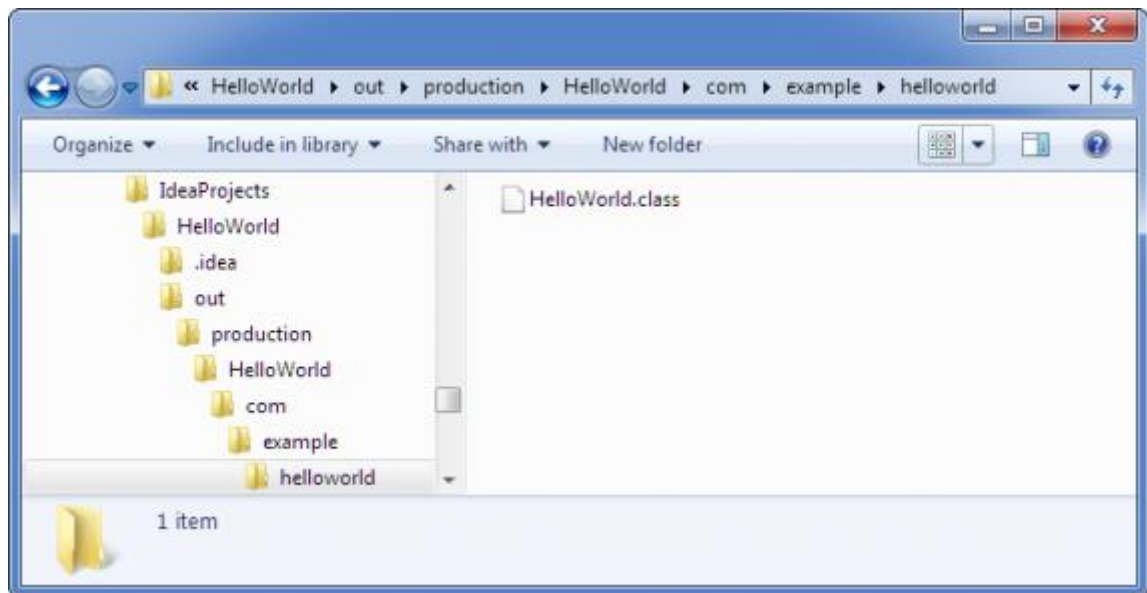
Строительство в данном конкретном случае- просто компиляция исходного файла Java в файл класса. Таким образом, любой из вариантов в меню **Build** (**Make Project**, **Make Module 'HelloWorld'**, или **Compile 'HelloWorld.java'**) могут быть использованы для этой цели.

Давайте попробуем построить проект. (Клавиатурный эквивалент для этой команды CTRL + F9. Обратите внимание, что эта ссылка отображается прямо в меню как полезный намек.)

Подождите, пока IntelliJ IDEA завершит компиляцию. Когда этот процесс будет завершен, соответствующая информация отображается в строке состояния.



Теперь, если вы перейдете в папку модуля вывода (по умолчанию это папка \out\production\, в нашем случае, и папка и называются HelloWorld), вы увидите там структуру папок для пакета com.example.helloworld и HelloWorld.class файл в папке HelloWorld.



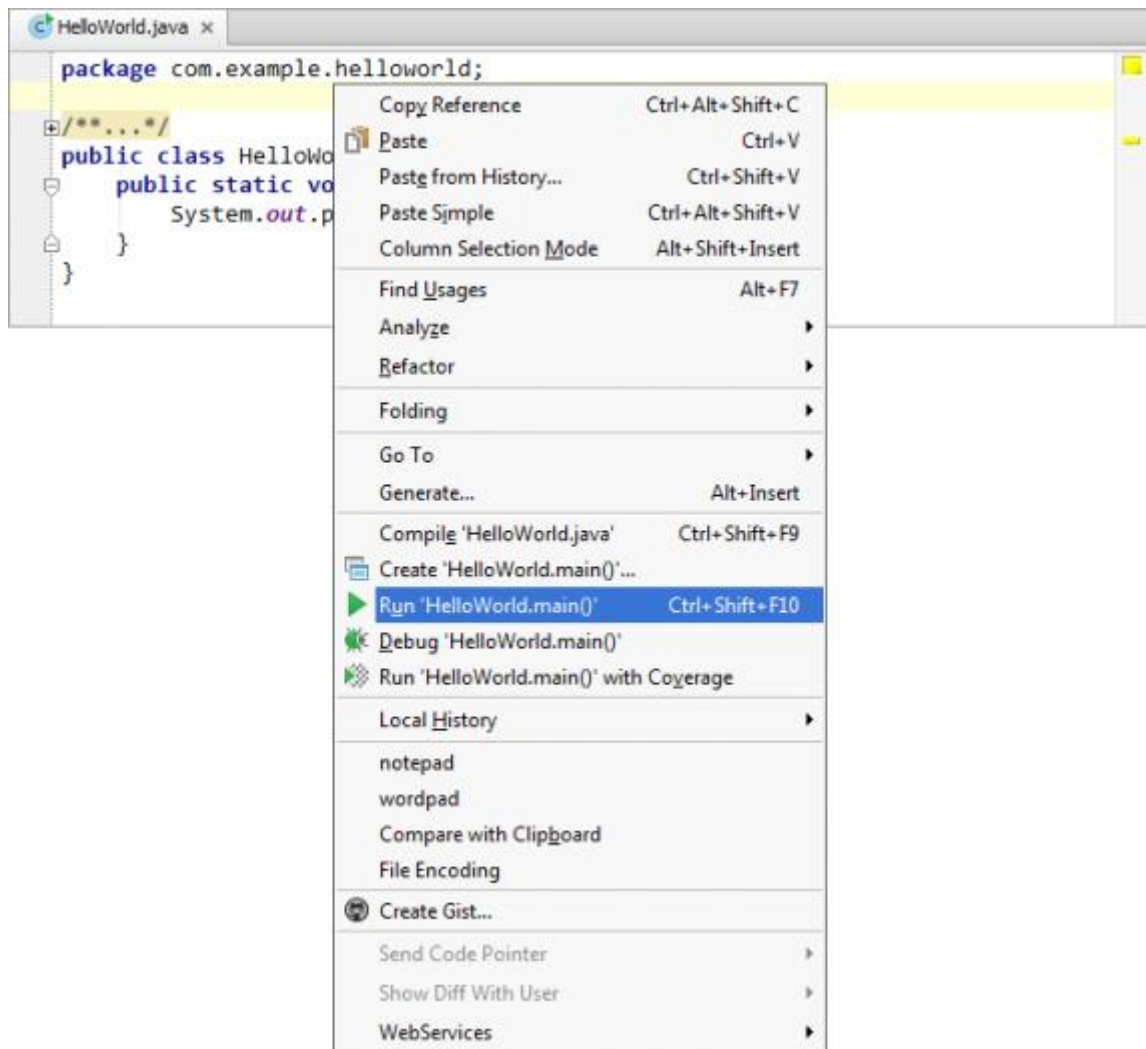
Если вы хотите разобраться в строительстве приложения лучше, обратитесь к разделам IntelliJ IDEA Help: Build Process, Compilation Types, Configuring Module Compiler Output и Configuring Project Compiler Output.

### **Запуск приложения.**

Приложение IntelliJ IDEA выполняются согласно тому, что называется конфигурацией запуска/отладки (Run/Debug). Такая конфигурация, как правило, должна быть создана до запуска приложения. (Более подробную информацию см. в разделе Running, Debugging and Testing в IntelliJ IDEA Help.)

В случае класса HelloWorld, нет необходимости создавать конфигурацию запуска и отладки заранее. Класс содержит метод `main()`. Такие классы могут быть запущены сразу, прямо из редактора. Для этой цели существует команда Run '`<ClassName>.main()`' в контекстном меню для класса.

Таким образом, чтобы запустить класс, щелкните правой кнопкой мыши где-нибудь в области редактирования и выберите Run 'HelloWorld.main ()'.



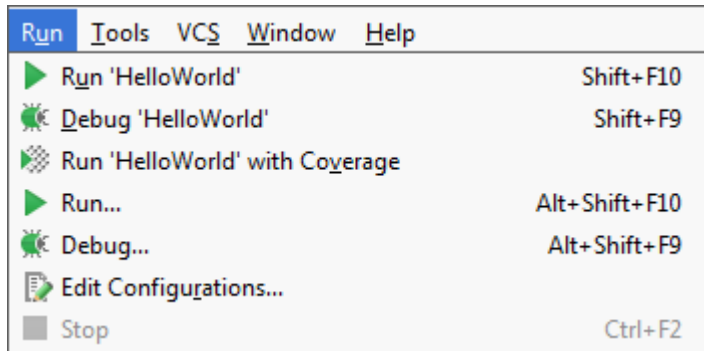
В результате выполнения команды **Run** появляется окно в нижней части экрана. Оно отвечает за отображение всех выходных данных, указанных в конфигурации команды. (Более подробную информацию см. в разделе Run Tool Window, в справке IntelliJ IDEA.)



Первая строка в окне содержит командную строку IntelliJ IDEA, используемую для запуска класса, включая все опции и аргументы. Последняя строка показывает, что процесс завершился нормально, бесконечных циклов не произошло. И, наконец, вы видите вывод программы Hello, World! между этими двумя строками.

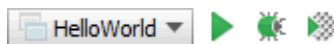
На этом этапе наше упражнение закончено. Однако, есть заключительные замечания, которые стоит сделать, связанные с запуском приложений IntelliJ IDEA:

Варианты для запуска приложений можно найти в главном меню.



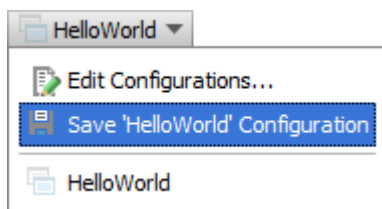
Большинство имен команд в этом меню говорят сами за себя. Опция редактирования конфигурации запуска открывает диалоговое окно для создания и редактирования конфигураций запуска. Также отметим, что сочетания клавиш (см. справа в меню) доступны для большинства команд.

На главной панели инструментов есть область, содержащая кнопки, связанные с запуском приложений. К ним относятся кнопки выбора конфигурации запуска и отладки (Run/Debug) и значки для запуска приложений в различных режимах.



Выбор конфигурации позволяет выбрать Run/Debug конфигурации, которые вы хотите использовать. Он также позволяет получить доступ к настройке Run/Debug конфигурации (**Edit Configurations**) и выполнения других задач, связанных с работой функций Run/Debug.

(В результате запуска класса HelloWorld, Run/Debug конфигурация HelloWorld была сохранена как временная. Теперь вы можете сохранить эту конфигурацию запуска (**Save Configuration «HelloWorld»**), чтобы превратить ее в постоянную.)



Варианты для запуска приложений и для работы с Run/Debug конфигурациями, в случае необходимости, также присутствуют как команды контекстного меню в окне **Project**.

## **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Когда был создан язык Java?
2. Какое ПО необходимо для компилирования и запуска java-программы?
3. Что входит в состав JDK?
4. Для чего нужна интегрированная среда разработки? Перечислите их.

## **8. Рекомендуемая литература**

1. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
2. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
3. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с

4. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
5. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. –240с.
6. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Создание линейных программ»

Минск  
2018



## Лабораторная работа № 2

### Тема работы: «Создание линейных программ»

#### 1. Цель работы

Научить создавать линейные программы. Закрепить применение арифметических блоков, и ввода-вывода данных.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

	Функция
	$y = \frac{5x + \sin x^2}{2x + \operatorname{tg} x} +  \sin x^2 $
	$y = \frac{\log_3  x - 10 }{2x - 7} - \cos(3x + 5)$
	$y = \frac{\cos(x_1 + 5) + x_2}{3x_1 + 6x_2} *  \sin x_1^2 $
	$y = \frac{\cos(3x + 5) + 5^x}{4.35x} +  \cos x^2  + \frac{5}{x}$
	$y = \frac{3x_1 + 2^{x_2}}{\sin(x_1 + x_2)} + \frac{5}{ \sin x_2 } + 6$
	$y = \frac{\sin^2(x + 3.5) + \lg x}{3x + e^{x+1.35}}$
	$y = \frac{x_1 + 5.5x_2}{\cos x_1 + \ln x_2} + \operatorname{tg}(x_1 + 4.3)$
	$y = \frac{3^{x-1.4} + e^x}{4.5 + x} + \operatorname{tg} 3x$
	$y = \frac{x_1^{x_2} + 3.5 \operatorname{tg} x_1}{3x_1 + 5x_2} + \frac{5x_1}{x_2 + 6} + 5$
0	$y = \frac{\cos^3(x + 2.5) + 1.5x}{x^2 + 3.5} + \frac{3x^2}{8x} + 8.5$

	Функция
1	$y = \frac{\ln(x+1.3)+5}{1.35x^2+6} + \sqrt{\frac{x+1.3}{(35x)^2+6}}$
2	$y = \frac{2x_1+1.4^{x_2}}{\operatorname{tg} x_1+2x_2} + \sqrt[3]{6x_1+8^{x_2}}$
3	$y = \frac{3x+\operatorname{tg} x}{2.36x+6} + 2x+1.4^x$
4	$y = \frac{e^{x-1.3}+\sin x}{x+3.5} + 2x+1.4\sin x$
5	$y = \frac{x+3\cos(x^2+1.5)}{\operatorname{tg} x+4.56} + \sin x \frac{2+x}{\sqrt{5^x+56x}}$
6	$y = \frac{x+3\cos(x^2+1.5)}{\operatorname{tg} x+4.56} + \sin(2x) + \cos(x^2+5)$
7	$y = \frac{\cos^2(x_1+1.3)+x_2}{2x_1+e^{x_2}} + \cos(x_2^2+1.5) + \frac{x_2}{\cos(x_1^2+1.5)}$
8	$y = \frac{5x_1+1.3^{x_2}}{\cos(x_1+x_2)} + \cos(x_2^2+1.5) + \frac{5+x_2}{\sqrt{x_1}}$
9	$t = \frac{2\cos\left(x - \frac{\pi}{6}\right)}{0.5 + \sin^2 y} \left(1 + \frac{z^2}{3 - z^2/5}\right).$
0	$u = \frac{\sqrt[3]{8+ x-y ^2+1}}{x^2+y^2+2} - e^{ x-y } \left(g^2z+1\right)^x.$
1	$v = \frac{1+\sin^2\left(x+y\right)}{\left x-\frac{2y}{1+x^2y^2}\right } x^{ y } + \cos^2\left(\operatorname{arctg}\frac{1}{z}\right).$

	Функция
2	$w =  \cos x - \cos y ^{+2 \sin^2 y} \left( 1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4} \right).$
3	$\alpha = \ln \left( y^{-\sqrt{ x }} \right) \left( x - \frac{y}{2} \right) + \sin^2 \arctg \sqrt{x}.$
4	$\beta = \sqrt{10} \sqrt{x + x^{y+2}} \left( \arcsin^2 z -  x - y  \right).$
5	$\gamma = 5 \arctg \sqrt{x} - \frac{1}{4} \arccos \sqrt{\frac{x + 3 x - y  + x^2}{ x - y z + x^2}}.$
6	$\varphi = \frac{e^{ x-y }  x - y ^{x+y}}{\arctg \sqrt{x} + \arctg \sqrt{y}} + \sqrt[3]{x^6 + \ln^2 y}.$
7	$\psi = \left  \frac{y}{x^x} - \sqrt[3]{\frac{y}{x}} \right  + \sqrt{y - x} \frac{\cos y - \frac{z}{\sqrt{y - x}}}{1 + \sqrt{y - x}}.$
8	$a = 2^{-x} \sqrt{x + \sqrt[4]{ y }} \sqrt[3]{e^{x-1/\sin z}}.$
9	$b = \frac{y^{\sqrt[3]{ x }} + \cos^3 \sqrt{y} \left(  x - y  \left( 1 + \frac{\sin^2 z}{\sqrt{x + y}} \right) \right)}{e^{ x-y } + \frac{x}{2}}.$
0	$c = 2^{\sqrt{x}} + \sqrt{x}^y - \frac{y \left( \arctg(z) - \frac{\pi}{6} \right)}{ x  + \frac{1}{y^2 + 1}}.$
1	$f = \frac{\sqrt[4]{y + \sqrt[3]{x-1}}}{ x - y  \left( \sin^2 z + \tg z \right)}.$

	Функция
2	$g = \frac{y^{x+1}}{\sqrt[3]{ y-2 } + 3} + \frac{x + \frac{y}{2}}{2 x+y } \left( \sqrt[3]{+1} \right)^{1/\sin z}$
3	$h = \frac{x^{y+1} + e^{y-1}}{1 + x y - \operatorname{tg} z } \left( \sqrt[3]{+ y-x } \right)^{\frac{ y-x ^2}{2}} - \frac{ y-x ^3}{3}$

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### 4. Основные теоретические сведения

#### Комментарии

В программу можно вставить комментарии, которые не учитывает компилятор. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий один или несколько операторов, пометив их символами комментария, как говорят программисты, "закомментеровав" их. Комментарии вводятся таким образом:

- ✓ за двумя наклонными чертами подряд // ( без пробела между ними) начинается комментарий, продолжающийся до конца строки;
- ✓ за наклонной чертой и звездочкой /\* начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты \*/ (без пробелов между этими знаками).

Комментарии очень удобны для чтения и понимания кода, они превращают программу в документ, описывающий ее действия. Программу с хорошими комментариями называют *самодокументированной*. Поэтому в *Java* введены комментарии третьего типа, а в состав JDK — программа *javadoc*, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними: за наклонной чертой и двумя звездочками подряд /\*\* (без пробелов) начинается комментарий, который может занимать несколько строк до звездочки (одной) и наклонной черты \*/ и обрабатываться программой *javadoc*. В такой комментарий можно вставить указания программе *javadoc*, которые начинаются с символа @.

Именно так создается документация к JDK.  
Добавим комментарии к примеру.

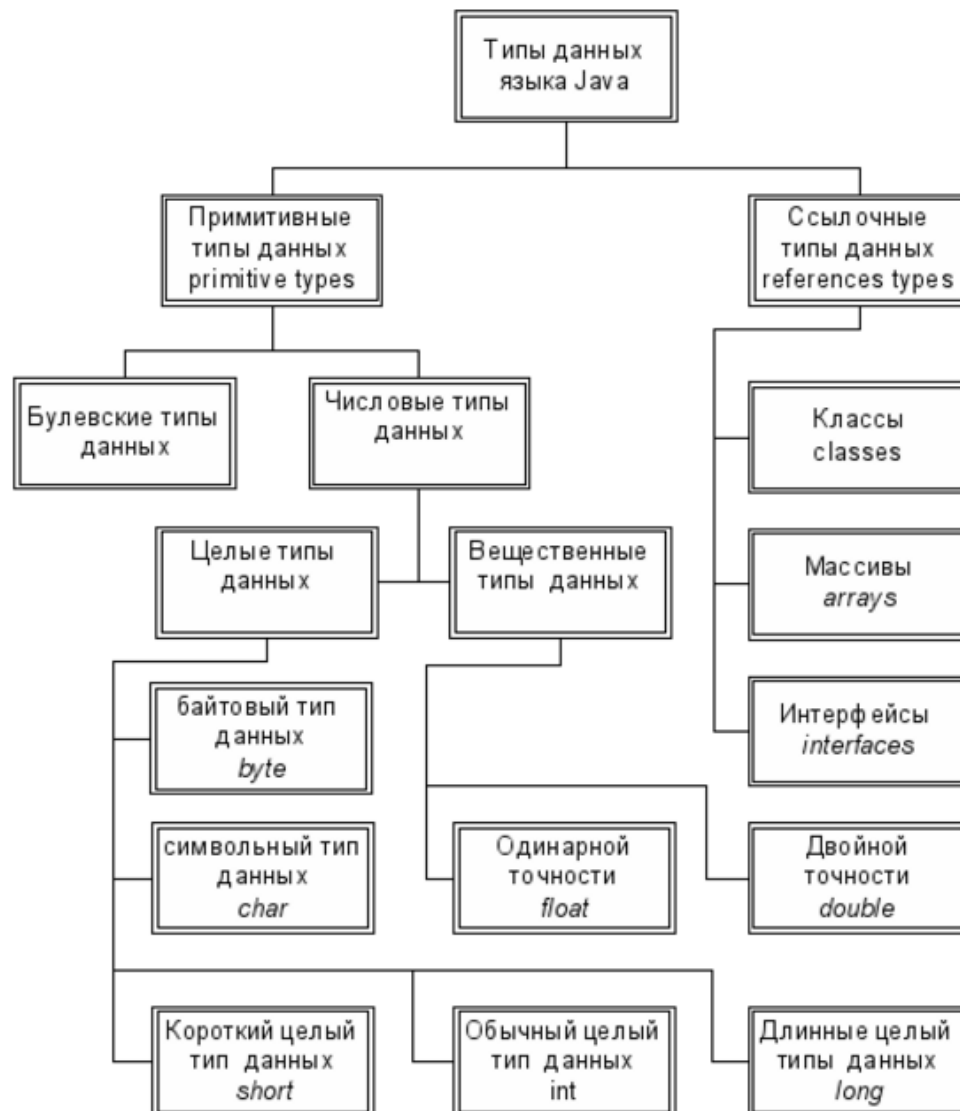
**Листинг 2.1.** Первая программа с комментариями

```
/**
 * Разъяснение содержания и особенностей программы...
 * @author Имя Фамилия (автора)
 * @version 1.0 (это версия программы)
 */
class HelloWorld2
{
    // HelloWorld - это только имя
    // Следующий метод начинает выполнение программы
    public static void main(String[] args){
        // args не используются
        /* Следующий метод выводит свой аргумент на
           экран дисплея*/
        System.out.println("Hello, Aero-space University
                               !");
        // Следующий вызов закомментирован, метод не будет
        //выполняться
        // System.out.println("Farewell, 20th Century!");
    }
}
```

Звездочки в начале строк не имеют никакого значения, они написаны просто для выделения комментария. Пример, конечно, перегружен пояснениями (это плохой стиль), здесь просто показаны разные формы комментариев.

## Примитивные типы данных и операции

Все типы исходных данных, встроенные в язык Java, разделяются на две группы: *примитивные типы* (primitive types) и *ссылочные типы* (reference types).



Ссылочные типы разделяют на *массивы* (arrays), *классы* (classes) и *интерфейсы* (interfaces).

Примитивных типов всего восемь. Их можно разделить на *логические* (иногда говорят *булевые*) тип boolean и *числовые* (numeric).

К числовым типам относятся *целые* и *вещественные* (floating-point) типы.

Целых типов пять: byte, short, int, long, char.

Символы можно использовать везде, где используется тип int. Например, их можно использовать в арифметических вычислениях, скажем, можно написать 2 "Ж", тогда к двойке будет прибавляться кодировка Unicode '\u0416' буквы 'Ж'. В десятичной форме это число 1046 и в результате сложения получим 1048.

Напомним, что в записи 2 "Ж" плюс понимается как сцепление строк, двойка будет преобразована в строку, в результате получится строка "2ж".

Вещественных типов два: float и double.

### **Целочисленные типы в JAVA**

Тип	Размер (бит)	Диапазон
byte	8 бит	от -128 до 127
short	16 бит	от -32768 до 32767
char	16 бит	без знаковое целое число, представляющее собой символ UTF-16 (буквы и цифры)
int	32 бит	от -2147483648 до 2147483647
long	64 бит	от -9223372036854775808 до 9223372036854775807

### Типы с плавающей точкой в JAVA

Тип	Размер (бит)	Диапазон
float	32	от -1.4e-45f до 3.4e+38
double	64	от -4.9e-324 до 1.7e+308

### Имена

*Имена* (names) переменных, классов, методов и других объектов могут быть простыми (общее название — *идентификаторы* (idenifiers)) и *составными* (qualified names). Идентификаторы в Java состояются из так называемых *букв Java* (Java letters) и арабских цифр 0 — 9, причем первым символом идентификатора не может быть цифра. (Действительно, как понять запись 2e3: как число 2000,0 или как имя переменной?) В число букв *Java* обязательно входят прописные и строчные латинские буквы, знак доллара \$ и знак подчеркивания \_, а так же символы национальных алфавитов.

Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Вот примеры правильных идентификаторов:

```
a1      my_var    var3_5    _var    veryLongVarName
aName   theName  a2Vh36kBnMt456dX
```

В именах лучше не использовать строчную букву l , которую легко спутать с единицей, и букву o, которую легко принять за нуль.

Служебные слова Java, такие как class, void, static, зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

*Составное имя* (qualified name) — это несколько идентификаторов, разделенных точками без пробелов, например, уже встречавшееся имя System.out.println.

## Операции над целыми типами

Арифметические операции:

- ✓ сложение (плюс);
- ✓ вычитание - (дефис);
- ✓ умножение \* (звездочка);
- ✓ деление / (наклонная черта — слэш);
- ✓ взятие остатка от деления (деление по модулю) % (процент);
- ✓ инкремент (увеличение на единицу) ++;
- ✓ декремент (уменьшение на единицу) --.

Между сдвоенными плюсами и минусами нельзя оставлять пробелы. Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает опять целое (так называемое "*целое деление*"), например,  $5/2$  даст в результате 2, а не 2.5, а  $5/(-3)$  даст -1. Дробная часть числа попросту отбрасывается, происходит усечение частного, т.к. в Java принято целочисленное деление. Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать  $5/2.0$  или  $5.0/2$  или  $5.0/2.0$  и получим 2.5 как результат деления вещественных чисел.

Операция *деление по модулю* определяется так:

$$a \% b = a - (a / b) * b$$

Например,  $5\%2$  даст в результате 1, а  $5\%(-3)$  даст, 2, т.к.

$5 = (-3)*(-1) 2$ , но  $(-5)\%3$  даст -2, поскольку  $-5 = 3 * (-1) - 2$ .

Операции *инкремент* и *декремент* означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать 5 или (a b). Например, после приведенных выше описаний i даст -99, а j -- даст 99. Операции можно записать и перед переменной: i, — j. Разница проявится только в выражениях: при первой форме записи (*постфиксной*) в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи (*префиксной*) сначала изменится переменная, и ее новое значение будет участвовать в выражении.

Например, после приведенных выше описаний, (k) 5 даст в результате 10004, а переменная k примет значение 10000. Но в той же исходной ситуации (k) 5 даст 10005, а переменная k станет равной 10000.

## Операции присваивания

*Простая операция присваивания* записывается знаком равенства =, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной:



$x = 3.5, y = 2 * (x - 0.567) / (x^2), b = x < y, bb = x > y \& \& b$

Операция присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части.

Операция присваивания имеет еще одно побочное действие: переменная, стоящая слева, получает приведенное значение правой части, старое ее значение теряется.

В операции присваивания левая и правая части неравноправны, нельзя написать  $3.5 = x$ . После операции  $x = y$  изменится переменная  $x$ , став равной  $y$ , а после  $y = x$  изменится  $y$ .

Кроме простой операции присваивания есть еще 11 *составных* операций присваивания (compound assignment operators):

$=, -=, *=, /=, \%, \&, |=, ^=, <<=, >>=, >>>=$

Все составные операции действуют по одной схеме:

$x \text{ op} = a$  эквивалентно  $x = (\text{тип } x), \text{ т. е. } (x \text{ op } a)$ .

Предположим, что переменная `ind` типа `short` определена со значением 1. Присваивание `ind = 7.8` даст в результате число 8, то же значение получит и переменная `ind`. Эта операция эквивалентна простой операции присваивания `ind = (short)(ind + 7.8)`.

Перед присваиванием, при необходимости, автоматически производится приведение типа. Поэтому:

```
byte b = 1;  
b = b + 10; // Ошибка!  
b = 10; // Правильно!
```

Перед сложением `b + 10` происходит повышение `b` до типа `int`, результат сложения тоже будет типа `int` и, в первом случае, не может быть присвоен переменной `b` без явного приведения типа. Во втором случае перед присваиванием произойдет сужение результата сложения до типа `byte`.

### Блок

Блок может содержать в себе нуль или несколько операторов с целью их использования как одного оператора в тех местах, где по правилам языка можно записать только один оператор. Например, `{x=5; y=?;}`. Можно записать и пустой блок `{ }`.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

### Библиотека Math

Класс **Math** содержит методы для выполнения основных числовых операций, таких как нахождение экспоненты, логарифма, квадратного корня и т. д.

Класс содержит две константы типа double: **E** и **PI**. Все методы в классе Math статичны.

Тип	Метод	Описание
double	abs(double a)	Возвращает абсолютное значение (модуль) числа типа double.
float	abs(float a)	Возвращает абсолютное значение (модуль) числа типа float .
int	abs(int a)	Возвращает абсолютное значение (модуль) числа типа int.
long	abs(long a)	Возвращает абсолютное значение (модуль) числа типа long.
double	acos(double a)	Возвращает арккосинус значения. Возвращенный угол находится в диапазоне от 0 до $\pi$ .
double	asin(double a)	Возвращает арксинус значения. Возвращенный угол в диапазоне от $-\pi/2$ до $\pi/2$ .
double	atan(double a)	Возвращает арктангенс значения. Возвращенный угол в диапазоне от $-\pi/2$ до $\pi/2$ .
double	cbrt(double a)	Возвращает кубический корень аргумента.
double	ceil(double a)	Возвращает наименьшее целое число, которое больше аргумента.
double	copySign(double	Возвращает аргумент с тем же знаком, что у

Тип	Метод	Описание
	magnitude, double sign)	второго аргумента.
double	copySign(float magnitude, float sign)	Возвращает аргумент с тем же знаком, что у второго аргумента.
double	cos(double a)	Возвращает косинус аргумента.
double	cosh(double x)	Возвращает гиперболический косинус аргумента.
int	decrementExact(int a)	Возвращает значение аргумента уменьшенное на единицу.
long	decrementExact(long a)	Возвращает значение аргумента уменьшенное на единицу.
double	exp(double a)	Возвращает экспоненту аргумента.
double	floor(double a)	Возвращает наибольшее целое число, которое меньше или равно аргументу.
double	hypot(double x, double y)	Возвращает длину гипотенузы ( $\sqrt{x^2 + y^2}$ ).
double	IEEERemainder(double f1, double f2)	Возвращает остаток от деления f1 на f2.
int	incrementExact(int a)	Возвращает значение аргумента увеличенное на единицу.
long	incrementExact(long a)	Возвращает значение аргумента увеличенное на единицу.

Тип	Метод	Описание
double	log(double a)	Возвращает натуральный логарифм (по основанию e).
double	log10(double a)	Возвращает логарифм по основанию 10.
double	max(double a, double b)	Возвращает больший из аргументов.
float	max(float a, float b)	Возвращает больший из аргументов.
int	max(int a, int b)	Возвращает больший из аргументов.
long	max(long a, long b)	Возвращает больший из аргументов.
double	min(double a, double b)	Возвращает меньший из аргументов.
float	min(float a, float b)	Возвращает меньший из аргументов.
int	min(int a, int b)	Возвращает меньший из аргументов.
long	min(long a, long b)	Возвращает меньший из аргументов.
int	multiplyExact(int x, int y)	Возвращает произведение аргументов ( $x*y$ ).
long	multiplyExact(long x, long y)	Возвращает произведение аргументов ( $x*y$ ).
int	negateExact(int a)	Возвращает отрицательное значение аргумента.
long	negateExact(long a)	Возвращает отрицательное значение

Тип	Метод	Описание
		аргумента.
double	pow(double a, double b)	Возвращает значение первого аргумента, возведенное в степень второго аргумента.
double	random()	Возвращает случайное число от 0.0 (включительно) до 1 (не включительно).
double	rint(double a)	Возвращает округленное значение аргумента.
int	round(double a)	Возвращает округленное значение аргумента.
double	signum(double a)	Возвращает знак аргумента.
float	signum(float a)	Возвращает знак аргумента.
double	sin(double a)	Возвращает синус аргумента.
double	sinh(double a)	Возвращает гиперболический синус аргумента.
double	sqrt(double a)	Возвращает квадратный корень аргумента.
int	subtractExact(int x, int y)	Возвращает разность аргументов (x-y).
long	subtractExact(long x, long y)	Возвращает разность аргументов (x-y).
double	tan(double a)	Возвращает тангенс аргумента.
double	tanh(double a)	Возвращает гиперболический тангенс аргумента.

Тип	Метод	Описание
double	toDegrees()	Преобразует радианы в градусы.
int	toIntExact(long value)	Преобразует аргумент типа long в int.
double	toRadians()	Преобразует градусы в радианы.

**abs(double a)** — возвращает абсолютное значение (модуль) числа типа double.

Например необходимо найти наибольшее из двух чисел

```
double d1 = 5.0;
```

```
double d2 = 3.0;
```

```
double max = Math.max(d1, d2); //равен 5.0
```

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

## 7. Контрольные вопросы и задания

1. Перечислите примитивные типы данных.
2. Почему существует несколько видов целых чисел?
3. Блок кода – это?

4. Назначение применения комментариев.
5. Какие бывают виды комментариев?

## **8. Рекомендуемая литература**

7. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
8. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
9. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
- 10.Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
- 11.Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. –240с.
- 12.Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.



Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ с использованием управляющих инструкций»

Минск  
2018

## Лабораторная работа № 3

### Тема работы: «Разработка программ с использованием управляющих инструкций»

#### 1. Цель работы

Закрепить умение работы с управляющими конструкциями.

#### 2. Задание

Три задания обязательны для выполнения. Задачи третьего задания необходимо решить двумя способами: с помощью циклов **while** и **do..while**.

Номер варианта соответствует вашему номеру по списку.

Примечание: найти сумму бесконечного ряда с заданной степенью точности можно одним из двух способов:

- ✓ если бесконечный ряд убывающий, то необходимо производить вычисления до тех пор, пока очередной член ряда больше введенной точности;
- ✓ если бесконечный ряд неубывающий, то необходимо производить вычисления до тех пор, пока разность двух соседних членов по модулю больше введенной точности.

#### Задания 1 по теме if

1. Введите два положительных числа и покажите, что среднее арифметическое этих чисел не меньше их среднего геометрического.
2. Введите два положительных числа и покажите, что среднее геометрическое этих чисел не меньше их среднего гармонического.
3. Введите три числа, найдите наименьшее отношение максимального к минимальному чисел.
4. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник равнобедренным.
5. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник равносторонним.
6. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник разносторонним.
7. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник остроугольным.
8. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник тупоугольным.
9. Даны в градусах величины двух углов треугольника. Определите, является ли данный треугольник прямоугольным.
10. Даны длины сторон треугольника. Определите, является ли данный треугольник прямоугольным.
11. Даны длины сторон треугольника. Определите, является ли данный

- треугольник равнобедренным.
12. Даны длины сторон треугольника. Определите, является ли данный треугольник равносторонним.
  13. Даны длины сторон треугольника. Определите, является ли данный треугольник разносторонним.
  14. Даны длины трех отрезков. Определите, можно ли из этих отрезков сложить треугольник.
  15. Даны радиус круга и сторона квадрата. Проверьте, пройдет ли квадрат в круг?
  16. Даны радиус круга и сторона квадрата. Проверьте, пройдет ли круг в квадрат?
  17. Два треугольника заданы координатами своих вершин. Определите радиус, описанной окружности, какого треугольника больше.
  18. Два треугольника заданы координатами своих вершин. Определите радиус, вписанной окружности, какого треугольника больше.
  19. Два треугольника заданы координатами своих вершин. Определите, можно ли из этих треугольников составить прямоугольник.
  20. Найдите наибольшее значение из трех  $f(1)$ ,  $f(2)$  и  $f(3)$ , где  $f(x) = \sin(5x)$ .
  21. Прямоугольник задан координатами двух противоположных своих вершин. Определите, принадлежит ли точка с заданными координатами области прямоугольника, если стороны прямоугольника параллельны осям координат.
  22. Прямоугольник задан координатами своих вершин. Определите, принадлежит ли окружность с заданным радиусом и координатами центра области прямоугольника.
  23. Треугольник задан координатами вершин. Определите находится ли отрезок внутри данного треугольника.
  24. Треугольник задан координатами вершин. Определите, принадлежит ли точка с заданными координатами области треугольника?
  25. Треугольник задан координатами своих вершин. Определите величину в градусах большего угла треугольника.
  26. Треугольник задан координатами своих вершин. Определите величину в градусах меньшего угла треугольника.
  27. Треугольник задан координатами своих вершин. Определите длину большей стороны треугольника.
  28. Треугольник задан координатами своих вершин. Определите длину большей медианы треугольника.
  29. Треугольник задан координатами своих вершин. Определите длину меньшей стороны треугольника.
  30. Треугольник задан координатами своих вершин. Определите длину меньшей высоты треугольника.

## Задание 2 по теме for

1. Напечатайте таблицу стоимости порций сыра весом 50, 100, 150, ..., 1000 г (цена 1 кг 280 руб.)
2. Начав тренировки, спортсмен в первый день пробежал 10 км. Каждый следующий день он увеличивал дневную дистанцию на 10 % от дистанции предыдущего дня. Какой суммарный путь пробежит спортсмен за  $N$  дней?
3. Определите суммарный объем (в литрах) 12-ти вложенных друг в друга шаров со стенками 5 мм. Внутренний диаметр внутреннего шара равен 10 см. Считайте, что шары вкладываются друг в друга без зазоров.
4. Вычислите сумму  $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$ .
5. Пусть  $n$  – натуральное число и пусть  $n!!$  означает  $1 \cdot 3 \cdot \dots \cdot n$  для нечетного  $n$  и  $2 \cdot 4 \cdot \dots \cdot n$  для четного  $n$ . Для заданного натурального  $n$  вычислите  $n!!$ .
6. Вычислите сумму  $\sum_{i=1}^n \frac{1}{i!}$ .
7. Найдите  $n$ -й член ряда Фибоначчи, элементы которого вычисляются по формулам:  $a_1 = a_2 = 1$ ;  $a_i = a_{i-1} + a_{i-2}$ , ( $i > 2$ ).  

П р и м е ч а н и е. Для нахождения членов ряда используйте только две переменные  $a$  и  $b$ .
8. Найдите сумму  $n$ -членов ряда Фибоначчи, используя также только две переменные для нахождения суммы ряда.
9. Выведите на экран таблицу значений:  $1, 1 + h, 1 + 2h, \dots, 1 + nh$ .
10. Проверьте формулу  $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$ .
11. Проверьте формулу  $1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(n+1)}{6}$ .
12. Вычислите сумму  $\sum_{i=1}^n \frac{1}{2^i}$ .
13. Вычислите число сочетаний из  $n$  по  $m$   $C_n^m = \frac{n!}{m!(n-m)!}$ .
14. Вычислите число размещений из  $n$  по  $m$   

$$A_n^m = \frac{n!}{(n-m)!} = n(n-1) \dots (n-m+1)$$
15. Вычислите сумму  $\sum_{i=1}^n \frac{1}{i^3}$ .
16. Вычислите сумму  $\sum_{i=1}^n (n-1)^{2^i}$ .
17. Вычислите сумму  $\sum_{i=1}^n \frac{(n-1)^{2^i}}{2^i}$ .
18. Вычислите произведение  $P = 1 \cdot 3 \cdot 5 \cdot 7 \cdot \dots \cdot (2n + 1)$  для заданного  $n$ .
19. Вычислите произведение  $P = 2 \cdot 4 \cdot 6 \cdot 8 \cdot \dots \cdot 2n$  для заданного  $n$ .
20. Вычислите сумму  $S = 1 \cdot 3 + 3 \cdot 5 + 5 \cdot 7 + \dots + (2n - 1)(2n + 1)$  для заданного  $n$ .

21. Вычислите произведение  $S = (1 + 3) \cdot (3 + 5) \cdot \dots \cdot ((2n - 1) + (2n + 1))$  для заданного  $n$ .

22. Составьте таблицу умножения для заданного числа  $N$ , которая содержит результаты умножения  $1 \cdot N, 2 \cdot N, \dots, N \cdot N$ .

23. Проверьте формулу

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{(n+1)^2}{4}.$$

24. Вычислите сумму  $A = \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \dots + \frac{1}{(n-1) \cdot n}$ .

25. Вычислите сумму

$$A = \frac{1}{1 \cdot 4} + \frac{1}{4 \cdot 7} + \frac{1}{7 \cdot 10} + \dots + \frac{1}{(3 \cdot (n-1) + 1) \cdot (3 \cdot n + 1)}.$$

26. Вычислите сумму

$$A = \frac{1}{1 \cdot 3 \cdot 5} + \frac{1}{3 \cdot 5 \cdot 7} + \dots + \frac{1}{(2n-1) \cdot (2n+1) \cdot (2n+3)}.$$

27. Вычислите сумму

$$A = \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{2 \cdot 3 \cdot 4} + \dots + \frac{1}{n \cdot (n+1) \cdot (n+2)}.$$

28. Вычислите произведение  $\left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{3}\right) \dots \left(1 - \frac{1}{n+1}\right)$ .

29. Вычислите сумму

$$A = \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \frac{1}{2 \cdot 3 \cdot 4 \cdot 5} + \dots + \frac{1}{n \cdot (n+1) \cdot (n+2) \cdot (n+3)}.$$

30. Проверьте справедливость неравенства для заданного  $k$ .

$$\frac{1 \cdot 3 \cdot 5 \cdot \dots \cdot (2k-1)}{2 \cdot 4 \cdot 6 \cdot \dots \cdot 2k} < \frac{1}{\sqrt{3k-1}}.$$

### Задание 3 по теме while и do..while

1. Значение функции  $\sin^2(x)$  можно вычислить с помощью разложения ее в ряд Маклорена

$$\sin^2(x) = x^2 - \frac{x^4}{3} + \frac{2x^6}{45} - \dots + (-1)^{n-1} 2^{2n-1} \frac{x^{2n}}{(2n)!} + \dots$$

2. Введите два натуральных числа  $m$  и  $n$ . Проверьте, являются ли данные числа взаимно-простыми.

3. Напишите программу сложения двух рациональных дробей. Если полученный результат является сократимой дробью, то сократите эту дробь.

4. Напишите программу умножения двух рациональных дробей. Если полученный результат является сократимой дробью, то сократите эту дробь.

5. Вычислите значение корня  $n$ -ой степени  $y = \sqrt[n]{x}$  с точностью EPS с использованием итерационной формулы Ньютона:

$$Y_0 = 1$$

$$Y_i = 1/n ((n-1)Y_{i-1} + X/Y^{n-1}_{i-1}) \quad (i = 1, 2, 3, \dots).$$

Вычисления производить пока  $|Y_i - Y_{i-1}|$  не станет меньше EPS.

Определите количество итераций, за которое достигается эта точность.

6. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=0}^{\infty} (-1)^n \times \frac{n}{n^3 + 2}, \quad \alpha = 0,001$$

7. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=1}^{\infty} (-1)^n \times \frac{1}{3n^2}, \quad \alpha = 0,0001$$

8. Вычислите и выведите на экран значения функции  $y = \frac{1}{x}$ ,

превосходящие  $\varepsilon$  для  $x$ , принимающего значения 1, 2, ... . Значение  $\varepsilon$  введите с клавиатуры.

9. Введите натуральное число  $n$ . Определите количество цифр в этом числе.

10. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=1}^{\infty} (-1)^{n+1} \times \frac{1}{n}, \quad \alpha = 0,0001$$

11. Вычислите значение функции с помощью ряда Маклорена с заданной точностью  $S(x)$ :

$$12. S(x) = x - \frac{x^3}{3} + \dots + (-1)^n \frac{x^{2n+1}}{2n+1} + \dots$$

13. Вычислите значение числа  $\pi$  с заданной точностью, используя формулу  $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$ . Выведите количество слагаемых, которое понадобилось для вычислений.

14. Вычислите значение числа  $\pi$  с заданной точностью, используя формулу  $\frac{\pi}{8} = \frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots$

15. Вычислите значение функции с помощью ряда Маклорена с заданной точностью  $S(x)$ :

$$16. S(x) = 1 + \frac{2x}{1!} + \dots + \frac{(2x)^n}{n!} + \dots$$

17. Вычислите значение функции с помощью ряда Маклорена с заданной точностью  $S(x)$ :  $S(x) = \frac{x^3}{3} - \frac{x^5}{15} + \dots + (-1)^{n+1} \frac{x^{2n+1}}{4n^2 - 1} + \dots$

18. Вычислите значение функции с помощью ряда Маклорена с заданной точностью  $S(x)$ :  $S(x) = 1 - \frac{x^2}{2!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!} + \dots$

19. Вычислите значение квадратного корня  $y = \sqrt{x}$  с точностью EPS с использованием итерационной формулы Ньютона:

$$Y_0 = 1$$

$$Y_i = 1/2 (Y_{i-1} + X/Y_{i-1}) \quad (i = 1, 2, 3, \dots).$$

Вычисления производить пока  $|Y_i - Y_{i-1}|$  не станет меньше EPS.

Определите количество итераций, за которое достигается эта точность.

20. Найдите сумму членов ряда  $S = 1 + 1/2 + 1/4 + 1/8 + \dots$

Сумму вычислять, пока очередной член ряда не станет меньше EPS.

21. Вычислите значение кубического корня  $y = \sqrt[3]{x}$  с точностью EPS с использованием итерационной формулы Ньютона:

$$Y_0 = 1$$

$$Y_i = 1/3 (2Y_{i-1} + X/Y_{i-1}^2) \quad (i = 1, 2, 3, \dots).$$

Вычисления производить пока  $|Y_i - Y_{i-1}|$  не станет меньше EPS.

Определите количество итераций, за которое достигается эта точность.

22. Алгоритм Евклида нахождения НОД( $m, n$ ) основан на следующих свойствах этой величины: пусть  $m$  и  $n$  – два натуральных числа и пусть  $m \geq n$ . Тогда для чисел  $m, n$  и  $r$ , где  $r$  – остаток от деления  $m$  на  $n$ , выполняется равенство  $\text{НОД}(m, n) = \text{НОД}(n, r)$ . Используя алгоритм Евклида, найдите наибольший общий делитель  $m$  и  $n$ .

23. Значение функции  $\text{LN}(1 + X)$  можно вычислить с помощью разложения ее в ряд Маклорена

$$\text{LN}(1 + X) = X - X^2/2 + X^3/3 - X^4/4 + \dots$$

Вычислите  $\text{LN}(1 + X)$  с точностью EPS, т.е., вычисление суммы ряда нужно продолжать до тех пор, пока абсолютная величина очередного члена ряда не станет меньше EPS. Определите количество членов ряда, которое для этого понадобилось.

24. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=0}^{\infty} (-1)^n \times \frac{n}{n^3 + 2}, \quad \alpha = 0,001$$

25. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=1}^{\infty} (-1)^n \times \frac{1}{3n^2}, \quad \alpha = 0,0001$$

26. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=1}^{\infty} (-1)^{n+1} \times \frac{1}{n}, \quad \alpha = 0,0001$$

27. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :

$$\sum_{n=1}^{\infty} (-1)^{n+1} \times \frac{1}{(2n)^3}, \quad \alpha = 0,001$$

28. Вычислите сумму ряда с заданной степенью точности

$$\alpha: \sum_{n=0}^{\infty} (-1)^n \times \frac{1}{n(2n+1)}, \quad \alpha = 0,001$$

29. Вычислите сумму ряда с заданной степенью точности

$$\alpha: \sum_{n=1}^{\infty} (-1)^n \times \frac{1}{(2n+1)}, \quad \alpha = 0,0001$$

30. Вычислите сумму ряда с заданной степенью точности  $\alpha$ :



$$\sum_{n=1}^{\infty} \left( \frac{1}{2} \right)^n \times \frac{n}{2^n}, \quad \alpha = 0,001$$

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### 4. Основные теоретические сведения

#### Оператор **if**

Условный оператор **if** часто применяется программистами и имеется во всех языках программирования. Оператор **if** позволяет вашей программе в зависимости от условий выполнить оператор или группу операторов, основываясь на значении булевой переменной или выражения. Оператор **if** является основным оператором выбора в Java и позволяет выборочно изменять ход выполнения программы - и это одно из основных отличий между программированием и простым вычислением.

Оператор **if** начинается с ключевого слова **if**. Ключевое слово **if** должно сопровождаться булевым выражением, заключённым в скобки. Самая простая форма выглядит так:

```
if (условие) оператор; // если условие истинно, то выполняется оператор
```

Здесь условие - это булево выражение, имеющее значение *true* или *false*. Если условие истинно, то оператор или группа операторов выполняется, если ложно, то оператор не выполняется. Очень часто булево выражение в операторе **if** содержит какое-нибудь сравнение, но можно использовать булеву переменную или константу

```
// Если кот голоден
if (isHungry)
```

Оператор **if** продолжается заключённым в фигурные скобки фрагментом, который называют блоком операторов. Если используется только один оператор, то фигурные скобки можно опустить. Но практика показывает, что лучше их всегда использовать, особенно в сложных проектах, когда постоянно приходится что-то переделывать.

Напишем следующий пример:

```
if (2 * 2 == 5)
{
    mResultEditText.setText("Дважды два равно пяти!");
}
```

Как вы думаете, что появится на экране? Правильно, ничего, так как оператор не будет выполняться, потому что условие  $2 * 2 == 5$  является ложным.

Обратите внимание, что оператор равенства состоит из двух символов знака равно. Об этом часто забывают начинающие программисты.

Вот список операторов, которые можно использовать в условных выражениях:

Оператор	
<	Меньше чем
<=	Меньше или равно
>	Больше чем
>=	Больше или равно
==	Равно
!=	Не равно

Результат сравнения удобно использовать для изменения логики программы. Например, если кот голоден, то накормить его. Или если число нечётное, то сделать его чётным.

```
if(number % 2 != 0)
    ++number;
```

Существует расширенный вариант оператора **if** с использованием ключевого слова **else**:

```
if (условие) оператор; // если условие истинно, то выполняется первый оператор
else оператор;         // если условие ложно, то выполняется оператор после
else
```

В этом случае при выполнении условия оператора **if** иницируется только один оператор, если условие не выполняется, то также иницируется только один оператор, который относится к **else**. Также можно использовать блоки операторов, тогда синтаксис будет выглядеть так:

```
if (условие)
{
    оператор1;
    оператор2;
}
else
{
    оператор1;
    оператор2;
}
```

Обе части оператора **if** и **else** не могут выполняться одновременно. А условное выражение, управляющее оператором **if** должно возвращать булево значение.

Например, необходимо вычисления функции:  $Y(x) = x * x$ , при  $x < 0$  и  $Y(x) = 2 * x$ , при  $x \geq 0$ :

```

int x, y;
// Чтение значений x, y из консоли
Scanner in = new Scanner(System.in);
System.out.println("Enter x :");
x = in.nextLine();
System.out.println("Enter y :");
y = in.nextLine();

if (x < 0)
    y = x*x;
else
    y = 2*x;
System.out.printf ("Результат: %i\n", y);

```

### Вложенные операторы if

Вложенный оператор **if** используется для дальнейшей проверки данных после того, как условие предыдущего оператора **if** принимает значение **true**. Иными словами, вложенный оператор применяется в тех случаях, когда для выполнения действия требуется соблюдение сразу нескольких условий, которые не могут быть указаны в одном условном выражении. Необходимо помнить, что во вложенных операторах **if-else** вторая часть **else** всегда относится к ближайшему оператору **if**, за условным выражением которого следует оператор ; или блок операторов. Вот небольшой пример:

```

if(i == 10)
{
    if(j < 20) a = b;
    if(k > 100) c = d;
    else a = c; // else относится к if(k > 100)
}
else a = d; // else относится к if(i == 10)

```

### Цепочка операторов if-else-if

Часто используется цепочка операторов if-else-if - конструкция, состоящая из вложенных операторов if:

```

if (condition)
    statement;
else if (condition)
    statement;
else if (condition)
    statement;
.
.
.
else
    statement;

```

Условные выражения оцениваются сверху вниз. Как только найдено условие, принимающее значение **true**, выполняется ассоциированный с этим условием оператор, а остальная часть цепочки пропускается. Если ни одно из условий не принимает значение **true**, то выполняется последний оператор **else**, который можно рассматривать как оператор по умолчанию. Если же последний оператор **else** отсутствует, а все условные выражения принимают значение **false**, то программа не выполняет никаких действий.

Напишем пример, вычисляющий время года, когда коты поют свои мартовские песни.

```
int month = 3; // март
String season; // время года

if(month == 1 || month == 2 || month == 12)
    season = "Зимушка-зима";
else if (month == 3 || month == 4 || month == 5)
    season = "Весна";
else if (month == 6 || month == 7 || month == 8)
    season = "Лето";
else if (month == 9 || month == 10 || month == 11)
    season = "Осень";
else
    season = "Вы с какой планеты?";

mInfoTextView.setText("Мартовские песни коты поют, когда на дворе " +
season);
```

### Тернарный оператор

Продвинутые программисты часто используют тернарный оператор : вместо if-else. Тернарный оператор использует три операнда и записывается в форме:

```
логическое_условие ? выражение1 : выражение2
```

Если **логическое\_условие** истинно, т.е. возвращает **true**, то берётся (или вычисляется) первое выражение слева от двоеточия, если возвращается **false**, то берётся второе выражение справа от двоеточия.

Например, нужно вычислить, какое из двух чисел больше и занести результат в третью переменную:

```
int largerNum;
int lowNum = 9;
int highNum = 27;

if(lowNum < highNum) // если первое число меньше второго
{
    largerNum = highNum;
} else { // иначе
    largerNum = lowNum;
}

При тернарном варианте код будет следующим:
```

```
int lowNum = 9;
int highNum = 27;
int largerNum = lowNum < highNum ? highNum : lowNum;
```

### Логические операторы

Логические операторы работают только с операндами типа **boolean**. Все логические операторы с двумя операндами объединяют два логических значения, образуя результирующее логическое значения. Не путайте с побитовыми логическими операторами.

**Таблица логических операторов в Java**

Оператор	Описание
&	Логическое AND (И)
&&	Сокращённое AND
	Логическое OR (ИЛИ)
	Сокращённое OR
^	Логическое XOR (исключающее OR (ИЛИ))
!	Логическое унарное NOT (НЕ)
&=	AND с присваиванием
=	OR с присваиванием
^=	XOR с присваиванием
==	Равно
!=	Не равно
?:	Тернарный (троичный) условный оператор

Логические операторы **&**, **|**, **^** действуют применительно к значениям типа **boolean** точно так же, как и по отношению к битам целочисленных значений. Логический оператор **!** инвертирует (меняет на противоположный) булево состояние: *!true == false* и *!false == true*.

**Таблица. Результаты выполнения логических операторов**

A	B	A   B	A & B	A ^ B	!A
false	False	false	false	false	true
true	False	true	false	true	false
false	True	true	false	true	true
true	True	true	true	false	false

### Сокращённые логические операторы

Кроме стандартных операторов **AND** (&) и **OR** (!) существуют сокращённые операторы **&&** и **||**.

Если взглянуть на таблицу, то видно, что результат выполнения оператора **OR** равен **true**, когда значение операнда A равно **true**, независимо от значения операнда B. Аналогично, результат выполнения оператора **AND** равен **false**, когда значение операнда A равно **false**, независимо от значения операнда B. Получается, что нам не нужно вычислять значение второго операнда, если результат можно определить уже по первому операнду. Это становится удобным в тех случаях, когда значение правого операнда зависит от значения левого.

Рассмотрим следующий пример. Допустим, мы ввели правило - кормить или не кормить кота в зависимости от числа пойманных мышек в неделю. Причём число мышек зависит от веса кота. Чем больше кот, тем больше он должен поймать мышей.

```
int mouse; // число мышек
int weight; // вес кота в граммах
mouse = 5;
weight = 4500;

if (mouse != 0 & weight / mouse < 1000) {
    mInfoTextView.setText("Можно кормить кота");
}
```

Если запустить программу, то пример будет работать без проблем - пять мышей в неделю вполне достаточно, чтобы побаловать кота вкусным завтраком. Если он поймает четырёх мышей, то начнутся проблемы с питанием кота, но не с программой - она будет работать, просто не будет выводить сообщение о разрешении покормить дармоеда.

Теперь возьмём крайний случай. Кот обленился и не поймал ни одной мышки. Значение переменной **mouse** будет равно 0, а в выражении есть оператор деления. А делить на 0 нельзя и наша программа закроется с ошибкой.

Казалось бы, мы предусмотрели вариант с 0, но Java вычисляет оба выражения **mouse != 0** и **weight / mouse < 1000**, несмотря на то, что уже в первом выражении возвращается **false**.

Перепишем условие следующим образом (добавим всего лишь один символ):

```
if (mouse != 0 && weight / mouse < 1000) {  
    mInfoTextView.setText("Можно кормить кота");  
}
```

Теперь программа работает без краха. Как только Java увидела, что первое выражение возвращает **false**, то второе выражение с делением просто игнорируется.

Сокращённые варианты операторов **AND** и **OR** принято использовать в тех ситуациях, когда требуются операторы булевой логики, а их односимвольные родственники используются для побитовых операций.

### Оператор switch

В отличие от операторов **if-then** и **if-then-else**, оператор **switch** применим к известному числу возможных ситуаций. Можно использовать простые типы **byte**, **short**, **char**, **int**. Также можно использовать **Enum** и **String** (начиная с JDK7), и специальные классы, которые являются обёрткой для примитивных типов: **Character**, **Byte**, **Short**, **Integer**.

Дублирование значений **case** не допускается. Тип каждого значения должен быть совместим с типом выражения.

Команду **switch** часто называют командой выбора. Выбор осуществляется в зависимости от целочисленного выражения. Форма команды выглядит так:

```
switch (ВыражениеДляСравнения) {  
    case Совпадение1:  
        команда;  
        break;  
    case Совпадение2:  
        команда;  
        break;  
    case Совпадение3:  
        команда;  
        break;  
    default:  
        оператор;  
        break;  
}
```

Параметр *ВыражениеДляСравнения* - выражение, в результате вычисления которого получается целое число (как правило). Команда **switch** сравнивает результат *ВыражениеДляСравнения* с каждым последующим *Совпадением*. Если обнаруживается совпадение, то исполняется команда или набор команд, которые прописаны за данным оператором. Если совпадений не будет, то исполняется команда после ключевого слова **default**. Однако оператор **default** не является обязательным. В этом случае при отсутствии совпадений программа не выполняет никаких действий.



Каждая секция **case** обычно заканчивается командой **break**, которая передаёт управление к концу команды **switch**.

Рассмотрим простейший пример с месяцами. Запустим наш учебный проект и добавим код в обработчик нажатия кнопки:

```
int month = 3;
String monthString;
switch (month) {
    case 1: monthString = "Январь";
            break;
    case 2: monthString = "Февраль";
            break;
    case 3: monthString = "Март";
            break;
    case 4: monthString = "Апрель";
            break;
    case 5: monthString = "Май";
            break;
    case 6: monthString = "Июнь";
            break;
    case 7: monthString = "Июль";
            break;
    case 8: monthString = "Август";
            break;
    case 9: monthString = "Сентябрь";
            break;
    case 10: monthString = "Октябрь";
            break;
    case 11: monthString = "Ноябрь";
            break;
    case 12: monthString = "Декабрь";
            break;
    default: monthString = "Не знаем такого";
            break;
}
mInfoTextView.setText(monthString);
```

Запустите проект и нажмите кнопку - в текстовом поле появится слово *Март* (любимый месяц котов).

При желании, можно переписать пример с использованием **if-then-else**:

```
int month = 3;
if (month == 1) {
    mInfoTextView.setText("Январь");
} else if (month == 2) {
    mInfoTextView.setText("Февраль");
}
... // и так далее
```

В каждом блоке **case** имеется оператор **break**, который прерывает свой блок кода. Его нужно использовать обязательно, иначе выполнение кода продолжится. Хотя иногда это и используется.

```
java.util.ArrayList<String> futureMonths = new
java.util.ArrayList<String>();

int month = 8;

switch (month) {
case 1:
```

```

        futureMonths.add("January");
    case 2:
        futureMonths.add("February");
    case 3:
        futureMonths.add("March");
    case 4:
        futureMonths.add("April");
    case 5:
        futureMonths.add("May");
    case 6:
        futureMonths.add("June");
    case 7:
        futureMonths.add("July");
    case 8:
        futureMonths.add("August");
    case 9:
        futureMonths.add("September");
    case 10:
        futureMonths.add("October");
    case 11:
        futureMonths.add("November");
    case 12:
        futureMonths.add("December");
        break;
    default:
        break;
}

if (futureMonths.isEmpty()) {
    mInfoTextView.setText("Invalid month number");
} else {
    for (String monthName : futureMonths) {
        mInfoTextView.setText(monthName);
    }
}

```

Если код в блоках **case** совпадает, то блоки можно объединить. Например, код для подсчёта дней в месяце:

```

int month = 2;
int year = 2012;
int numDays = 0;

switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numDays = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numDays = 30;
        break;
    case 2:
        if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))

```

```

        numDays = 29;
    else
        numDays = 28;
    break;
default:
    mInfoTextView.setText("Несуществующий месяц");
    break;
}
mInfoTextView.setText("Число дней = " + numDays);

```

При изучении оператора `if` мы рассматривали пример с временами года. Перепишем его с использованием оператора **switch**:

```

int month = 3;
String season;

switch (month) {
case 12:
case 1:
case 2:
    season = "Зимушка-зима";
    break;
case 3:
case 4:
case 5:
    season = "Весна";
    break;
case 6:
case 7:
case 8:
    season = "Лето";
    break;
case 9:
case 10:
case 11:
    season = "Осень";
    break;
default:
    season = "Вы с какой планеты?";
}

mInfoTextView.setText("Мартовские песни коты поют, когда на дворе "
    + season);

```

Следующий пример случайным образом генерирует английские буквы. Программа определяет, гласные они или согласные:

```

Random random = new Random();

for (int i = 0; i < 100; i++) {
    int c = random.nextInt(26) + 'a';
    mInfoTextView.setText((char)c + ", " + c + ": ");
    switch (c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        mInfoTextView.setText("Гласная");
        break;
    case 'y':

```

```

        case 'w':
            mInfoTextView.setText("Условная гласная");
            break;

        default:
            mInfoTextView.setText("Согласная");
            break;
    }
}

```

Так как метод **Random.nextInt(26)** генерирует значение между 0 и 26, для получения символа нижнего регистра остаётся прибавить смещение 'a', при этом символ **a** автоматически преобразуется к типу **int**. Символы в секциях **case** также представляют собой целочисленные значения, используемые для сравнения.

Чтобы вывести переменную **c** в символьном виде, её необходимо преобразовать к типу **char**, иначе значение будет выведено в числовом виде.

В Java SE 7 появилась возможность использовать объект **String** в операторе **switch**. Возможно, это будет работать и на Android в будущем (Upd.: вроде уже работает):

```

public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
        }
    }
}

```

```

        case "october":
            monthNumber = 10;
            break;
        case "november":
            monthNumber = 11;
            break;
        case "december":
            monthNumber = 12;
            break;
        default:
            monthNumber = 0;
            break;
    }

    return monthNumber;
}

public static void main(String[] args) {

    String month = "August";

    int returnedMonthNumber =
        StringSwitchDemo.getMonthNumber(month);

    if (returnedMonthNumber == 0) {
        System.out.println("Invalid month");
    } else {
        System.out.println(returnedMonthNumber);
    }
}
}

```

### Важные свойства оператора **switch**:

- ✓ Оператор **switch** отличается от оператора **if** тем, что может выполнять проверку только равенства, а оператор **if** может вычислять результат булева выражения любого типа.
- ✓ Две константы **case** в одном и том же операторе **switch** не могут иметь одинаковые значения
- ✓ Оператор **switch** эффективнее набора вложенных операторов **if**

### Цикл **for**

Конструкция **for** управляет циклами. Команда выполняется до тех пор, пока управляющее логическое выражение не станет ложным.

Блок-схема.

Цикл **for** является наиболее распространённым циклом в программировании, поэтому его следует изучить. Цикл **for** проводит инициализацию перед первым шагом цикла. Затем выполняется проверка условия цикла, и в конце каждой итерации происходит изменение управляющей переменной. Выглядит следующим образом:

```

for (инициализация; логическое выражение (условие); шаг (итерация))
    команда

```

Любое из трёх выражений цикла (инициализация, логическое выражение или шаг) можно пропустить. Перед выполнением каждого шага цикла

проверяется условие цикла. Если условие окажется ложным, то выполнение продолжится с инструкции, следующей за конструкцией **for**.

Помните, что выражение инициализации выполняется один раз, затем вычисляется условие, которое должно быть булевым выражением.

Как правило, цикл **for** используют для перебора. В качестве имени первой переменной часто используют **i** (сокр. от **init**), но вы можете использовать любое имя.

Простейший пример:

```
for (int x = 0; x < 9; x = x + 1)
    System.out.println("\nЗначение x: " + x);
```

```
Значение x: 0
Значение x: 1
Значение x: 2
Значение x: 3
Значение x: 4
Значение x: 5
Значение x: 6
Значение x: 7
Значение x: 8
```

В этом примере переменной **x** присваивается начальное значение, равное нулю. Затем выполняется проверка условия в логическом выражении (**x < 9**), если результат проверки истинен, то выполняется оператор после выражения цикла. После чего процесс повторяется. Процесс продолжается до тех пор, пока результат проверки условия не станет ложным.

Третье выражение в цикле - шаг, то есть, на какое значение нужно изменить переменную. Строго говоря, в таком виде (**x = x + 1**) современные программисты не пишут, так как есть укороченная форма записи (**x++**). Предыдущий пример можно переписать по другому:

```
for (int x = 0; x < 9; x++)
```

Эта запись является классической и правильной, если нам нужно посчитать от 0 до 8. Может возникнуть соблазн написать, например, так:

```
for (int x = 0; x <= 8; x++)
```

Результат будет таким же, но такой код нежелателен. Старайтесь писать традиционно. Особенно это проявляется при работе с массивами.

Увеличение значения переменной на единицу - весьма распространённый случай. Но это не является обязательным условием цикла, вы можете установить шаг и с помощью умножения, вычитания и других действий.

Например, мы хотим вывести процесс уменьшения жизней у очень любопытной кошки:

```
for (int life = 9; life >= 0; life--)  
    System.out.println ("\nУ кошки осталось жизней: " + life);
```

```
У кошки осталось жизней: 9  
У кошки осталось жизней: 8  
У кошки осталось жизней: 7  
У кошки осталось жизней: 6  
У кошки осталось жизней: 5  
У кошки осталось жизней: 4  
У кошки осталось жизней: 3  
У кошки осталось жизней: 2  
У кошки осталось жизней: 1  
У кошки осталось жизней: 0
```

Например, выводим чётные числа.

```
for (int x = 0; x < 9; x += 2)  
    System.out.println(" " + x); // между кавычками пробел
```

Получим:

```
0 2 4 6 8
```

Если нужно выполнить несколько операторов в цикле, то используют фигурные скобки.

```
for (int kitten = 1; kitten < 10; kitten++) {  
    mInfoTextView.append("\nСчитаем котят: " + kitten);  
    mResultEditText.setText("Ура! Нас подсчитали");  
}
```

В этом примере выполняется цикл с выводом числа подсчитанных котят, а также выводится текст в текстовом поле. Кстати, это один из примеров неправильного кода, когда в текстовом поле девять раз подряд выводится одна и та же строка. Мы этого не замечаем, но в реальности процессор выполняет лишнюю работу и второй оператор безусловно нужно вынести за пределы блока кода, который относится к циклу. Подобные ошибки очень часто встречаются у начинающих программистов, которые забывают, как работает цикл.

Когда мы объявляем переменную в первой части оператора **for(int i = 0; ...)**, то область видимости переменной ограничена телом цикла и эта переменная не доступна на другом участке кода. Это подходящий вариант, если переменная больше нигде не используется. При этом переменная имеет область видимости и продолжительность существования, совпадающие с видимостью и продолжительностью жизни самого цикла. Вне цикла переменная прекратит своё существование.

Если управляющую переменную цикла нужно использовать в других частях приложения, то её не следует объявлять внутри цикла.

```
int i; // эта переменная нам понадобится не только в цикле
for(i = 0; i < 10; i++){
    // что-то делаем
}

// можем использовать переменную где-то ещё
x = i + 10;
```

С другой стороны, если видимость переменной ограничена в пределах цикла, то не будет никакого конфликта, если вы будете использовать одинаковые имена переменных в разных циклах **for**, так как они не будут друг другу мешать.

### Использование нескольких переменных

Иногда требуется указать несколько переменных в инициализационной части цикла. Для начала посмотрим на стандартный пример:

```
int a, b;
b = 4;

for(a = 1; a < b; a++) {
    mInfoTextView.append("a = " + a + "\n");
    mInfoTextView.append("b = " + b + "\n");
    b--;
}
```

В данном цикле используются две переменные. Можно включить обе переменные в оператор **for**, чтобы не выполнять обработку второй переменной вручную:

```
int a, b;

for(a = 1, b = 4; a < b; a++, b--) {
    mInfoTextView.append("a = " + a + "\n");
    mInfoTextView.append("b = " + b + "\n");
}
```

Как видно из кода, запятая служит разделителем для двух переменных. Теперь оба разделённых запятой оператора в итерационной части цикла выполняются при каждом выполнении цикла. Данный код считается более эффективным, хотя на практике встречается редко.

Части цикла могут быть пустыми.

Оставим пустым первое выражение.

```
int i = 0;
for(; i < 10; i++){
    // что-то делаем
    mInfoTextView.append("\n" + i);
}
```

В следующем примере нет инициализационного и итерационного выражения:

```
int i;
boolean kind = false;
```



```

i = 0;
for( ; !kind; ) {
    mInfoTextView.append("i равно " + i + "\n");
    if(i == 10) done = true;
    i++;
}

```

А можно вообще все три части оператора оставить пустыми:

```

for( ; ; ) {
    //...
}

```

В этом случае создаётся бесконечный цикл, который никогда не завершится. Практического смысла данный код не имеет.

## Цикл **while**

Оператор цикла **while** есть практически во всех языках программирования. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно.

Форма цикла **while** следующая:

```

while (условие) {
    // тело цикла
}

```

Здесь *условие* должно быть любым булевым выражением. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда *условие* становится ложным, управление передаётся строке кода, которая идёт после цикла. Если в цикле используется только один оператор, то фигурные скобки можно опустить (но лучше так не делать).

Логическое выражение вычисляется перед началом цикла, а затем каждый раз перед выполнением очередного повторения оператора.

Напишем пример с использованием цикла **while**, который выполняет обратный отсчёт от 10 до 0:

```

int counter = 10;
while (counter > 0) {
    mInfoTextView.append("Осталось " + counter + " сек.\n");
    counter--;
}

```

Программа выведет десять строк:

```

Осталось 10 сек.
Осталось 9 сек.
Осталось 8 сек.
Осталось 7 сек.
Осталось 6 сек.
Осталось 5 сек.
Осталось 4 сек.
Осталось 3 сек.
Осталось 2 сек.
Осталось 1 сек.

```

Если нужно увеличивать от единицы до 10, то код будет следующим.

```

int counter = 1;
while(counter < 11){

```

```

        System.out.println(counter);
        counter++;
    }

```

Поскольку цикл **while** вычисляет своё условное выражение в начале цикла, то тело цикла не будет выполняться, если условие с самого начала было ложным.

```

boolean isHungry; // голоден ли кот
isHungry = true; // где вы видали сытого кота?
while(!isHungry) {
    mInfoTextView.setText("Случилось чудо - кот не голоден");
}

```

Вы никогда не увидите сообщение, так как сытый кот - это из области фантастики.

Тело цикла **while** может быть пустым. Например:

```

int i, j;
i = 10;
j = 30;
// вычисляем среднее значение двух переменных
while (++i < --j)
    ; // у цикла нет тела
mInfoTextView.setText("Среднее значение равно " + i);

```

Пример работает следующим образом. Значение переменной **i** увеличивается, а значение переменной **j** уменьшается на единицу. Затем программа сравнивает два новых значения переменных. Если новое значение переменной **i** меньше нового значения переменной **j**, то цикл повторяется. На каком-то этапе значения обоих переменных сравняются и цикл прекратит свою работу. При этом переменная **i** будет содержать среднее значение исходных значений двух переменных. Достаточно изуверский способ вычисления среднего значения, но здесь главное увидеть пример цикла без тела. Все действия выполняются внутри самого условного выражения. Учтите, если значение первой переменной с самого начала будет больше второй переменной, то код пойдёт коту под хвост.

Профессиональные программисты часто используют циклы без тела, в которых само по себе управляющее выражение может выполнять все необходимые действия.

## Цикл do-while

Конструкция цикла:

```

do
    // команда (тело цикла)
while(условие-логическое выражение)

```

Отличие цикла **do-while** от цикла [while](#) состоит в том, что цикл **do-while** выполняется по крайней мере один раз, даже если условие изначально ложно. В цикле **while** такое не произойдёт, так как тело цикла не отработается. Цикл **do-while** используется реже, чем **while**.

Бывают ситуации, когда проверку прерывания цикла желательно выполнять в конце цикла, а не в его начале. И данный цикл позволяет это

сделать. При каждом повторении цикла **do-while** программа сначала выполняет тело цикла, а затем вычисляет условное выражение. Если это выражение истинно, то цикл повторяется. В противном случае выполнение цикла прерывается. Как и в других циклах Java, условие должно иметь булево значение.

Перепишем пример из урока по циклу **while** на новый лад с использованием цикла **do-while**:

```
int counter = 10;
do {
    mInfoTextView.append("Осталось " + counter + " сек.\n");
    counter--;
} while (counter > 0);
```

Если теперь изменить условие **counter < 0**, то цикл отработает один раз и выведет одну строку:

Осталось 10 сек.

Пример можно переписать следующим образом:

```
int counter = 10;
do {
    mInfoTextView.append("Осталось " + counter + " сек.\n");
} while (--counter > 0);
```

Мы объединили декремент счётчика и сравнение с нулём в одном выражении. Программа работает следующим образом. Вначале она выполняет операцию уменьшения на единицу и возвращая новое значение счётчика. Затем сравнивается значение с нулём. Если оно больше нуля, выполнение цикла продолжается. В противном случае цикл прерывается.

## Оператор break

Оператор **break** завершает последовательность операторов в операторе **switch**, позволяет выйти из цикла и в качестве оператора безусловного перехода (**goto**).

Рассмотрим пример выхода из цикла. Используя оператор **break**, можно вызвать немедленное завершение цикла, пропуская условное выражение и другой код в теле цикла. Когда приложение встречает оператор **break** внутри цикла, оно прекращает выполнение цикла и передаёт управление оператору, следующему за циклом.

```
for(int i = 0; i < 100; i++) {
    if(i == 5) break; // выходим из цикла, если i равно 5
    mInfoTextView.append("i: " + i + "\n");
}
mInfoTextView.append("Цикл завершён");
```

Получим следующий результат:

```
i: 0
i: 1
i: 2
i: 3
```

```
i: 4
```

### Цикл завершён

Вроде хотели, чтобы цикл `for` выполнялся от 0 до 100, а сами коварно вышли из него, когда значение переменной `i` стало равным 4.

Код можно переписать с использованием цикла `while`:

```
int i = 0;

while(i < 100) {
    if(i == 5) break; // выходим из цикла, если i равно 5
    mInfoTextView.append("i: " + i + "\n");
    i++;
}
mInfoTextView.append("Цикл завершён");
```

Результат будет такой же.

При использовании вложенных циклов оператор **break** осуществляет выход только из самого внутреннего цикла, не оказывая влияния на внешний цикл. Создадим пример с использованием вложенных циклов:

```
for (int i = 1; i < 4; i++) {
    mInfoTextView.append("Проход " + i + ": ");
    for (int j = 0; j < 100; j++) {
        if (j == 5)
            break; // выходим из цикла, если j равно 5
        mInfoTextView.append(j + " ");
    }
    mInfoTextView.append("\n");
}
mInfoTextView.append("Циклы завершены");
```

Получим следующий результат:

```
Проход 1: 0 1 2 3 4
Проход 2: 0 1 2 3 4
Проход 3: 0 1 2 3 4
Циклы завершены
```

В примере оператор **break** прервал выполнение второго цикла, но первый цикл продолжал работать.

В цикле можно использовать несколько операторов **break**, но увлекаться не стоит. Лучше переделайте код.

### Оператор `continue`

Иногда требуется, чтобы повторение цикла начиналось с более раннего оператора его тела. В циклах **while** и **do-while** оператор **continue** вызывает передачу управления непосредственно управляющему условному выражению цикла. В цикле **for** управление передаётся вначале итерационной части цикла **for**, а потом условному выражению. При этом во всех циклах промежуточный код пропускается.

В следующем примере выводим два числа в каждой строке:

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

```

        mInfoTextView.append(i + " ");
        if (i % 2 == 0)
            continue;
        mInfoTextView.append("\n");
    }

```

В этом коде оператор `%` служит для проверки чётности значения переменной `i`. Если значение чётное, выполнение цикла продолжится дальше, а если нечётное, то произойдёт переход на новую строку. В результате мы получим следующий текст:

```

0 1
2 3
4 5
6 7
8 9

```

Как и оператор **break**, оператор **continue** может содержать метку содержащего его цикла, который нужно продолжить. Создадим пример вывода треугольной таблицы умножения чисел от 0 до 9.

```

outer: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (j > i) {
            mInfoTextView.append("\n");
            continue outer;
        }
        mInfoTextView.append(" " + (i * j));
    }
    mInfoTextView.append("\n");
}

```

В этом примере оператор **continue** прерывает цикл подсчёта значений переменной `j` и продолжает его со следующей итерации цикла подсчёта переменной `i`. На экране увидим текст в таком виде:

```

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
и т.д.

```

Данный оператор в практике встречается достаточно редко. Но на всякий случай помните о нём. Иногда он может пригодиться.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_  
Фамилия, инициалы учащегося \_\_\_\_\_  
Дата выполнения работы \_\_\_\_\_  
Тема работы: \_\_\_\_\_  
Цель работы: \_\_\_\_\_  
Оснащение работы: \_\_\_\_\_  
Результат выполнения работы: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Назначение оператора if?
2. В чем отличие между операторами ветвления и выбора?
3. Цикл это?
4. Какие есть циклы в языке Java?
5. Какой оператор производит досрочную остановку цикла?
6. Какой оператор прерывает текущую итерацию цикла?

## **8. Рекомендуемая литература**

13. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
14. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
15. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
16. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
17. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
18. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ с использованием массивов»

Минск  
2018

## Лабораторная работа № 4

### Тема работы: «Разработка программ с использованием массивов»

#### 1. Цель работы

Закрепить навык работы с массивами различных видов.

#### 2. Задание

Два задания обязательны для выполнения.

Номер варианта соответствует вашему номеру по списку.

Примечание: в Java нумерация элементов массива идет с 0, в примерах с 1. Сделайте соответствующие поправки.

#### Задания 1 по теме одномерные массивы

1. Вычислите  $A_0 + A_1x + A_2x^2 + A_3x^3 + \dots + A_Nx^N$ , используя схему Горнера. В соответствии со схемой Горнера данный многочлен преобразуется к виду:  
$$(\dots((A_Nx + A_{N-1})x + A_{N-2})x + \dots + A_1)x + A_0.$$
2. «Переверните» последовательность  $A_1, A_2, A_3, \dots, A_N$ , т.е. поменяйте местами  $A_1$  с  $A_N$ ,  $A_2$  с  $A_{N-1}$  и т.д. Встроенную функцию не использовать.
3. Вычислите  $A_1x + A_2x^2 + A_3x^3 + \dots + A_Nx^N$ , не используя схему Горнера.
4. Вычислите  $A_1 - A_2 + A_3 - \dots + (-1)^{N-1}A_N$ .
5. Вычислите  $A_1A_2A_3 \dots A_N$ .
6. Вычислите  $-\frac{A_1}{1!} + \frac{A_2}{2!} - \dots + (-1)^N \frac{A_N}{N!}$ .
7. Даны координаты  $\{(x_i; y_i)\}$ ,  $(i = 1, \dots, n)$   $n$  заводов потребителей сырья и координаты места добычи сырья  $(x_C; y_C)$ . Найдите расстояния от места добычи сырья до каждого завода, а также среднее арифметическое этих расстояний.
8. Дан массив  $A$ , состоящий из  $n$  элементов. Сформируйте «сглаженный» массив, заменив в исходном все элементы, кроме крайних, по формуле

$$A_i = \frac{A_{i-1} + A_i + A_{i+1}}{3}, \quad i = 2, 3, \dots, n-1.$$

При сглаживании используются лишь старые значения элементов массива.

9. Дан массив  $a$ , состоящий из  $n$  элементов. Вычислите  $a_1, a_1 + a_2, a_1 + a_2 + a_3, \dots, a_1 + a_2 + a_3 + \dots + a_n$ .
10. Дан массив  $a$ , состоящий из  $n$  элементов. Измените ее следующим образом  $a_1, -a_2, a_3, \dots, (-1)^{n-1}a_n$ .
11. Дан массив  $a$ , состоящий из  $n$  элементов. Получите массив  $b$ , где  $b_k = 2a_k + k$ .
12. Дан массив  $a$ , состоящий из  $n$  элементов. Найдите сумму элементов



массива, стоящих на нечетных местах.

13. Дан массив  $a$ , состоящий из  $n$  элементов. Получите новый массив, поменяв элементы, стоящие на четных местах, с элементами, стоящими на нечетных местах, т.е.  $a_1$  с  $a_2$ ,  $a_3$  с  $a_4$ ,  $a_5$  с  $a_6$  и т.д.
14. Дан массив  $a$ , состоящий из  $n$  элементов. Найдите среднее арифметическое значение элементов массива, стоящих на четных местах.
15. Дан массив  $a$ , состоящий из  $n$  элементов. Найдите сумму  $a_1 + 2a_2 + 3a_3 + \dots + na_n$ .
16. Дан числовой массив  $A$ , состоящий из  $n$  элементов. Найдите среднее арифметическое положительных элементов этого массива.
17. Дан числовой массив  $A$ , состоящий из  $n$  натуральных чисел. Определите количество элементов массива, которые больше заданного числа.
18. Дан числовой массив  $A$ , состоящий из  $n$  натуральных чисел. Определите количество элементов массива, являющихся нечетными числами.
19. Дан числовой массив  $A$ , состоящий из  $n$  натуральных чисел. Определите количество элементов массива, являющихся кратными 7.
20. Дан числовой массив  $A$ , состоящий из  $n$  натуральных чисел. Определите количество элементов массива кратных 3, но не кратных 5.
21. Дан числовой массив  $A$ , состоящий из  $n$  натуральных чисел. Определите количество элементов массива, удовлетворяющих условию  $A_i < (A_{i-1} + A_{i+1})/2$ .
22. Дана последовательность  $x_1, x_2, \dots, x_n$ . Определите количество элементов последовательности, больших среднего арифметического значения положительных элементов последовательности.
23. Дана последовательность натуральных чисел  $x_1, x_2, \dots, x_n$ . Измените данную последовательность так, чтобы в начале стояли все четные, а затем – нечетные элементы последовательности.
24. Дана последовательность целых чисел  $x_1, x_2, \dots, x_n$ . Измените данную последовательность так, чтобы в начале стояли все нулевые элементы, затем – отрицательные, а затем – положительные элементы последовательности.
25. Даны массивы  $a$  и  $b$ , состоящие из  $n$  элементов каждый. Получите массив  $c$ , где  $c_k = a_k + b_k$ .
26. Даны массивы  $a$  и  $b$ , состоящие из  $n$  элементов каждый. Получите массив  $c$ , где  $c_k = a_k \cdot b_k$ .
27. Даны массивы  $a$  и  $b$ , состоящие из  $n$  элементов каждый. Получите новые массивы  $a$  и  $b$ , элементы которых вычисляются по правилу:  $a_i = a_i + b_i$ ,  $b_i = a_i - b_i$ .
28. Даны массивы  $a$  и  $b$ , состоящие из  $n$  элементов каждый. Получите новые массивы  $a$  и  $b$ , элементы которых вычисляются по правилу:  $a_i = b_i$ ,  $b_i = -a_i$ .
29. Информация о температуре воздуха за месяц задана в виде массива. Определите, сколько раз температура опускалась ниже  $0^\circ\text{C}$ . Число дней

конкретного месяца введите с клавиатуры.

30. Проверьте, является ли данная числовая последовательность  $a_1, a_2, \dots, a_n$  возрастающей.

## Задания 2 по теме многомерные массивы

1. В данной матрице определите количество столбцов, у которых элементы расставлены в порядке возрастания.
2. Дан массив  $A$ , состоящий из  $n$  натуральных чисел. Найдите элемент массива, сумма цифр которого наибольшая.
3. Дана действительная квадратная матрица  $A$  порядка  $n$ . Найдите количество строк матрицы, сумма модулей элементов которых больше 1.
4. Дана действительная квадратная матрица порядка  $n$ . Найдите сумму элементов, расположенных в заштрихованной части матрицы (рис. 4.1).
5. Дана действительная квадратная матрица порядка  $n$ . Найдите сумму элементов, расположенных в заштрихованной части матрицы (рис. 4.2).
6. Дана действительная квадратная матрица порядка  $n$ . Найдите сумму элементов, расположенных в заштрихованной части матрицы (рис. 4.3).
7. Дана квадратная матрица  $A$  порядка  $n$ . Найдите номер строки матрицы, в которой больше всего единичных элементов.
8. Дана квадратная матрица  $A$  порядка  $n$ . Найдите среднее арифметическое положительных элементов каждого столбца матрицы.
9. Дана квадратная матрица  $A$  порядка  $n$ . Найдите сумму положительных элементов матрицы, стоящих под главной диагональю.
10. Дана квадратная матрица  $A$  порядка  $n$ . Найдите суммы элементов тех строк матрицы, на главной диагонали которой стоят отрицательные элементы.
11. Дана квадратная матрица  $A$  порядка  $n$ . Постройте вектор, элементы которого являются наибольшими числами каждой строки матрицы.
12. Дана квадратная матрица  $A$  порядка  $n$ . Проверьте, является ли данная матрица симметричной.
13. Дана квадратная матрица  $A$  порядка  $n$ . Проверьте, является ли матрица единичной.
14. Дана квадратная матрица  $A$  порядка  $n$ . Транспонируйте данную матрицу.
15. Дана матрица  $A$  порядка  $n$ . Найдите два наибольших элемента матрицы с указанием номеров строк и столбцов, в которых они находятся.
16. Дана матрица  $A$  порядка  $n$ . Найдите наибольший среди отрицательных элементов матрицы.
17. Дана матрица  $A$  порядка  $n$ . Отсортируйте строки матрицы в порядке возрастания наибольших элементов в каждой строке.
18. Дана матрица  $A$  порядка  $n$ . Поменяйте местами наибольший и наименьший элементы матрицы.
19. Дана матрица  $A$  порядка  $n$ . Поменяйте местами строки: первую с

- последней, вторую с предпоследней и т.д.
20. Дана матрица  $A$  порядка  $n$ . Расставьте строки матрицы в порядке возрастания количества нулевых элементов.
  21. Дана матрица  $A$  порядка  $n$ . Расставьте элементы каждой строки в порядке возрастания.
  22. Дана матрица  $A$  порядка  $n$ . Расставьте элементы строк с четными номерами матрицы в порядке убывания.
  23. Дана матрица  $A$ , имеющая  $n$  строк и  $m$  столбцов, содержащая оценки группы за первый семестр. Выведите номера отличников (оценки не ниже 8).
  24. Даны две матрицы одинаковой размерности. Найдите разность этих матриц.
  25. Даны две матрицы одинаковой размерности. Найдите сумму этих матриц.
  26. Даны две последовательности  $a_1, a_2, \dots, a_n$  и  $b_1, b_2, \dots, b_n$  целых чисел. Среди членов каждой последовательности нет повторяющихся чисел. Получите все члены последовательности  $b_1, b_2, \dots, b_n$ , которые не входят в последовательность  $a_1, a_2, \dots, a_n$ .
  27. Если все элементы какой-либо строки данной матрицы равны между собой, то все элементы такой строки замените нулями.
  28. Найдите наибольшую сумму модулей элементов строк заданной матрицы.
  29. Определите количество строк заданной матрицы, которые упорядочены по возрастанию.
  30. Сложите две треугольные матрицы порядка  $n$ , у которых только элементы над главной диагональю отличны от нуля.

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### 4. Основные теоретические сведения

В любом языке программирования используются массивы, удобные для работы с большим количеством однотипных данных. Если вам нужно обработать сотни переменных, то вызывать каждую по отдельности становится мучительным занятием. В таких случаях проще применить массив. Для наглядности представьте себе собранные в один ряд пустые коробки. В каждую коробочку можно положить что-то одного типа, например, котиков. Теперь, даже не зная их по именам, вы можете выполнить команду **Накормить кота из 3 коробки**. Сравните с командой **Накормить Рыжика**. Чувствуете разницу? Вам не обязательно знать котиков по именам, но вы всё равно сможете справиться с заданием. Завтра в этих коробках могут оказаться другие коты, но это не

составит для вас проблемы, главное знать номер коробки, который называется индексом.



Еще раз повторим теорию. Массивом называется именованное множество переменных одного типа. Каждая переменная в данном массиве называется элементом массива. Чтобы сослаться на определенный элемент в массиве нужно знать имя массива в соединении с целым значением, называемым индексом. Индекс указывает на позицию конкретного элемента относительно начала массива. Обратите внимание, что первый элемент будет иметь индекс 0, второй имеет индекс 1, третий - индекс 2 и так далее. Данное решение было навязано математиками, которым было удобно начинать отсчёт массивов с нуля.

### Объявление массива

Переменную массива можно объявить с помощью квадратных скобок:

```
int[] cats; // мы объявили переменную массива
```

Возможна и альтернативная запись:

```
int cats[]; // другой вариант
```

Здесь квадратные скобки появляются после имени переменной. В разных языках программирования используются разные способы, и Java позволяет вам использовать тот вариант, к которому вы привыкли. Но большинство предпочитает первый вариант. Сами квадратные скобки своим видом напоминают коробки, поэтому вам будет просто запомнить.

Мы пока только объявили массив, но на самом деле его ещё не существует, так как не заполнен данными. Фактически значение массива равно *null*.

### Определение массива

После объявления переменной массива, можно определить сам массив с помощью ключевого слова **new** с указанием типа и размера. Например, массив должен состоять из 10 целых чисел:

```
cats = new int[10];
```

Можно одновременно объявить переменную и определить массив (в основном так и делают):

```
int[] cats = new int[10];
```

Если массив создается таким образом, то всем элементам массива автоматически присваиваются значения по умолчанию. Например, для числовых значений начальное значение будет 0. Для массива типа *boolean* начальное значение будет равно *false*, для массива типа *char* - *'\u0000'*, для массива типа класса (объекты) - *null*.

Последнее правило может запутать начинающего программиста, который забудет, что строка типа **String** является объектом. Если вы объявите массив из десяти символьных строк следующим образом:

```
String[] catNames = new String[10];
```

То у вас появятся строки со значением *null*, а не пустые строки, как вы могли бы подумать. Если же вам действительно нужно создать десять пустых строк, то используйте, например, такой код:

```
for (int i = 0; i < 10; i++)  
    catNames[i] = "";
```

### Доступ к элементам массива

Обращение к элементу массива происходит по имени массива, за которым следует значение индекса элемента, заключенного в квадратные скобки. Например, на первый элемент нашего массива **cats** можно ссылаться как на **cats[0]**, на пятый элемент как **cats[4]**.

В качестве индекса можно использовать числа или выражения, которые создают положительное значение типа **int**. Поэтому при вычислении выражения с типом **long**, следует преобразовать результат в **int**, иначе получите ошибку. С типами **short** и **byte** проблем не будет, так как они полностью укладываются в диапазон **int**.

### Инициализация массива

Не всегда нужно иметь значения по умолчанию. вы можете инициализировать массив собственными значениями, когда он объявляется, и определить количество элементов. Вслед за объявлением переменной массива добавьте знак равенства, за которым следует список значений элементов, помещенный в фигурные скобки. В этом случае ключевое слово **new** не используется:

```
int[] cats = {2, 5, 7, 8, 3, 0}; // массив из 6 элементов
```

Можно смешать два способа. Например, если требуется задать явно значения только для некоторых элементов массива, а остальные должны иметь значения по умолчанию.

```
int[] cats = new int[6]; // массив из шести элементов с начальным
значением 0 для каждого элемента
cats[3] = 5; // четвертому элементу присвоено значение 5
cats[5] = 7; // шестому элементу присвоено значение 7
```

Массивы часто используют в циклах. Допустим, 5 котов отчитались перед вами о количестве пойманных мышек. Как узнать среднее арифметическое значение:

```
int[] mice = {4, 8, 10, 12, 16};
int result = 0;

for(int i = 0; i < 5; i++){
    result = result + mice[i];
}
result = result / 5;
System.out.println("Среднее арифметическое: " + result);
```

Массив содержит специальное поле **length**, которое можно прочитать (но не изменить). Оно позволяет получить количество элементов в массиве. Данное свойство удобно тем, что вы не ошибётесь с размером массива. Последний элемент массива всегда **mice[mice.length - 1]**. Предыдущий пример можно переписать так:

```
int[] mice = { 4, 8, 10, 12, 16 };
int result = 0;

for (int i = 0; i < mice.length; i++) {
    result = result + mice[i];
}
result = result / mice.length; // общий результат делим на число элементов
в массиве
System.out.println("Среднее арифметическое: " + result);
```

Теперь длина массива вычисляется автоматически, и если вы создадите новый массив из шести котов, то в цикле ничего менять не придётся.

Если вам нужно изменять длину, то вместо массива следует использовать списочный массив **ArrayList**. Сами массивы неизменяемы.

Будьте осторожны с копированием массивов. Массив - это не числа, а специальный объект, который по особому хранится в памяти. Чтобы не загромождать вас умными словами, лучше покажу на примере.

Допустим, у нас есть одна переменная, затем мы создали вторую переменную и присвоили ей значение первой переменной. А затем проверим их.

```
int a = 5;
int b = a;
System.out.println("a = " + a + "\nb = " + b);
```

Получим ожидаемый результат.

```
a = 5
b = 5
```

Попробуем сделать подобное с массивом.

```
int[] anyNumbers = {2, 8, 11};
int[] luckyNumbers = anyNumbers;
luckyNumbers[2] = 25;
mInfoTextView.setText("anyNumbers: " + Arrays.toString(anyNumbers)
    + "\nluckyNumbers: " + Arrays.toString(luckyNumbers));
```

Получим результат.

```
anyNumbers: [2, 8, 25];
luckyNumbers: [2, 8, 25];
```

Мы скопировали первый массив в другую переменную и в ней поменяли третий элемент. А когда стали проверять значения у обоих массивов, то оказалось, что у первого массива тоже поменялось значение. Но мы же его не трогали! Магия. На самом деле нет, просто массив остался прежним и вторая переменная обращается к нему же, а не создаёт вторую копию. Помните об этом.

Если же вам реально нужна копия массива, то используйте метод `Arrays.copyOf()`

Если ваша программа выйдет за пределы индекса массива, то программа остановится с ошибкой времени исполнения **`ArrayOutOfBoundsException`**. Это очень частая ошибка у программистов, проверяйте свой код.

### Перебор значений массива

Массивы часто используются для перебора всех значений. Стандартный способ через цикл **`for`**

```
int aNums[] = { 2, 4, 6 };

for (int i = 0; i < aNums.length; i++) {
    String strToPrint = "aNums[" + i + "]= " + aNums[i];
}
```

Также есть укороченный вариант записи

```
for (int num : aNums) {
    String strToPrint = num;
}
```

Нужно только помнить, что в этом случае мы не имеем доступа к индексу массива, что не всегда подходит для задач. Поэтому используется только для обычного перебора элементов.

### Многомерные массивы

Для создания многомерных массивов используются дополнительные скобки:

```
int[][] a = {
    { 1, 2, 3 },
    { 4, 5, 6 }
}
```

Также массив может создаваться ключевым словом **new**:

```
// трехмерный массив фиксированной длины
int[][][] b = new int[2][4][4];
```

### Двумерный массив

Двумерный массив - это массив одномерных массивов. Если вам нужен двумерный массив, то используйте пару квадратных скобок:

```
String[][] arr = new String[4][3];
```

```
arr[0][0] = "1";
arr[0][1] = "Васька";
arr[0][2] = "121987102";

arr[1][0] = "2";
arr[1][1] = "Рыжик";
arr[1][2] = "2819876107";
```

```
arr[2][0] = "3";
arr[2][1] = "Барсик";
arr[2][2] = "412345678";
```

```
arr[3][0] = "4";
arr[3][1] = "Мурзик";
arr[3][2] = "587654321";
```

Представляйте двумерный массив как таблицу, где первые скобки отвечают за ряды, а вторые - за колонки таблицы. Тогда пример выше представляют собой таблицу из четырёх рядов и трёх колонок.

1	Васька	121987102
2	Рыжик	2819876107
3	Барсик	412345678
4	Мурзик	587654321

Для двумерных массивов часто используются два цикла **for**, чтобы заполнить элементы данными слева направо и сверху вниз. Напишем такой код:

```
int[][] twoD = new int[3][4]; // объявили двухмерный массив
int i, j, k = 0;

for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++) {
        twoD[i][j] = k;
        k++;
    }

for (i = 0; i < 3; i++) {
```



```

        for (j = 0; j < 4; j++)
            System.out.println(twoD[i][j] + " ");
        System.out.println("\n");
    }

```

В данном примере мы сначала заполнили двумерный массив данными, а затем снова прошли по этому массиву для считывания данных.

Логическое представление данного двухмерного массива будет выглядеть следующим образом:

[0, 0]	[0, 1]	[0, 2]	[0, 3]
[1, 0]	[1, 1]	[1, 2]	[1, 3]
[2, 0]	[2, 1]	[2, 2]	[2, 3]

Первое число в скобках обозначают ряд (строку), а второе число - столбец. Принято считать, что в массиве **new int[M][N]** первый размер означает количество строк, а второй - количество столбцов.

На экране после запуска примера мы увидим следующее:

```

0 1 2 3
4 5 6 7
8 9 10 11

```

При резервировании памяти под многомерный массив необходимо указать память только для первого измерения. Для остальных измерений память можно выделить отдельно.

```

int[][] twoD = new int[3][]; // память под первое измерение
// далее резервируем память под второе измерение
twoD[0] = new int[4];
twoD[1] = new int[4];
twoD[2] = new int[4];

```

В данном примере особого смысла в этом нет.

Еще одна интересная особенность при создании массива связана с запятой. Посмотрите на пример.

```

int[][] a = {{1, 2, 3}, {4, 0, 0},,};
System.out.println(Arrays.deepToString(a));

```

Вроде в конце используется лишняя запятая, но её наличие не приведёт к ошибке (только одна запятая). Это бывает удобно, когда надо скопировать или вставить кусок массива в коде. Кстати, метод **deepToString()** класса **Arrays** очень удобен для вывода двухмерных массивов.

Чтобы совсем сбить вас с толку, приведу ещё один правильный пример.

```

Integer[] Integer[] = {{1, 2, 3}, {4, 0, 0},,};
System.out.println(Arrays.deepToString(Integer));

```

Квадратные скобки можно использовать двумя способами. Сначала мы поставили скобки у типа переменной, а потом у имени переменной. При этом мы использовали в качестве имени имя класса **Integer**. Однако, Java догадывается, что на этот раз используется не класс, а имя и разрешает такой синтаксис. Но лучше так не выпендриваться.

### Размер имеет значение

Размер двумерного массива измеряется интересным способом. Длина массива определяется по его первой размерности, то есть вычисляется количество рядов.

```
int[][] matrix = new int[4][5];
System.out.println(matrix.length);
```

А если мы хотим узнать количество столбцов в ряду? Тогда указываете ряд, а затем вычисляете у него количество столбцов.

```
// число колонок у третьего ряда
System.out.println(matrix[2].length);
```

Не забывайте, что в массивах ряды могут содержать разное количество столбцов.

### Сложить два массива

Предположим, у вас есть два массива, и вам нужно их соединить и получить общий массив.

```
private double[] concatArray(double[] a, double[] b) {
    if (a == null)
        return b;
    if (b == null)
        return a;
    double[] r = new double[a.length + b.length];
    System.arraycopy(a, 0, r, 0, a.length);
    System.arraycopy(b, 0, r, a.length, b.length);
    return r;
}
```

Вместо типа *double* вы можете использовать другие типы. Вот например, пример сложения двух строковых массивов:

```
// метод для склеивания двух строковых массивов
private String[] concatArray(String[] a, String[] b) {
    if (a == null)
        return b;
    if (b == null)
        return a;
    String[] r = new String[a.length + b.length];
    System.arraycopy(a, 0, r, 0, a.length);
    System.arraycopy(b, 0, r, a.length, b.length);
    return r;
}
```

```
String[] week1 = new String[] { "Понедельник", "Вторник", "Среда" };
```

```
String[] week2 = new String[] { "Четверг", "Пятница", "Суббота",
    "Воскресенье" };
```

```
String[] week = concatArray(week1, week2); // будет возвращён массив всех
семи дней недели
```

### Взять часть массива

Аналогично, если вам нужно взять только часть из большого массива, то воспользуйтесь методом:

```
// start - с какой позиции нужно получить новый массив, отсчёт с 0
private double[] copyPartArray(double[] a, int start) {
    if (a == null)
        return null;
    if (start > a.length)
        return null;
    double[] r = new double[a.length - start];
    System.arraycopy(a, start, r, 0, a.length - start);
    return r;
}

public void onClick(View v) {
    double[] digits = new double[] {6.5, 3.1, 5.72};
    double[] part = copyPartArray(digits, 1);
    Toast.makeText(v.getContext(), part[1] + "",
Toast.LENGTH_LONG).show();
}
```

Здесь вы также можете использовать другие типы вместо *double*. Вот пример использования:

```
private String[] copyPartArray(String[] a, int start) {
    if (a == null)
        return null;
    if (start > a.length)
        return null;
    String[] r = new String[a.length - start];
    System.arraycopy(a, start, r, 0, a.length - start);
    return r;
}

public void onClick(View v) {
    String[] weekday = new String[] { "Понедельник", "Вторник", "Среда"
};

    // нам нужен массив со второго элемента
    String[] week = copyPartArray(weekday, 1); // вернёт Вторник и Среда
    // выводим второй элемент из полученного массива, т.е. Среда
    Toast.makeText(v.getContext(), week[1], Toast.LENGTH_LONG).show();
}
```

### Перемешать элементы массива

Бывает необходимость перемешать элементы массива в случайном порядке. В интернете нашёл готовый метод по алгоритму Fisher–Yates (прим.: Fisher - это рыбак, который ловит рыбу, например, кот)

```
// Implementing Fisher-Yates shuffle
static void shuffleArray(int[] ar) {
    Random rnd = new Random();
    for (int i = ar.length - 1; i > 0; i--) {
```

```

        int index = rnd.nextInt(i + 1);
        // Simple swap
        int a = ar[index];
        ar[index] = ar[i];
        ar[i] = a;
    }
}

// создадим массив и перемешаем его
int[] mSolutionArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    13, 14 };
shuffleArray(mSolutionArray);

Log.i("Array", Arrays.toString(mSolutionArray));

```

Помните, что размер массива фиксируется и не может меняться на протяжении его жизненного цикла. Если вам нужно изменять, то используйте **ArrayList**, который способен автоматически выделять дополнительное пространство, выделяя новый блок памяти и перемещая в него ссылки из старого.

При выходе за границу массива происходит исключение **RuntimeException**, свидетельствующее об ошибке программиста.

### Метод **arraycopy()** - Копирование массива

Стандартная библиотека Java содержит статический метод **System.arraycopy()**, который копирует массивы значительно быстрее, чем при ручном копировании в цикле **for**.

В аргументах **arraycopy()** передается исходный массив, начальная позиция копирования в исходном массиве, приёмный массив, начальная позиция копирования в приёмном массиве и количество копируемых элементов. Любое нарушение границ массива приведет к исключению.

Разработчик Avi Yehuda написал программу, которая вычисляет время на копирование с помощью цикла **for** и с помощью метода **arraycopy()** на примере с миллионом элементов. Ручное копирование у него заняло 182 мс, с помощью метода **arraycopy()** - 12 мс. Разница колоссальна.

```

public class ArrayCopyTestActivity extends Activity {
    private static final int SIZE_OF_ARRAY = 1000000;
    private long time;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Integer [] sourceArray = new Integer[SIZE_OF_ARRAY];
        Integer [] dst = new Integer[SIZE_OF_ARRAY];
        fillArray(sourceArray);

        startBenchmark();
        naiveCopy(sourceArray, dst);
    }
}

```

```

        stopBenchmark();

        startBenchmark();
        System.arraycopy(sourceArray, 0, dst, 0, sourceArray.length);
        stopBenchmark();
    }

    private void naiveCopy(Integer [] src, Integer [] dst) {
        for (int i = 0; i < src.length; i++) {
            dst[i]=src[i];
        }
    }

    private void fillArray(Integer [] src) {
        for (int i = 0; i < src.length; i++) {
            src[i]=i;
        }
    }

    private void startBenchmark() {
        time = System.currentTimeMillis();
    }

    private void stopBenchmark() {
        time = System.currentTimeMillis() - time;
        Log.d("array copy test", "time="+time);
    }
}

```

## Поиск элементов строкового массива по начальным символам

Допустим, у нас есть строковый массив и нам нужно по первым символам найти все слова, которые входят в данный массив.

```

// Сам метод
public static ArrayList<String> searchFromStart(String[] inputArray,
String searchText) {
    ArrayList<String> outputArray = new ArrayList<>();

    for (int i = 0; i < inputArray.length; i++) {
        if (searchText.compareToIgnoreCase(inputArray[i].substring(0,
            searchText.length())) == 0) {
            outputArray.add(inputArray[i]);
        }
    }
    return outputArray;
}

// Массив строк
final String[] catNamesArray = new String[] { "Рыжик", "Барсик", "Мурзик",
    "Мурка", "Васька", "Томасина", "Бобик", "Кристина", "Пушок",
    "Дымка", "Кузя", "Китти", "Барбос", "Масяня", "Симба" };

// Применим метод. Ищем по буквам "мур":
List<String> words = searchFromStart(catNamesArray, "мур");
Toast.makeText(getApplicationContext(), words.get(0).toString() + ":" +
words.get(1).toString(), Toast.LENGTH_SHORT).show();

```

Вернёт списочный массив из двух элементов: Мурзик и Мурка.

## Класс Arrays

Класс **java.util.Arrays** предназначен для работы с массивами. Он содержит удобные методы для работы с целыми массивами:

- ✓ **copyOf()** – предназначен для копирования массива
- ✓ **copyOfRange()** – копирует часть массива
- ✓ **toString()** – позволяет получить все элементы в виде одной строки
- ✓ **sort()** — сортирует массив методом quick sort
- ✓ **binarySearch()** – ищет элемент методом бинарного поиска
- ✓ **fill()** – заполняет массив переданным значением (удобно использовать, если нам необходимо значение по умолчанию для массива)
- ✓ **equals()** – проверяет на идентичность массивы
- ✓ **deepEquals()** – проверяет на идентичность массивы массивов
- ✓ **asList()** – возвращает массив как коллекцию

## Сортировка массива

Сортировка (упорядочение по значениям) массива **a** производится методами **Arrays.sort(a)** и **Arrays.sort(a, index1, index2)**. Первый метод упорядочивает в порядке возрастания весь массив, второй — часть элементов (от индекса **index1** до индекса **index2**). Имеются и более сложные методы сортировки. Элементы массива должны быть сравнимы (поддерживать операцию сравнения).

Простой пример

```
// задаём числа в случайном порядке
int[] numbers = new int[]{1, 23, 3, 8, 2, 4, 4};
// сортируем
Arrays.sort(numbers);
// проверяем
mInfoTextView.setText(Arrays.toString(numbers));
```

## Сортировка массива для ArrayAdapter

Массивы часто используются в адаптерах для заполнения данными компоненты **Spinner**, **ListView** и т.п.

Предположим, у вас есть массив строк и его нужно отсортировать перед отдачей массива адаптеру **ArrayAdapter**. Это позволит вывести строки в упорядоченном виде, например, в **ListView**:

```
String[] catsNames = {
    "Васька",
    "Кузя",
    "Барсик",
    "Мурзик",
    "Леопольд",
    "Бегемот",
    "Рыжик",
    "Матроскин"
};

// Сортируем перед передачей адаптеру
```

```
Arrays.sort(catsNames);

ArrayAdapter<String> adapter;

adapter = new ArrayAdapter<>(
    this,
    android.R.layout.simple_list_item_1,
    catsNames);

setListAdapter(adapter);
```

У метода **sort()** есть перегруженные версии, где можно указать диапазон массива, в пределах которого следует произвести сортировку.

## Копирование массивов

Метод **Arrays.copyOf(оригинальный\_массив, новая\_длина)** — возвращает массив-копию новой длины. Если новая длина меньше оригинальной, то массив усекается до этой длины, а если больше, то дополняется нулями.

```
// первый массив
int[] anyNumbers = {2, 8, 11};
// копия второго массива
int[] luckyNumbers = Arrays.copyOf(anyNumbers, anyNumbers.length);
luckyNumbers[2] = 25;
mInfoTextView.setText("anyNumbers: " + Arrays.toString(anyNumbers)
    + "\nluckyNumbers: " + Arrays.toString(luckyNumbers));
```

Теперь первый массив останется без изменений, а со вторым массивом делайте что хотите. Смотрим на результат.

```
anyNumbers: [2, 8, 11];
luckyNumbers: [2, 8, 25];
```

Можно создать увеличенную копию, когда копируются все значения из маленького массива, а оставшиеся места заполняются начальными значениями, например, нулями.

```
// три элемента
int[] small_array = {1, 2, 3};
// создадим массив с пятью элементами
int[] big_array = Arrays.copyOf(small_array, 5);
mInfoTextView.setText("big_array: " + Arrays.toString(big_array));
```

Получим результат:

```
big_array: [1, 2, 3, 0, 0]
```

Метод **Arrays.copyOfRange(оригинальный\_массив, начальный\_индекс, конечный\_индекс)** — также возвращает массив-копию новой длины, при этом копируется часть оригинального массива от начального индекса до конечного -1.

```
// Массив из четырех элементов
String[] array_1 = {
    "Васька",
    "Мурзик",
    "Рыжик",
    "Барсик"};
// Сортировка массива
```

```

Arrays.sort(array_1);
// Копируем первые три элемента массива во второй массив
String[] array_2 = Arrays.copyOf(array_1, 3);
// Копируем нужные элементы из первого массива
// в диапазоне от второго элемента до последнего в третий массив
String[] array_3 = Arrays.copyOfRange(array_1,
    2, array_1.length);

Log.i(TAG, Arrays.toString(array_1));
Log.i(TAG, Arrays.toString(array_2));
Log.i(TAG, Arrays.toString(array_3));

```

## Метод Arrays.toString()

Если использовать вызов метода **toString()** непосредственно у массива, то получите что-то непонятное и нечитаемое.

```

String[] catNames = {
    "Баська",
    "Мурзик",
    "Рыжик",
    "Барсик"};
Log.i(TAG, catNames.toString());

// Вернёт
[Ljava.lang.String;@4222bd88

```

Метод **Arrays.toString(массив)** возвращает строковое представление массива со строковым представлением элементов, заключенных в квадратные скобки. В примерах выше мы уже вызывали данный метод.

Метод **deepToString()** удобен для вывода многомерных массивов. Этот метод мы также уже использовали выше.

## Метод Arrays.fill() - наполнение массива одинаковыми данными

Метод **Arrays.fill()** позволяет быстро заполнить массив одинаковыми значениями. У метода есть восемнадцать перегруженных версий для разных типов и объектов.

Метод **fill()** просто дублирует одно заданное значение в каждом элементе массива (в случае объектов копирует одну ссылку в каждый элемент):

```

int size = 4;
boolean[] test1 = new boolean[size];
int[] test2 = new int[size];
String[] test3 = new String[size];
Arrays.fill(test1, true); // присваиваем всем true
Toast.makeText(getApplicationContext(), Arrays.toString(test1),
    Toast.LENGTH_LONG).show();
Arrays.fill(test2, 9); // присваиваем всем 9
Toast.makeText(getApplicationContext(), Arrays.toString(test2),
    Toast.LENGTH_LONG).show();
Arrays.fill(test3, "Мяу!"); // Ну вы поняли
Toast.makeText(getApplicationContext(), Arrays.toString(test3),
    Toast.LENGTH_LONG).show();

```

Запустив код, вы увидите, что на экране по очереди выводятся значения:

```

[true, true, true, true]
[9, 9, 9, 9]
[Мяу!, Мяу!, Мяу!, Мяу!]

```



Можно заполнить данными в нужном интервале за два прохода:

```
int size = 4;
String[] test3 = new String[size];
Arrays.fill(test3, "Мяу! ");
Arrays.fill(test3, 2, 3, "Гав!");
Toast.makeText(getApplicationContext(), Arrays.toString(test3),
    Toast.LENGTH_LONG).show();
```

Сначала массив заполнится мяуканьем кота 4 раза, а потом на третью позицию попадает слово **Гав!**:

```
[Мяу!, Мяу!, Гав!, Мяу!]
```

Как видите, метод заполняет весь массив, либо диапазон его элементов. Но получаемые одинаковые данные не слишком интересны для опытов, но пригодятся для быстрых экспериментов.

### Метод equals() - сравнение массивов

Класс **Arrays** содержит метод **equals()** для проверки на равенство целых массивов. Чтобы два массива считались равными, они должны содержать одинаковое количество элементов, и каждый элемент должен быть эквивалентен соответствующему элементу другого массива.

Напишем код в своей учебной программе.

```
// Создаем два массива
int[] a1 = new int[10];
int[] a2 = new int[10];
// заполняем их девятками
Arrays.fill(a1, 9);
Arrays.fill(a2, 9);
mInfoTextView.setText("Сравним: " + Arrays.equals(a1, a2));
```

Мы создали два массива и заполнили их одинаковыми числами. При сравнении мы получим **true**. Добавим в код строчку кода, которая заменит один элемент во втором массиве:

```
//Изменим один элемент у второго массива
a2[3] = 11;
mInfoTextView.setText("Сравним: " + Arrays.equals(a1, a2));
```

Теперь при сравнении будет выдаваться **false**.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_  
Фамилия, инициалы учащегося \_\_\_\_\_  
Дата выполнения работы \_\_\_\_\_  
Тема работы: \_\_\_\_\_  
Цель работы: \_\_\_\_\_  
Оснащение работы: \_\_\_\_\_  
Результат выполнения работы: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

### **7. Контрольные вопросы и задания**

1. Дайте определение массиву?
2. Как объявить массив?
3. Как объявить многомерный массив?
4. Как обратиться к элементу массива?
5. Область применения массивов?

### **8. Рекомендуемая литература**

19. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
20. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
21. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
22. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
23. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
24. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка классов и использование их в программах»

Минск  
2018

## Лабораторная работа № 5

### Тема работы: «Разработка классов и использование их в программах»

#### 1. Цель работы

Закрепить навык создания и применения классов при решении различных задач.

#### 2. Задание

Два задания обязательны для выполнения.

В первом задании необходимо произвести проектирование классов по теме вашего варианта. Нужно описать:

- ✓ не менее 3 классов (имя класса – существительное);
- В каждом классе реализовать:
  - ✓ не менее 4 полей (к каждому полю соответствуют нужные свойства);
  - ✓ не менее 2 конструкторов (без параметров и со всеми параметрами);
  - ✓ не менее 3 методов.

Также привести по одному примеру (объекту) каждого класса.

Во втором задании необходимо реализовать класс по условию варианта, заготовить несколько контрольных примеров. Организовать проверку существования необходимой фигуры.

Номер варианта соответствует вашему номеру по списку.

*Примечание: начиная с 21 учащегося в списке необходимо определить свой вариант следующим образом:  $N \% 20$ , где  $N$  – ваш номер по списку,  $\%$  - остаток от деления.*

#### Задание 1: проектирование класса

Вариант 1. Здравоохранение.

Вариант 2. Защита человека от воздействия окружающей среды. Безопасность.

Вариант 3. Метрология и измерения.

Вариант 4. Машиностроение.

Вариант 5. Электроника.

Вариант 6. Телекоммуникации. Аудио- и видеотехника.

Вариант 7. Информационные технологии.

Вариант 8. Офисное оборудование.

Вариант 9. Технология получения изображений.

Вариант 10. Дорожно-транспортная техника.

Вариант 11. Железнодорожная техника.

- Вариант 12. Судостроение и морские сооружения.
- Вариант 13. Авиационная и космическая техника.
- Вариант 14. Сельское хозяйство.
- Вариант 15. Производство пищевых продуктов.
- Вариант 16. Химическая промышленность.
- Вариант 17. Горное дело и полезные ископаемые.
- Вариант 18. Добыча и переработка нефти, газа и смежные производства.
- Вариант 19. Металлургия.
- Вариант 20. Гражданское строительство.
- Вариант 21. Военная техника.
- Вариант 22. Бытовая техника и торговое оборудование.
- Вариант 23. Образование, среднее полное общее образование.
- Вариант 24. Образование, высшее профессиональное образование.
- Вариант 25. Библиотечное дело.
- Вариант 26. Телекоммуникации.
- Вариант 27. Компьютерные сети.
- Вариант 28. Гражданская техника.
- Вариант 29. Космическая техника.
- Вариант 30. Химическое производство.

## Задание 2: создание класса

### **Вариант 1**

Класс Треугольник

Поля: три стороны

Операции:

- увеличение/уменьшение размера сторон в заданное количество раз;
- вычисление периметра;
- вычисление площади;
- определение значений углов.

### **Вариант 2.**

Класс Треугольник

Поля: три стороны

Операции:

- увеличение/уменьшение размера сторон на заданное количество процентов;
- вычисление средней линии для любой из сторон;
- определение вида треугольника по величине углов (Остроугольный, Тупоугольный, Прямоугольный);
- определение значений углов.

### **Вариант 3.**

Класс Треугольник

Поля: две стороны и угол между ними

Операции:

увеличение/уменьшение размера угла на заданное количество процентов;

определение вида треугольника по числу равных сторон (Разносторонний, Равнобедренный, Равносторонний);

определение расстояния между центрами вписанной и описанной окружностей.

определение значений углов.

### **Вариант 4.**

Класс Треугольник

Поля: две стороны и угол между ними

Операции:

уменьшение/увеличение размера угла (из полей) в заданное количество раз;

вычисление длины биссектрисы принадлежащей любому углу;

вычисление длин отрезков, на которые биссектриса делит любую сторону;

определение значений углов.

### **Вариант 5.**

Класс Треугольник

Поля: сторона и два прилежащих к ней угла

Операции:

уменьшение/увеличение размера стороны (из полей) в заданное количество раз;

вычисление длины медианы, принадлежащей любой стороне;

определение подобен ли другой треугольник данному (указанному по индексу массива);

определение значений сторон.

### **Вариант 6.**

Класс Треугольник

Поля: сторона и два прилежащих к ней угла

Операции:

увеличение/уменьшение значения любого угла (из полей) на заданное количество процентов;  
вычисление длины высот, принадлежащей любой стороне;  
определение значений сторон.

#### **Вариант 7.**

Класс Прямоугольный треугольник

Поля: две стороны

Операции:

увеличение/уменьшение размера любой стороны (из полей) на заданное количество процентов;  
вычисление радиуса описанной окружности;  
вычисление полупериметра;  
определение значений углов.

#### **Вариант 8.**

Класс Прямоугольный треугольник

Поля: сторона и угол

Операции:

уменьшение/увеличение размера любой стороны (из полей) на заданный процент;  
вычисление радиуса вписанной окружности;  
определение расстояния между центрами вписанной и описанной окружностей;  
вычисление квадратного корня из площади;  
определение значений сторон.

#### **Вариант 9.**

Класс Равнобедренный треугольник

Поля: основание и боковая сторона

Операции: увеличение/уменьшение размера на определенный процент;

вычисление длины медианы, принадлежащей любой стороне;

вычисление периметра и площади;

определение значений углов.

#### **Вариант 10.**

Класс Равнобедренный треугольник

Поля: боковая сторона и угол при основании

Операции:

увеличение/уменьшение размера в заданное количество раз;

вычисление длины биссектрисы принадлежащей любому углу;  
вычисление длины высот, принадлежащей любой стороне;  
определение значений сторон.

### **Вариант 11.**

Класс Параллелограмм

Поля: две стороны и угол между ними

Операции:

увеличение/уменьшение размера любой из сторон (из полей) на определенный процент;

вычисление периметра и площади;

вычисление диагоналей;

вычисление высоты.

### **Вариант 12.**

Класс Параллелограмм

Поля: две стороны и диагональ (прилегающая к ним так, что бы образовать треугольник)

Операции:

увеличение/уменьшение размера в заданное количество раз;

вычисление квадратного корня из периметра и площади;

вычисление диагонали и стороны;

вычисление высоты.

### **Вариант 13.**

Класс Прямоугольник

Поля: две стороны

Операции:

увеличение/уменьшение размера любой из сторон на определенный процент;

вычисление периметра и площади;

вычисление диагонали.

### **Вариант 14.**

Класс Квадрат

Поля: Сторона

Операции:

увеличение/уменьшение размера на определенный процент;

вычисление периметра и площади;

вычисление диагонали.



**Вариант 15.**

Класс Ромб

Поля: сторона и диагональ (меньшая)

Операции:

увеличение/уменьшение размера на определенный процент;  
вычисление периметра и площади;  
вычисление высоты.

**Вариант 16.**

Класс Трапеция

Поля: четыре стороны

Операции:

увеличение/уменьшение размера в заданное количество раз;  
вычисление периметра и площади;  
определение подобна ли другая трапеция данной (указанной по индексу массива);  
определение размера средней линии;  
вычисление высоты.

**Вариант 17.**

Класс Окружность

Поля: радиус

Операции:

увеличение/уменьшение размера на определенный процент;  
вычисление длины окружности и площади круга;  
определение диаметра.

**Вариант 18.**

Класс Сегмент окружности

Поля: хорда и высота сегмента

Операции:

увеличение/уменьшение размера в заданное количество раз;  
вычисление площади;  
определение длины дуги;  
вычисление длины окружности и ее диаметра.

**Вариант 19.**

Класс Сектор окружности

Поля: радиус и центральный угол

Операции:

увеличение/уменьшение размера в заданное количество раз;

вычисление площади;  
определение длины дуги;  
вычисление длины окружности и ее диаметра.

### **Вариант 20.**

Класс Круговое кольцо

Поля: внешний и внутренний диаметр

Операции:

увеличение/уменьшение размера в заданное количество раз;

вычисление площади;

вычисление среднего радиуса;

вычисление толщины кольца.

## **3. Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

## **4. Основные теоретические сведения**

### **Классы**

Java позволяет создавать классы, которые представляют объекты из реального мира. Например, можно создать класс **Car** (автомобиль) или **Animal** (животное) и задать им различные свойства. Для класса **Car** логично создать такие свойства как двери, колёса, лобовое стекло и т.д. Имея класс **Car**, можно создать новые классы Легковушки, Грузовики, Автобусы, которые будут иметь все свойства класса **Car**, а также свои собственные свойства. У класса **Animal** соответственно можно задать свойства Лапы, Хвост, а затем создать наш любимый класс **Cat**, у которого будет ещё дополнительное свойство Усы. Иными словами, классы могут наследовать свойства от других классов. Родительский класс называется суперклассом. Внутри классов могут быть объявлены поля и методы.

Для объявления класса служит ключевое слово **class**. Вспомним стандартную строчку кода из Android-проекта:

```
public class MainActivity extends Activity {
    // код внутри класса
}
```

Упрощённая общая форма для класса может иметь следующий вид:

```
class ИмяКласса {
    тип переменная_экземпляра1;

    тип имяМетода(список параметров) {
        // тело метода
    }
}
```

В Java принято начинать имена класса с большой буквы. В классе могут быть несколько переменных и методов. Переменные, определённые внутри класса (не метода), называются переменными экземпляра или полями (fields). Код пишется внутри класса. Методы и переменные внутри класса являются членами класса.

## Объекты

Новый объект (или экземпляр) создаётся из существующего класса при помощи ключевого слова **new**:

```
Cat barsik = new Cat(); // создали кота из класса Cat
```

В большинстве случаев вы будете использовать такой способ. Пусть вас не удивляет, что приходится дважды использовать слово **Cat**, оно имеет разный смысл.

Слева от оператора присваивания = определяется имя переменной и его тип **Cat**. В правой части выражения происходит выделение памяти для нового экземпляра класса **Cat** и инициализируется экземпляр. Оператор присваивания присваивает переменной ссылку на только что созданный объект. Имена объектов не нужно начинать с большой буквы, как у класса. Так вы будете различать, где класс, а где экземпляр класса. Если имя экземпляра класса состоит из нескольких слов, то используется верблюжья нотация, когда все первые буквы слов, кроме первой, пишутся с большой - superBlackCat.

Если вы помните, при объявлении примитивных типов мы указывали нужный тип в самом начале.

```
int catAge;
```

Поэтому код **Cat barsik** также определяет его тип. Он не всегда может совпадать с именем класса.

```
Pet barsik = new Cat();
```

В этом примере используется тип класса домашних любимцев **Pet**, а обращаемся к классу котов **Cat**.

Теперь подробнее.

Простой пример создания класса **Box** (коробка для кота):

```
class Box {  
    int width; // ширина коробки  
    int height; // высота коробки  
    int depth; // глубина коробки  
}
```

При таком варианте Java автоматически присвоит переменным значения по умолчанию. Например, для **int** это будет значение 0. Но не всегда значения по умолчанию подойдут в вашем классе. Если вы создали переменную для описания количества лап у кота, то логично сразу присвоить значение 4. Поэтому считается хорошей практикой сразу присваивать нужные значения полям класса, не полагаясь на систему.

Вам нужно создать отдельный файл **Box.java**, в который следует вставить код, описанный выше. О том, как создавать новый файл для класса я не буду здесь расписывать.

Сам класс - это просто шаблон, заготовка. Чтобы ваше приложение могло использовать данный шаблон, нужно создать на его основе объект при помощи ключевого слова **new**:

```
Box catBox = new Box; // создали реальный объект с  
именем catBox на основе шаблона Box
```

Объект **catBox**, объявленный в коде вашей программы, сразу займёт часть памяти на устройстве. При этом объект будет содержать собственные копии переменных экземпляра **width**, **height**, **depth**. Для доступа к этим переменным используется точка (.). Если мы хотим присвоить значение переменной **width**, то после создания объекта класса можете написать код:

```
catBox.width = 400; // ширина коробки для кота 400  
миллиметров
```

Если мы хотим вычислить объём коробки, то нужно перемножить все значения размеров коробки:

```
Box catBox = new Box();  
  
catBox.width = 400;  
catBox.height = 200;  
catBox.depth = 250;
```

```
int volume = catBox.width * catBox.height *
catBox.depth;
```

```
mInfoTextView.setText("Объём коробки: " + volume);
```

Каждый объект содержит собственные копии переменных экземпляра. Вы можете создать несколько объектов на основе класса **Box** и присваивать разные значения для размеров коробки. При этом изменения переменных экземпляра одного объекта никак не влияют на переменные экземпляра другого объекта. Давайте объявим два объекта класса **Box**:

```
Box bigBox = new Box(); // большая коробка
Box smallBox = new Box(); // маленькая коробка

int volume;

// присвоим значения переменным для большой коробки
bigBox.width = 400;
bigBox.height = 200;
bigBox.depth = 250;

// присвоим значения переменным для маленькой коробки
smallBox.width = 200;
smallBox.height = 100;
smallBox.depth = 150;

// вычисляем объём первой коробки
volume = bigBox.width * bigBox.height * bigBox.depth;
mInfoTextView.setText("Объём большой коробки: " +
volume + "\n");

// вычисляем объём маленькой коробки
volume = smallBox.width * smallBox.height *
smallBox.depth;
mInfoTextView.append("Объём маленькой коробки: " +
volume);
```

Когда мы используем конструкцию типа **Box bigBox = new Box();**, то в одной строке выполняем сразу два действия - объявляем переменную типа класса и резервируем память под объект. Можно разбить конструкцию на отдельные части:

```
Box bigBox; // объявляем ссылку на объект
bigBox = new Box(); // резервируем память для объекта
Box
```

Обычно такую конструкцию из двух строк кода не используют на практике, если нет особых причин.

Когда мы используем ключевое слово **new** и указываем имя класса, то после имени ставим круглые скобки, которые указывают на конструктор класса. О них поговорим позже.

### Ключевое слово **final**

Поле может быть объявлено как **final** (финальное). Это позволяет предотвратить изменение содержимого переменной, по сути, это становится константой. Финальное поле должно быть инициализировано во время его первого объявления.

```
final int FILE_OPEN = 1;
```

Теперь можно пользоваться переменной **FILE\_OPEN** так, как если бы она была константой, без риска изменения их значений. Принято записывать имена заглавными буквами.

Кроме полей, **final** можно использовать для параметров метода (препятствует изменению в пределах метода) и у локальных переменных (препятствует присвоению ей значения более одного раза).

Также слово **final** можно применять к методам, чтобы предотвратить его переопределение.

```
class Cat {
    final void meow() {
        System.out.println("Мяу");
    }
}

class Kittent extends Cat {
    // Этот метод создать не получится
    void meow() {
        System.out.println("Да хоть гав-гав, всё равно не заведётся");
    }
}
```

Ещё один вариант использования ключевого слова **final** - предотвращение наследования класса. При этом неявно все методы класса также становятся финальными. Поэтому нельзя одновременно объявить класс абстрактным и финальным, поскольку абстрактный класс является лишь шаблоном и только его подклассы реализуют методы.

```
final class Tail {
    // ...
}
```

```
// Следующий класс недопустим
class BigTail extends Tail {
    // Ошибка! Класс Хвост нельзя переопределять.
}
```

### Ключевое слово **instanceof** - Проверка принадлежности к классу

Иногда требуется проверить, к какому классу принадлежит объект. Это можно сделать при помощи ключевого слова **instanceof**. Это булев оператор, и выражение **foo instanceof Foo** истинно, если объект **foo** принадлежит классу **Foo** или его наследнику, или реализует интерфейс **Foo** (или, в общем виде, наследует класс, который реализует интерфейс, который наследует **Foo**).

Возьмём пример с рыбками, которые знакомы котам не понаслышке. Пусть у нас есть родительский класс **Fish** и у него есть унаследованные подклассы **SaltwaterFish** и **FreshwaterFish**. Мы можем протестировать, относится ли заданный объект к классу или подклассу по имени

```
SaltwaterFish nemo = new SaltwaterFish();
if(nemo instanceof Fish) {
    // рыбка Немо относится к классу Fish
    // это может быть класс Fish (родительский класс)
или подкласс типа
    // SaltwaterFish или FreshwaterFish.

    if(nemo instanceof SaltwaterFish) {
        // Немо - это морская рыбка!
    }
}
```

Данная проверка удобна во многих случаях. В Android очень много классов, которые происходят от класса **View** - **TextView**, **CheckBox**, **Button**, имеющие свои собственные наборы свойств. И если имеется метод с параметром **View**, то при помощи **instanceof** можно разделить логику кода:

```
void checkforTextView(View view)
{
    if(view instanceof TextView)
    {
        // Код для элемента TextView
    } else {
        // Для других элементов View
    }
}
```

**import** - Импорт класса

Оператор `import` сообщает компилятору Java, где найти классы, на которые ссылается код. Любой сложный объект использует другие объекты для выполнения тех или иных функций, и оператор импорта позволяет сообщить о них компилятору Java. Оператор импорта обычно выглядит так:

```
import ClassNameToImport;
```

За ключевым словом **import** следуют класс, который нужно импортировать, и точка с запятой. Имя класса должно быть полным, то есть включать свой пакет. Чтобы импортировать все классы из пакета, после имени пакета можно поместить `.*`.

```
ru.alexanderklimov.MyClass.Cat.sayMeow();  
java.lang.System.out.println("Мяу");
```

### Статический импорт

Существует ещё статический импорт, применяемый для импорта статических членов класса или интерфейса. Это позволяет сократить количество кода. Например, есть статические методы **Math.pow()**, **Math.sqrt()**. Для вычислений сложных формул с использованием математических методов, код становится перегружен. К примеру, вычислим гипотенузу.

```
hypot = Math.sqrt(Math.pow(side1, 2) +  
Math.pow(side2, 2));
```

В данном случае без указания класса не обойтись, так как методы статические. Чтобы не набирать имена классов, их можно импортировать следующим образом:

```
import static java.lang.Math.sqrt;  
import static java.lang.Math.pow;
```

```
...
```

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

После импорта уже нет необходимости указывать имя класса.

Второй допустимый вариант, позволяющий сделать видимыми все статические методы класса:

```
import static java.lang.Math.*;
```

В этом случае вам не нужно импортировать отдельные методы. Но такой подход в Android не рекомендуется, так как требует больше памяти.

### Класс Class

На первый взгляд, класс **Class** звучит как "масло масляное". Тем не менее, класс с таким именем существует и он очень полезен.



Иногда из программы нужно получить имя используемого класса. Для этого есть специальные методы **getClass().getName()** и другие родственные методы. Допустим, нам нужно узнать имя класса кнопки, на которую мы нажимаем в программе.

```
public void onClick(View view) {
    String className = view.getClass().getName();
    String simpleName =
view.getClass().getSimpleName();
    String canonicalName =
view.getClass().getCanonicalName();

    if (canonicalName == null) {
        canonicalName = "null";
    }

    String s = "Имя класса: " + className + "\n" +
"SimpleName: " + simpleName
        + "\n" + "CanonicalName: " +
canonicalName + "\n";

    System.out.println(s);
}
```

Если нужно узнать имя класса активности, то достаточно кода:

```
// подставьте имя вашей активности
String className = MainActivity.class.getName();
Если вам известно имя класса, то можете получить сам класс:

try {
    // получим объект Class
    Class<?> myClass =
Class.forName("ru.alexanderklimov.test.MainActivity");
    mInfoTextView.setText(myClass.getName()); //
выводим в TextView

    Intent intent = new Intent(this, myClass);
    startActivity(intent);
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Метод **getSuperclass()** возвращает имя суперкласса. Остальные несколько десятков методов не столь популярны.

### **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

### **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

### **7. Контрольные вопросы и задания**

1. Как создать класс?
2. Как создать поля класса?
3. Модификаторы доступа полей?
4. Модификатор доступа у класса по умолчанию?

### **8. Рекомендуемая литература**

25. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
26. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
27. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с

- 28.Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
- 29.Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. –240с.
- 30.Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка методов и использование их в программах»

Минск  
2018

## Лабораторная работа № 6

### Тема работы: «Разработка методов и использование их в программах»

#### 1. Цель работы

Закрепить навык создание методов и конструкторов, для эффективного решения задачи.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо реализовать классы из задания 1 лабораторной работы №5, и в каждом классе реализовать:

- ✓ не менее 4 полей (к каждому полю соответствуют нужные свойства);
- ✓ не менее 2 конструкторов (без параметров и со всеми параметрами);
- ✓ не менее 4 методов (с возвращающим значением и без возвращающего значения).

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

Класс может содержать методы - один, два, три и больше в зависимости от сложности класса. Название метода всегда завершается двумя круглыми скобками, после которых идет блок кода, обрамлённый фигурными скобками. Например, метод **sayMeow()** класса **Cat** выводит мяуканье кошки. Внутри имени метода могут быть параметры, например, **sayMeow(3)** - кошка мяукает три раза. Параметров может быть несколько, тогда они разделяются запятыми.

Общая форма объявления метода выглядит следующим образом:

```
модификатор тип имяМетода (список_параметров) {  
    // тело метода  
}
```

Метод может не иметь параметров, в этом случае используются пустые скобки. Модификатор определяет видимость метода (**public**, **private**). Если модификатор не указан, то считается, что метод имеет модификатор **private** в пределах своего пакета.

Методы могут вызывать другие методы.

Каждый метод начинается со строки объявления внутри круглых скобок, которую называют сигнатурой метода:

```
public static void sayMeow(int count) {  
    // здесь ваш код  
}
```

Если рассмотреть данный метод, то можно сказать следующее. Ключевое слово **public** означает, что метод доступен для любого класса. Ключевое слово **static** означает, что нам не нужно создавать экземпляр (копию) объекта **Cat** в памяти. Ключевое слово **void** означает, что метод не возвращает никаких данных. Именем метода является слово перед круглыми скобками.

Если метод возвращает какие-то данные, то в теле метода используется оператор **return значение**, где **значение** - это возвращаемое значение. Тогда вместо **void** нужно указать возвращаемый тип.

Вспомним наш класс **Box**, в котором определены высота, ширина и глубина ящика. Зная эти величины, мы вычисляли объём коробки самостоятельно. Но мы можем расширить возможности класса, чтобы он сам мог вычислить объём внутри класса и предоставить нам готовые данные. Давайте добавим в класс новый метод:

```
class Box {  
    int width; // ширина коробки  
    int height; // высота коробки  
    int depth; // глубина коробки  
  
    // вычисляем объём коробки  
    String getVolume() {  
        return "Объём коробки: " + (width * height * depth);  
    }  
}
```

Теперь пробуем вычислить объём коробки с помощью готового метода, который есть в классе:

```
Box catBox = new Box();  
  
catBox.width = 400;  
catBox.height = 200;  
catBox.depth = 250;  
  
mInfoTextView.setText(catBox.getVolume());
```

Мы уже не вычисляем объём вручную, за нас это сделает класс **Box**, у которого есть готовый метод для вычисления объёмов.

Обращение к методу осуществляется как и к переменной через точку. Наличие круглых скобок позволяет различать метод от имени переменной. То есть, если вы увидите запись:

```
cat.getVolume; // это переменная  
cat.getVolume(); // это метод
```

Выше приведён немного искусственный пример, так как опытный программист никогда не назовёт переменную именем **getVolume**. Существует рекомендация, что для методов в начале имени нужно использовать глагол и

начинаться имя должно с маленькой буквы - переменные так называть не следует.

### Использование параметров

Параметры позволяют работать с различными данными. Допустим, мы хотим вычислить площадь прямоугольника со сторонами 3 и 5 см.

```
int getSquare() {  
    return 3 * 5;  
}
```

Метод работает, но область его применения слишком ограничена. Мы сможем вычислять площадь только одного прямоугольника с жёстко заданными размерами. Но прямоугольники бывают разными и нам нужен универсальный метод. Добиться решения задачи можно с помощью параметров. Перепишем метод следующим образом:

```
int getSquare(int a, int b) {  
    return a * b;  
}
```

Теперь мы можем вычислять площадь любого прямоугольника, зная его размеры. Возьмём тот же прямоугольник со сторонами 3 и 5 см и вычислим его площадь:

```
mInfoTextView.setText("Площадь прямоугольника: " + getSquare(3, 5));
```

В правильно написанных классах стараются избегать доступа к переменным класса напрямую. Следует использовать методы, которые позволяют избежать ошибок. В нашем классе **Box** использовались отдельные переменные **width**, **height**, **depth**. Код с использованием этих переменных слишком громоздкий, кроме того вы можете забыть про какую-нибудь переменную. Добавим в класс новый метод, который упростит наш код для вычисления объёма ящика:

```
package ru.alexanderklimov.test;  
  
class Box {  
    int width; // ширина коробки  
    int height; // высота коробки  
    int depth; // глубина коробки  
  
    // вычисляем объём коробки  
    int getVolume() {  
        return width * height * depth;  
    }  
  
    // устанавливаем размеры коробки  
    void setDim(int w, int h, int d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

Пробуем класс в действии:

```
Box catBox = new Box();
catBox.setDim(400, 200, 250);
int vol = catBox.getVolume();

mInfoTextView.setText("Объём ящика: " + vol);
```

Теперь мы не обращаемся к отдельным переменным класса, а вызываем метод **setDim()** для установки размеров ящика, после чего можно вызвать метод **getVolume()** для вычисления объёма. Естественно, вы можете реализовать свою логику в классе. Например, вы можете создать метод **getVolume()** с параметрами, которые отвечают за размеры ящика и возвращать сразу готовый результат.

## Класс Object

В Java есть специальный суперкласс **Object** и все классы являются его подклассами. Поэтому ссылочная переменная класса **Object** может ссылаться на объект любого другого класса. Так как массивы являются тоже классами, то переменная класса **Object** может ссылаться и на любой массив.

```
// применимо к любому классу
Object obj = new Cat("Barsik");
```

В таком виде объект обычно не используют. Чтобы с объектом что-то сделать, нужно выполнить приведение типов.

```
Cat cat = (Cat) obj;
```

У класса есть несколько важных методов.

- ✓ **Object clone()** - создаёт новый объект, не отличающийся от клонируемого
- ✓ **boolean equals(Object obj)** - определяет, равен ли один объект другому
- ✓ **void finalize()** - вызывается перед удалением неиспользуемого объекта
- ✓ **Class<?> getClass()** - получает класс объекта во время выполнения
- ✓ **int hashCode()** - возвращает хеш-код, связанный с вызывающим объектом
- ✓ **void notify()** - возобновляет выполнение потока, который ожидает вызывающего объекта
- ✓ **void notifyAll()** - возобновляет выполнение всех потоков, которые ожидают вызывающего объекта
- ✓ **String toString()** - возвращает строку, описывающий объект
- ✓ **void wait()** - ожидает другого потока выполнения
- ✓ **void wait(long millis)** - ожидает другого потока выполнения
- ✓ **void wait(long millis, int nanos)** - ожидает другого потока выполнения



Методы **getClass()**, **notify()**, **notifyAll()**, **wait()** являются финальными и их нельзя переопределять.

### Метод **hashCode()**

Хеш-код - это целое число, генерируемое на основе конкретного объекта. Его можно рассматривать как шифр с уникальным значением.

Для вычисления хеш-кода в классе **String** применяется следующий алгоритм.

```
int hash = 0;
for(int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

У любого объекта имеется хеш-код, определяемый по умолчанию, который вычисляется по адресу памяти, занимаемой объектом.

Значение хеш-кода возвращает целочисленное значение, в том числе и отрицательное.

Если в вашем классе переопределяется метод **equals()**, то следует переопределить и метод **hashCode()**.

### Метод **toString()**

Очень важный метод, возвращающий значение объекта в виде символьной строки.

Очень часто при использовании метода **toString()** для получения описания объекта можно получить набор бессмысленных символов, например, **[I@421199e8**. На самом деле в них есть смысл, доступный специалисту. Он сразу может сказать, что мы имеем дело с одномерным массивом (одна квадратная скобка), который имеет тип **int** (символ **I**). Остальные символы тоже что-то означают, но вам знать это не обязательно.

Если же вам нужно научное объяснение, то метод работает по следующему алгоритму (из документации).

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Обычно принято переопределять метод, чтобы он выводил результат в читаемом виде.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

Лабораторная работа № \_\_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Определение метода?
2. Метод с возвращаемым значением.
3. Метод без возвращаемого значения.
4. Параметры методов.
5. Как реализовать метод с изменяемым количеством параметров?
6. Механизм перегрузки методов.
7. Для чего необходима перегрузка методов?
8. Какие методы реализованы в классе Object?

## **8. Рекомендуемая литература**

31. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
32. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
33. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
34. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
35. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
36. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Перегрузка методов и передача аргументов»

Минск  
2018

## Лабораторная работа № 7

### Тема работы: «Перегрузка методов и передача аргументов»

#### 1. Цель работы

Закрепить навык перегрузки методов и конструкторов.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо к классам из задания 1 лабораторной работы №6, добавить:

- ✓ не менее 5 перегруженных методов (для тех методов, где позволяет логика);
- ✓ не менее 3 методов с изменяемым числом входных параметров.

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Перегрузка методов

Метод - имя для действия: прыгнуть, мяукнуть, отформатировать диск. Использование имён при написании кода упрощает её понимание и модификацию. Работа разработчика схожа с работой писателя - в обоих случаях требуется донести свою мысль до читателя/приложения.

Часто одно и то же слово имеет несколько разных значений - оно *перегружено*. Например, мы говорим "вымыть посуду" и "вымыть кота". Было бы глупо говорить "посудовымыть посуду" или "котовымыть кота", чтобы подчеркнуть разницу. Также и с методами. Можно создавать методы с одинаковыми именами, но с разным набором аргументов.

*Перегрузку (overloading) следует отличать от замещения (overriding) - иной реализации метода в подклассе первоначально определившего метод класса.*

Java, как и многие языки программирования, разрешает создавать внутри одного класса несколько методов с одним именем. Главное, чтобы у них различались параметры. Параметры могут различаться типами или количеством аргументов. Будьте внимательны, если вы зададите различные типы для возвращаемого значения, то этого будет недостаточно для создания перегруженной версии метода. Когда Java встречает вызов перегруженного

метода, то выбирает ту версию, параметры которой соответствуют аргументам, использованным в вызове.

Создадим класс **Cat** с набором перегруженных методов:

```
class Cat {
    void meow() {
        // параметры отсутствуют
    }

    void meow(int count) {
        // используется один параметр типа int
    }

    void meow(int count, int pause) {
        // используются два параметра типа int
    }

    long meow(long time) {
        // используется один параметр типа long
        return time;
    }

    double meow(double time) {
        // используется один параметр типа double
        return time;
    }
}
```

Вы можете вызвать любой метод из класса:

```
Cat kitty = new Cat();
kitty.meow();
kitty.meow(3);
kitty.meow(3, 2);
kitty.meow(4500.25);
```

Если присмотреться, то можно догадаться, какая именно версия метода вызывается в каждом конкретном случае.

Аналогично, перегрузка используется и для конструкторов.

### Методы с переменным числом параметров

Можно создавать методы с переменным числом параметров. В этом случае используется многоточие.

```
public static double getMaxValue(Object... args) {
    // код
}
```

По сути, создаётся массив типа **Object[]**.

Создадим метод, вычисляющий и возвращающий максимальное из произвольного количества значений.

```
public static double getMaxValue(double... values) {
    double largest = Double.MIN_VALUE;
    for(double v : values) {
        if (v > largest) {
            largest = v;
        }
    }
}
```

```
        return largest;
    }
}
```

Использование метода.

```
double variable = getMaxValue(5.3, 39.6, -4);
```

До Java 5 использовался следующий способ.

```
static void printArray(Object[] args) {
    ...
}
```

Подобный код может встречаться в старых проектах, но в Android практически не используется.

## Метод toString()

Каждый класс реализует метод **toString()**, так как он определён в классе **Object**. Но использовать метод по умолчанию не слишком удобно, так как не содержит полезной информации. Разработчики предпочитают переопределять данный метод под свои нужды. Сам метод имеет форму:

```
String toString()
```

Вам остаётся вернуть объект класса **String**, который будет содержать полезную информацию о вашем классе. Давайте возьмём для примера класс **Box**, созданный выше:

```
class Box {
    int width; // ширина коробки
    int height; // высота коробки
    int depth; // глубина коробки

    Box(int width, int height, int depth){
        this.width = width;
        this.height = height;
        this.depth = depth;
    }

    // вычисляем объём коробки
    String getVolume() {
        return "Объём коробки: " + (width * height * depth);
    }

    public String toString() {
        return "Коробочка для кота размером " + width + "x" + height + "x"
+ depth;
    }
}
```

Теперь можете узнать о классе **Box**:

```
Box box = new Box(4, 5, 6);
System.out.println(box.toString());
```

Метод очень часто используется при создании собственных классов и вам тоже придётся прибегать к этому способу.

Как вернуть из метода больше одного значения?

Методы в Java возвращают одно значение. А если хочется сразу вернуть сразу два и более значений? Например, у нас имеется массив чисел и мы хотим написать метод, который сразу возвращает минимальное и максимальное значение из него.

Одно из решений заключается в том, что нужно "запаковать" два значения полученные в одном методе с помощью конструктора специального класса, а затем "распаковать" их в другом методе, при обращении к конструктору.

## **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_

## **7. Контрольные вопросы и задания**

9. Определение метода?
10. Метод с возвращаемым значением.
11. Метод без возвращаемого значения.
12. Параметры методов.
13. Как реализовать метод с изменяемым количеством параметров?
14. Механизм перегрузки методов.
15. Для чего необходима перегрузка методов?

## **8. Рекомендуемая литература**

37. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
38. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
39. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
40. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
41. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
42. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.



Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ, реализующих механизм наследования»

Минск  
2018

## Лабораторная работа № 8

### Тема работы: «Разработка программ, реализующих механизм наследования»

#### 1. Цель работы

Закрепить навык наследования

#### 2. Задание

Задачи первого и второго уровня обязательны для всех.

Реализовать классы по заданию (в задачах второго уровня должно присутствовать не менее 4 классов). Наследование применяем в логически нужных местах.

В классе должны быть реализованы:

- ✓ поля (закрытые или защищённые), нужное количество;
- ✓ свойства (для каждого поля);
- ✓ конструкторы (не менее двух);
- ✓ наследование конструкторов;
- ✓ методы (не менее трех);
- ✓ переопределенные родительские методы (нужные по логике);
- ✓ метод toString().

Номер варианта соответствует вашему номеру по списку.

*Примечание: в задании 1 начиная с 25 учащегося в списке необходимо определить свой вариант следующим образом:  $N \% 24$ , где  $N$  – ваш номер по списку,  $\%$  - остаток от деления.*

*В задании 2 начиная с 19 учащегося в списке необходимо определить свой вариант следующим образом:  $N \% 18$ , где  $N$  – ваш номер по списку,  $\%$  - остаток от деления.*

#### Задание 1

1. Создать объект класса Текст, используя классы Предложение, Слово. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса Автомобиль, используя классы Колесо, Двигатель. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса Самолет, используя классы Крыло, Шасси, Двигатель. Методы: летать, задавать маршрут, вывести на консоль маршрут.

4. Создать объект класса Государство, используя классы Область, Район, Город. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса Планета, используя классы Материк, Океан, Остров. Методы: вывести на консоль название материка, планеты, количество материков.
6. Создать объект класса Звездная система, используя классы Планета, Звезда, Луна. Методы: вывести на консоль количество планет в звездной системе, название звезды, добавление планеты в систему.
7. Создать объект класса Компьютер, используя классы Винчестер, Дисковод, Оперативная память, Процессор. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера.
8. Создать объект класса Квадрат, используя классы Точка, Отрезок. Методы: задание размеров, растяжение, сжатие, поворот, изменение цвета.
9. Создать объект класса Круг, используя классы Точка, Окружность. Методы: задание размеров, изменение радиуса, определение принадлежности точки данному кругу.
10. Создать объект класса Щенок, используя классы Животное, Собака. Методы: вывести на консоль имя, подать голос, прыгать, бегать, кусать.
11. Создать объект класса Наседка, используя классы Птица, Кукушка. Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса Текстовый файл, используя классы Файл, Директория. Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса Одномерный массив, используя классы Массив, Элемент. Методы: создать, вывести на консоль, выполнить операции (сложить, вычесть, перемножить).
14. Создать объект класса Простая дробь, используя класс Число. Методы: вывод на экран, сложение, вычитание, умножение, деление.
15. Создать объект класса Дом, используя классы Окно, Дверь. Методы: закрыть на ключ, вывести на консоль количество окон, дверей.
16. Создать объект класса Цветок, используя классы Лепесток, Бутона. Методы: расцвести, завянуть, вывести на консоль цвет бутона.
17. Создать объект класса Дерево, используя классы Лист, Ветка. Методы: зацвести, опадать листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса Пианино, используя классы Клавиша, Педаль. Методы: настроить, играть на пианино, нажимать клавишу.

19. Создать объект класса Фотоальбом, используя классы Фотография, Страница. Методы: задать название фотографии, дополнить фотоальбом фотографией, вывести на консоль количество фотографий.
20. Создать объект класса Год, используя классы Месяц, День. Методы: задать дату, вывести на консоль день недели по заданной дате, рассчитать количество дней, месяцев в заданном временном промежутке.
21. Создать объект класса Сутки, используя классы Час, Минута. Методы: вывести на консоль текущее время, рассчитать время суток (утро, день, вечер, ночь).
22. Создать объект класса Птица, используя классы Крылья, Клюв. Методы: летать, садиться, питаться, атаковать.
23. Создать объект класса Хищник, используя классы Когти, Зубы. Методы: рычать, бежать, спать, добывать пищу.
24. Создать объект класса Гитара, используя класс Струна, Скворечник. Методы: играть, настраивать, заменять струну.

## Задание 2

1. Цветочница. Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.
2. Новогодний подарок. Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.
3. Домашние электроприборы. Определить иерархию электроприборов. Включить некоторые в розетку. Подсчитать потребляемую мощность. Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.
4. Шеф-повар. Определить иерархию овощей. Сделать салат. Подсчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.

5. Звукозапись. Определить иерархию музыкальных композиций. Записать на диск сборку. Подсчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.
6. Камни. Определить иерархию драгоценных и полудрагоценных камней. Отобрать камни для ожерелья. Подсчитать общий вес (в каратах) и стоимость. Провести сортировку камней ожерелья на основе ценности. Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.
7. Мотоциклист. Определить иерархию амуниции. Экипировать мотоциклиста. Подсчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.
8. Транспорт. Определить иерархию подвижного состава железнодорожного транспорта. Создать пассажирский поезд. Подсчитать общую численность пассажиров и багажа. Провести сортировку вагонов поезда на основе уровня комфортности. Найти в поезде вагоны, соответствующие заданному диапазону параметров числа пассажиров.
9. Авиакомпания. Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.
10. Таксопарк. Определить иерархию легковых автомобилей. Создать таксопарк. Подсчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.
11. Страхование. Определить иерархию страховых обязательств. Собрать из обязательств дериватив. Подсчитать стоимость. Провести сортировку обязательств в деривативе на основе уменьшения степени риска. Найти обязательство в деривативе, соответствующее заданному диапазону параметров.
12. Мобильная связь. Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Подсчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.

13. Фургон кофе. Загрузить фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся, к тому же, в разных физических состояниях (зерно, молотый, растворимый в банках и пакетиках). Учитывать объем кофе вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону параметров качества.
14. Игровая комната. Подготовить игровую комнату для детей разных возрастных групп. Игрушек должно быть фиксированное количество в пределах выделенной суммы денег. Должны встречаться игрушки родственных групп: маленькие, средние и большие машины, куклы, мячи, кубики. Провести сортировку игрушек в комнате по одному из параметров. Найти игрушки в комнате, соответствующие заданному диапазону параметров.
15. Налоги. Определить множество и сумму налоговых выплат физического лица за год с учетом доходов с основного и дополнительного мест работы, авторских вознаграждений, продажи имущества, получения в подарок денежных сумм и имущества, переводов из-за границы, льгот на детей и материальной помощи. Провести сортировку налогов по сумме.
16. Счета. Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. Вычисление общей суммы по счетам. Вычисление суммы по всем счетам, имеющим положительный и отрицательный балансы отдельно.
17. Туристические путевки. Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т. д.) для оптимального выбора. Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.
18. Кредиты. Сформировать набор предложений клиенту по целевым кредитам различных банков для оптимального выбора. Учитывать возможность досрочного погашения кредита и/или увеличения кредитной линии. Реализовать выбор и поиск кредита.

### **3. Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

## 4. Основные теоретические сведения

### Наследование

Наследование является неотъемлемой частью Java. При использовании наследования вы говорите: Этот новый класс похож на тот старый класс. В коде это пишется как **extends**, после которого указываете имя базового класса. Тем самым вы получаете доступ ко всем полям и методам базового класса. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем вы можете наследоваться от него и создать новый класс, который будет иметь свои уникальные характеристики. Главный наследуемый класс в Java называют суперклассом. Наследующий класс называют подклассом. Получается, что подкласс - это специализированная версия суперкласса, которая наследует все члены суперкласса и добавляет свои собственные уникальные элементы. К примеру, в Android есть класс **View** и подкласс **TextView**.

Чтобы наследовать класс, достаточно вставить имя наследуемого класса с использованием ключевого слова **extends**:

```
public class MainActivity extends Activity {  
  
}
```

В этом коде мы наследуемся от класса **Activity** и добавляем свой код, который будет отвечать за наше приложение.

Подкласс в свою очередь может быть суперклассом другого подкласса. Так например, упоминавший ранее класс **TextView** является суперклассом для **EditText**.

В производный класс можно добавлять новые методы.

Для каждого создаваемого подкласса можно указывать только один суперкласс. При этом никакой класс не может быть собственным суперклассом.

Хотя подкласс включает в себя все члены своего суперкласса, он не может получить доступ к тем членам суперкласса, которые объявлены как **private**.

Помните, мы создавали класс **Box** для коробки кота. Давайте наследуемся от этого класса и создадим новый класс, который будет иметь не только размеры коробки, но и вес.

В том же файле **Box.java** после последней закрывающей скобки добавьте новый код:

```
class HeavyBox extends Box {  
    int weight; // вес коробки  
  
    // конструктор  
    HeavyBox(int w, int h, int d, int m) {
```

```

        width = w;
        height = h;
        depth = d;
        weight = m; // масса
    }
}

```

Возвращаемся в главную активность и пишем код:

```

HeavyBox box = new HeavyBox(15, 10, 20, 5);

int vol = box.getVolume();

mInfoTextView.setText("Объём коробки: " + vol + " Вес
коробки: " + box.weight);

```

Обратите внимание, что мы вызываем метод **getVolume()**, который не прописывали в классе **HeavyBox**. Однако мы можем его использовать, так как мы наследовались от класса **Box** и нам доступны все открытые поля и методы. Заодно мы вычисляем вес коробки с помощью новой переменной, которую добавили в подкласс.

Теперь у нас появилась возможность складывать в коробку различные вещи.

При желании вы можете создать множество разных классов на основе одного суперкласса. Например, мы можем создать цветную коробку.

```

class ColorBox extends Box {
    int color; // цвет коробки

    // конструктор
    ColorBox(int w, int h, int d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c; // цвет
    }
}

```

### Ключевое слово **super**

В Java существует ключевое слово *super*, которое обозначает **суперкласс**, т.е. класс, производным от которого является текущий класс. В данном случае, супер не означает превосходство, скорее даже наоборот, дочерний класс имеет больше методов, чем родительский. Само слово пошло из теории множеств, где используется термин супермножество. Посмотрим, зачем это нужно.

В конструкторе **HeavyBox** мы дублировали поля **width**, **height** и **depth**, которые уже есть в классе **Box**. Это не слишком эффективно. Кроме того, возможны ситуации, когда суперкласс имеет закрытые члены данных, но мы



хотим иметь к ним доступ. Через наследование это не получится, так как закрытые члены класса доступны только родному классу. В таких случаях вы можете сослаться на суперкласс.

Ключевое слово **super** можно использовать для вызова конструктора суперкласса и для обращения к члену суперкласса, скрытому членом подкласса.

### Использование ключевого слова **super** для вызова конструктора суперкласса

Перепишем пример:

```
class HeavyBox extends Box {
    int weight; // вес коробки

    // конструктор
    // инициализируем переменные с помощью ключевого
слова super
    HeavyBox(int w, int h, int d, int m) {
        super(w, h, d); // вызов конструктора
суперкласса
        weight = m; // масса
    }
}
```

Вызов метода **super()** всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

При вызове метода **super()** с нужными аргументами, мы фактически вызываем конструктор **Box**, который инициализирует переменные **width**, **height** и **depth**, используя переданные ему значения соответствующих параметров. Вам остаётся инициализировать только своё добавленное значение **weight**. При необходимости вы можете сделать теперь переменные класса **Box** закрытыми. Проставьте у полей класса **Box** модификатор **private** и убедитесь, что вы можете обращаться к ним без проблем.

У суперкласса могут быть несколько перегруженных версий конструкторов, поэтому можно вызывать метод **super()** с разными параметрами. Программа выполнит тот конструктор, который соответствует указанным аргументам.

Вторая форма ключевого слова **super** действует подобно ключевому слову **this**, только при этом мы всегда ссылаемся на суперкласс подкласса, в котором она использована. Общая форма имеет следующий вид:

```
super.член
```

Здесь *член* может быть методом либо переменной экземпляра.

Подобная форма подходит в тех случаях, когда имена членов подкласса скрывают члены суперкласса с такими же именами.

```

class A {
    int i;
}

// наследуемся от класса A
class B extends A {
    int i; // имя переменной совпадает и скрывает
переменную i в классе A

    B(int a, int b) {
        super.i = a; // обращаемся к переменной i из
класса A
        i = b; // обращаемся к переменной i из класса
B
    }

    void show() {
        System.out.println("i из суперкласса: " +
super.i);
        System.out.println("i в подклассе: " + i);
    }
}

class MainActivity {
    B subClass = new B(1, 2);
    subClass.show();
}

```

В результате мы должны увидеть:

i из суперкласса: 1

i в подклассе: 2

Таким образом, знакомое нам выражение **super.onCreate(savedInstanceState)** обращается к методу **onCreate()** из базового класса.

### Создание многоуровневой иерархии

Мы использовали простые примеры, состоящие из суперкласса и подкласса. Можно строить более сложные конструкции, содержащие любое количество уровней наследования. Например, класс С может быть подклассом класса В, который в свою очередь является подклассом класса А. В подобных ситуациях каждый подкласс наследует все характеристики всех его суперклассов.

Напишем пример из трёх классов. Суперкласс **Box**, подкласс **HeavyBox** и подкласс **MoneyBox**. Последний класс наследует все характеристики классов

Box и HeavyBox, а также добавляет поле **cost**, которое содержит стоимость коробки.

### Box.java

```
package ru.alexanderklimov.expresscourse;

class Box {
    private int width; // ширина коробки
    private int height; // высота коробки
    private int depth; // глубина коробки

    // Конструктор для создания клона объекта
    Box(Box ob) { // передача объекта конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // Конструктор, используемый при указании всех
измерений
    Box(int w, int h, int d) {
        width = w;
        height = h;
        depth = d;
    }

    // Конструктор, используемый, когда ни одно из
измерений не указано
    Box() {
        // значение -1 используется
        // для указания неинициализированного
параллелепипеда
        width = -1;
        height = -1;
        depth = -1;
    }

    // Конструктор для создания куба
    Box(int len) {
        width = height = depth = len;
    }

    // вычисляем объём коробки
```

```

        int getVolume() {
            return width * height * depth;
        }
    }

```

## **HeavyBox.java**

```

package ru.alexanderklimov.expresscourse;

//Добавление веса
class HeavyBox extends Box {
    int weight; // вес коробки

    // Конструктор клона объекта
    HeavyBox(HeavyBox ob) { // передача объекта
конструктору
        super(ob);
        weight = ob.weight;
    }

    // Конструктор, используемый
    // при указании всех параметров
    HeavyBox(int w, int h, int d, int m) {
        super(w, h, d); // вызов конструктора
суперкласса
        weight = m; // масса
    }

    // Конструктор по умолчанию
    HeavyBox() {
        super();
        weight = -1;
    }

    // Конструктор для создания куба
    HeavyBox(int len, int m) {
        super(len);
        weight = m;
    }
}

```

## **MoneyBox**

```

package ru.alexanderklimov.expresscourse;

//Цена коробки

```

```

class MoneyBox extends HeavyBox {
    int cost;

    // Конструирование клона объекта
    MoneyBox(MoneyBox ob) { // передача объекта
конструктору
        super(ob);
        cost = ob.cost;
    }

    // Конструктор, используемый
    // при указании всех параметров
    MoneyBox(int w, int h, int d, int m, int c) {
        super(w, h, d, m); // вызов конструктора
суперкласса
        cost = c;
    }

    // Конструктор по умолчанию
    MoneyBox() {
        super();
        cost = -1;
    }

    // Конструктор для создания куба
    MoneyBox(int len, int m, int c) {
        super(len, m);
        cost = c;
    }
}

```

Код для основной активности, например, при щелчке кнопки:

```

public void onClick(View v) {
    MoneyBox moneybox1 = new MoneyBox(100, 200, 150,
100, 300);
    MoneyBox moneybox2 = new MoneyBox(20, 30, 40, 7,
10);

    int vol;
    vol = moneybox1.getVolume();
    Log.i("Java-курс", "Объем первой коробки равен "
+ vol);
    Log.i("Java-курс", "Вес первой коробки равен " +
moneybox1.weight);
}

```

```

        Log.i("Java-курс", "Цена коробки: " +
moneybox1.cost);

        vol = moneybox2.getVolume();
        Log.i("Java-курс", "Объем второй коробки равен "
+ vol);
        Log.i("Java-курс", "Вес второй коробки равен " +
moneybox2.weight);
        Log.i("Java-курс", "Цена коробки: " +
moneybox2.cost);
    }

```

В результате мы получим различные значения, вычисляемые в коде. Благодаря наследованию, класс **MoneyBox** может использовать классы **Box** и **HeavyBox**, добавляя только ту информацию, которая нам требуется для его собственного специализированного применения. В этом и состоит принцип наследования, позволяя повторно использовать код.

Метод **super()** всегда ссылается на конструктор ближайшего суперкласса в иерархии. Т.е. метод **super()** в классе **MoneyBox** вызывает конструктор класса **HeavyBox**, а метод **super()** в классе **HeavyBox** вызывает конструктор класса **Box**.

Если в иерархии классов конструктор суперкласса требует передачи ему параметров, все подклассы должны передавать эти параметры по эстафете.

В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Если метод **super()** не применяется, программа использует конструктор каждого суперкласса, заданный по умолчанию или не содержащий параметров.

Вы можете создать три класса **A**, **B**, **C**, которые наследуются друг от друга (**A←B←C**), у которых в конструкторе выводится текст и вызвать в основном классе код:

```

C c = new C();

```

Вы должны увидеть три строчки текста, определённые в каждом конструкторе класса. Поскольку суперкласс ничего не знает о своих подклассах, любая инициализация полностью независима и, возможно, обязательна для выполнения любой инициализацией, выполняемой подклассом.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;

4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Для чего необходимо наследование?
2. С помощью какого ключевого слова организуется наследование?
3. Поддерживает ли Java множественное наследование?
4. Как организовать иерархию классов?

## **8. Рекомендуемая литература**

43. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
44. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
45. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
46. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
47. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
48. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ с использованием наследования и  
переопределение методов»

Минск  
2018



## Лабораторная работа № 9

**Тема работы: «Разработка программ с использованием наследования и переопределение методов»**

### 1. Цель работы

Отработать навык реализации наследования, и переопределения методов супер класса.

### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо реализовать в задания 1 лабораторной работы №7 логически продуманное наследование, при этом реализовать:

- ✓ цепочку не менее чем из 3 классов ;
- ✓ в классах потоках переопределить необходимые методы супер классов;
- ✓ организовать наследование конструкторов супер классов.

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

## 4. Основные теоретические сведения

### Переопределение методов

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, то метод подкласса *переопределяет* метод суперкласса. Когда переопределённый метод вызывается из своего подкласса, он всегда будет ссылаться на версию этого метода, определённую подклассом. А версия метода из суперкласса будет скрыта.

Если нужно получить доступ к версии переопределённого метода, определённого в суперклассе, то используйте ключевое слово **super**.

Не путайте переопределение с перегрузкой. Переопределение метода выполняется только в том случае, если имена и сигнатуры типов двух методов идентичны. В противном случае два метода являются просто перегруженными.

В Java SE5 появилась запись **@Override**; она не является ключевым словом. Если вы собираетесь переопределить метод, используйте **@Override**, и компилятор выдаст сообщение об ошибке, если вместо переопределения будет случайно выполнена перегрузка.

Для закрепления материала создадим класс **Animal** с одним методом.

```
package ru.alexanderklimov.expresscourse;

public class Animal {

    String sleep() {
        return "Животные иногда спят";
    }
}
```

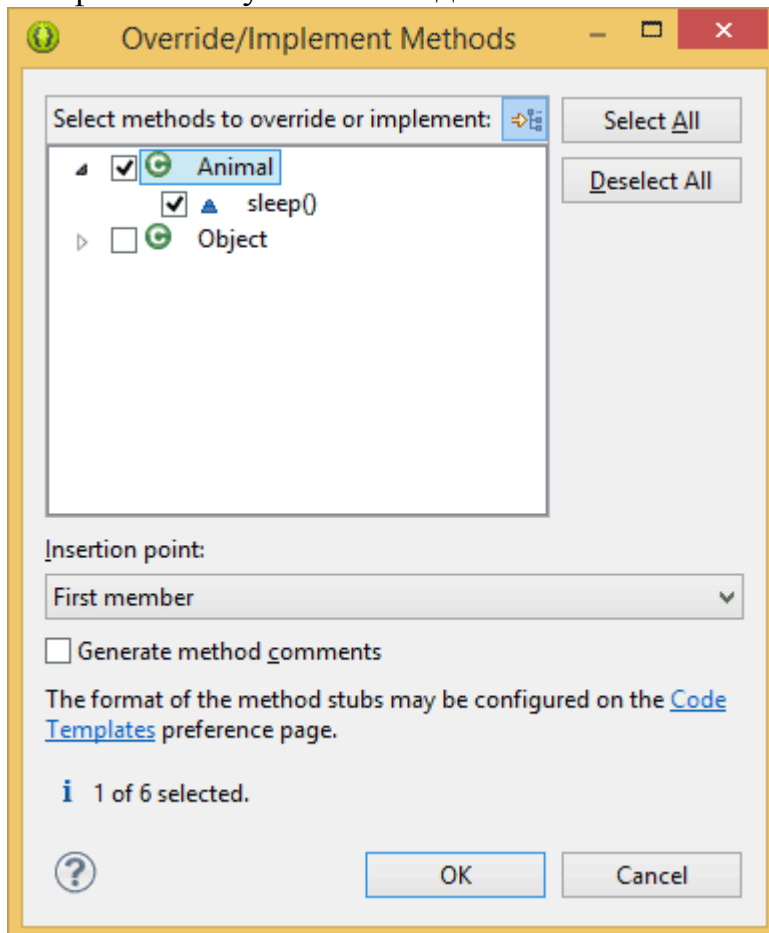
Теперь создадим класс **Cat**, наследующий от первого класса.

```
package ru.alexanderklimov.expresscourse;

public class Cat extends Animal {

}
```

Java знает, у родительского класса есть метод **sleep()**. Удостовериться можно следующим образом. Находясь в классе **Cat**, выберите в меню **Source | Override/Implement Methods...** Появится диалоговое окно, где можно отметить флажком нужный метод.



В результате в класс будет добавлена заготовка:

```
@Override
String sleep() {
    // TODO Auto-generated method stub
    return super.sleep();
}
```

Попробуем вызвать данный метод в основном классе активности:

```
public void onClick(View v) {
    Cat barsik = new Cat();
    mInfoTextView.setText(barsik.sleep());
}
```

Мы получим текст, который определён в суперклассе, хотя вызывали метод дочернего класса.

Но если мы хотим получить другой текст, совсем не обязательно придумывать новые методы. Достаточно закомментировать вызов метода из суперкласса и добавить свой вариант.

```
@Override
String sleep() {
    //return super.sleep();
    return "Коты постоянно спят!";
}
```

Запускаем программу и нажимаем на кнопку. И получим уже другой ответ, более соответствующий описанию среднестатистического кота. Заметьте, что код для щелчка кнопки мы не меняем, но система сама разберётся, что выводить нужно текст не из суперкласса, а из дочернего класса.

Рассмотрим другой пример переопределения методов. Создадим суперкласс **Figure**, который будет содержать размеры фигуры, а также метод для вычисления площади. А затем создадим два других класса **Rectangle** и **Triangle**, у которых мы переопределим данный метод.

```
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Площадь фигуры");
    }
}
```

```

        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // Переопределяем метод
    double area() {
        System.out.println("Площадь прямоугольника");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределяем метод
    double area() {
        System.out.println("Площадь треугольника");
        return dim1 * dim2 / 2;
    }
}

// В главной активности
Figure figure = new Figure(10, 10);
Rectangle rectangle = new Rectangle(8, 5);
Triangle triangle = new Triangle(10, 6);

Figure fig;

fig = figure;
mInfoTextView.setText("Площадь равна " + fig.area);

fig = rectangle;
mInfoTextView.setText("Площадь равна " + fig.area);

fig = triangle;
mInfoTextView.setText("Площадь равна " + fig.area);

```

Как видите, во всех классах используется одно и тоже имя метода, но каждый класс по своему вычисляет площадь в зависимости от фигуры. Это очень удобно и позволяет не придумывать новые названия методов в классах, которые наследуются от базового класса.

### **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

### **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы \_\_\_\_\_*

*Фамилия, инициалы учащегося \_\_\_\_\_*

*Дата выполнения работы \_\_\_\_\_*

*Тема работы: \_\_\_\_\_*

*Цель работы: \_\_\_\_\_*

*Оснащение работы: \_\_\_\_\_*

*Результат выполнения работы: \_\_\_\_\_*

\_\_\_\_\_

\_\_\_\_\_

### **7. Контрольные вопросы и задания**

5. Для чего необходимо наследование?
6. С помощью какого ключевого слова организуется наследование?
7. Поддерживает ли Java множественное наследование?
8. Как организовать иерархию классов?
9. Как переопределяется метод в языке Java?
10. Какое дает преимущество переопределение методов?

### **8. Рекомендуемая литература**

49. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
50. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
51. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
52. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
53. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
54. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ, реализующих и использующих интерфейсы»

Минск  
2018

## Лабораторная работа № 10

### Тема работы: «Разработка программ, реализующих и использующих интерфейсы»

#### 1. Цель работы

Закрепить реализации интерфейсов и абстрактных классов.

#### 2. Задание

Задачи первого и второго уровня обязательны для всех.

Создать и реализовать интерфейсы, также использовать наследование и полиморфизм для указанных предметных областей.

В классе должны быть реализованы:

- ✓ поля (закрытые или защищённые), нужное количество;
- ✓ свойства (для каждого поля);
- ✓ конструкторы (не менее двух);
- ✓ перегруженные операции;
- ✓ методы (нужное количество);
- ✓ метод toString();
- ✓ статические поля, если они нужны;
- ✓ статические свойства (если есть статические поля).

Номер варианта соответствует вашему номеру по списку.

***Примечание:** в задании начиная с 17 учащегося в списке необходимо определить свой вариант следующим образом:  $N \% 16$ , где  $N$  – ваш номер по списку,  $\%$  - остаток от деления.*

#### Задание 1

25. interface Издание <- abstract class Книга <- class Справочник и Энциклопедия
26. interface Абитуриент <- abstract class Студент <- class Студент-Заочник.
27. interface Сотрудник <- class Инженер <- class Руководитель.
28. interface Здание <- abstract class Общественное Здание <- class Театр.
29. interface Mobile <- abstract class Siemens Mobile <- class Model.
30. interface Корабль <- abstract class Военный Корабль <- class Авианосец.
31. interface Врач <- class Хирург <- class Нейрохирург.
32. interface Корабль <- class Грузовой Корабль <- class Танкер.
33. interface Мебель <- abstract class Шкаф <- class Книжный Шкаф.
34. interface Фильм <- class Отечественный Фильм <- class Комедия.
35. interface Ткань <- abstract class Одежда <- class Костюм.
36. interface Техника <- abstract class Плеер <- class Видеоплеер.



- 37. interface Транспортное Средство <- abstract class Общественный Транспорт <- class Трамвай.
- 38. interface Устройство Печати <- class Принтер <- class Лазерный Принтер.
- 39. interface Бумага <- abstract class Тетрадь <- class Тетрадь Для Рисования.
- 40. interface Источник Света <- class Лампа <- class Настольная Лампа.

## Задание 2

Необходимо в задания 1 лабораторной работы №9: для всех цепочек классов связанных наследованием, добавить абстрактные классы и интерфейсы, необходимые по логике.

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### 4. Основные теоретические сведения

#### Абстрактные классы и методы

Класс, содержащий абстрактные методы, называется абстрактным классом. Такие классы помечаются ключевым словом **abstract**.

Абстрактный метод не завершён. Он состоит только из объявления и не имеет тела:

```
abstract void yourMethod();
```

По сути, мы создаём шаблон метода. Например, можно создать абстрактный метод для вычисления площади фигуры в абстрактном классе Фигура. А все другие производные классы от главного класса могут уже реализовать свой код для готового метода. Ведь площадь у прямоугольника и треугольника вычисляется по разным алгоритмам и универсального метода не существует.

Если вы объявляете класс, производный от абстрактного класса, но хотите иметь возможность создания объектов нового типа, вам придётся предоставить определения для всех абстрактных методов базового класса. Если этого не сделать, производный класс тоже останется абстрактным, и компилятор заставит пометить новый класс ключевым словом **abstract**.

Можно создавать класс с ключевым словом **abstract** даже, если в нем не имеется ни одного абстрактного метода. Это бывает полезным в ситуациях, где

в классе абстрактные методы просто не нужны, но необходимо запретить создание экземпляров этого класса.

В тоже время абстрактный класс не обязательно должен иметь только абстрактные методы. Напомню ещё раз, что если класс содержит хотя бы один абстрактный метод, то он обязан быть сам абстрактным.

Создавать объект на основе абстрактного класса нельзя.

```
// нельзя. даже не пытайтесь
AbstractClass abstractClass = new AbstractClass();
```

Абстрактный класс не может содержать какие-либо объекты, а также абстрактные конструкторы и абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо сам быть объявлен абстрактным. Короче, я сам запутался. Пойду лучше кота поглажу.

Я вернулся. Давайте напишем пример для абстрактного класса.

Допустим, мы хотим создать абстрактный класс СферическийКонь и класс СферическийКоньВВакууме, наследующий от первого класса.

```
package ru.alexanderklimov.expresscourse;

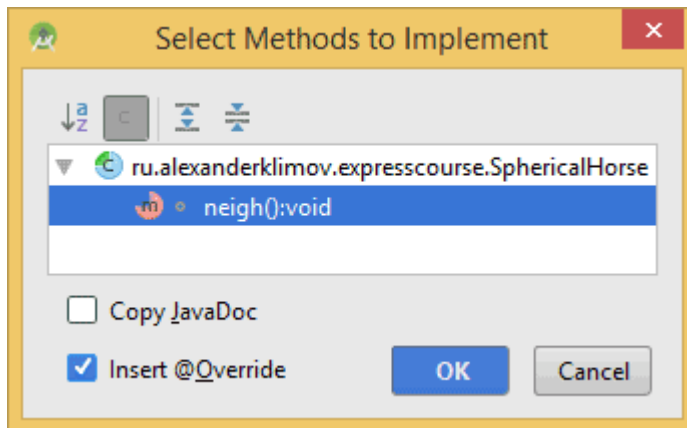
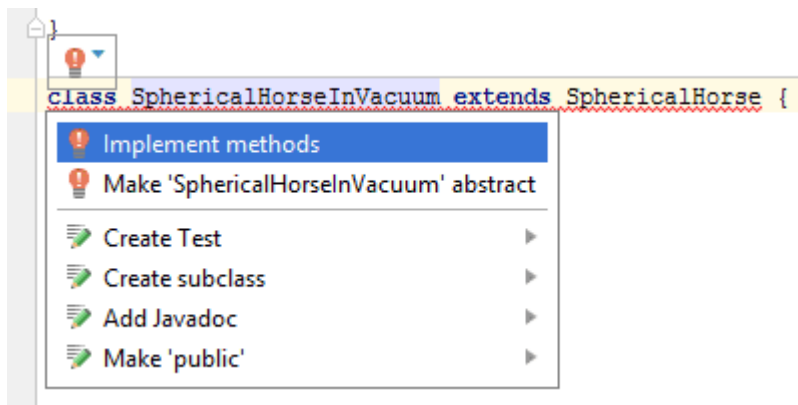
public abstract class SphericalHorse {
    // абстрактный метод ржать()
    abstract void neigh();

    // абстрактный класс может содержать и обычный
метод
    void gallop() {
        System.out.println("Куда прёшь?");
    }
}

class SphericalHorseInVacuum extends SphericalHorse {

}
```

Когда вы напишете такой код, то студия подчеркнёт второй класс красной волнистой линией и предложит реализовать обязательный метод, который определён в абстрактном классе.



Соглашаемся и дописываем в созданную заготовку свой код для метода.

```
@Override
void neigh() {
    System.out.println("Чё ты ржёшь?");
}
```

В главной активности напишем код для щелчка кнопки.

```
public void onClick(View view) {
    SphericalHorseInVacuum horse = new
SphericalHorseInVacuum();
    horse.neigh(); // на основе абстрактного метода
    horse.gallop(); // обычный метод
}
```

Обратите внимание, что абстрактный класс может содержать не только абстрактные, но и обычные методы.

Ранее мы создавали класс Фигура, у которого был метод вычисления площади фигуры. Метод ничего не делал, так как невозможно вычислить площадь неизвестной фигуры. Поэтому, этот метод можно сделать абстрактным, а в классах, производных от Фигуры, переопределить данный метод.

```
// абстрактный класс Фигура
```

```

abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // абстрактный метод для вычисления площади
    abstract double area();
}

// Клас Треугольник
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределяем метод
    double area() {
        return dim1 * dim2 / 2;
    }
}

// В активности
public void onClick(View view) {
    // Figure figure = new Figure(10, 10); // так
нельзя
    Triangle triangle = new Triangle(10, 8);

    Figure figure; // так можно, так как объект мы не
создаем
    figure = triangle;
    mInfoTextView.append("Площадь      равна      "      +
figure.area());
}

```

Фигура - это абстрактное понятие и мы не можем создать универсальный метод для вычисления площади. Поэтому мы создаём другой класс Треугольник и пишем код, вычисляющий площадь треугольника (загляните в учебник геометрии). Также вы можете создать новый класс Прямоугольник и написать свой код для вычисления площади этой фигуры.

Вам вряд ли придётся часто создавать абстрактные классы для своих приложений, но встречаться с ними вы будете постоянно, например, классы **AsyncTask**, **Service** и др.

Кот - понятие тоже абстрактное. А вот ваш любимец Васька или Мурзик уже конкретный шерстяной засранец. По запросу "abstract cat" мне выдало такую картинку. Всегда интересовал вопрос, где художники берут такую травку? (это был абстрактный вопрос).

## Интерфейсы

Ключевое слово **interface** используется для создания полностью абстрактных классов. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

Наличие слова **interface** означает, что именно так должны выглядеть все классы, которые реализуют данный интерфейс. Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того.

Чтобы создать интерфейс, используйте ключевое слово **interface** вместо **class**. Как и в случае с классами, вы можете добавить перед словом **interface** спецификатор доступа **public** (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс может содержать поля, но они автоматически являются статическими (**static**) и неизменными (**final**). Все методы и переменные неявно объявляются как **public**.

Класс, который собирается использовать определённый интерфейс, использует ключевое слово **implements**. Оно указывает, что интерфейс лишь определяет форму, а вам нужно наполнить кодом. Методы, которые реализуют интерфейс, должны быть объявлены как **public**.

Интерфейсов у класса может быть несколько, тогда они перечисляются за ключевым словом **implements** и разделяются запятыми.

Интерфейсы могут вкладываться в классы и в другие интерфейсы.

Если класс содержит интерфейс, но не полностью реализует определённые им методы, он должен быть объявлен как **abstract**.

Интерфейсы — это не классы. С помощью ключевого слова **new** нельзя создать экземпляр интерфейса:

```
x = new List(...); // Нельзя!
```

Но можно объявлять интерфейсные переменные:

```
List<String> catNames; // Можно!
```

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс.

```
List<String> catNames = new ArrayList<>();
```

Рассмотрим быстрый пример создания интерфейса. Выберите в меню **File | New | Interface** и придумайте имя для нового интерфейса. В полученной заготовке добавьте два имени метода (только имена, без кода).

```
package ru.alexanderklimov.quickcourse;

public interface SimpleInterface {
    String getClassName();
    int getAge();
}
```

Создайте или откройте какой-нибудь класс, к которому нужно применить интерфейс, и добавьте к нему **implements SimpleInterface**. Среда разработки подчеркнёт красной линией имя класса и предложит добавить методы, которые требуются интерфейсом. Соглашаемся и получаем результат:

```
package ru.alexanderklimov.quickcourse;

public class Cat extends Animal implements
SimpleInterface {

    @Override
    public String getClassName() {
        return null;
    }

    @Override
    public int getAge() {
        return 0;
    }
}
```

Среда разработки сгенерировала два метода и использовала в качестве возвращаемых результатов значения по умолчанию. Это могут быть и нулевые значения и **null**. Осталось подправить шаблоны созданных методов под свои задачи. Например, так:

```
@Override
public String getClassName() {
    return "Cat";
}

@Override
public int getAge() {
    return 5;
}
```

Первый метод возвращает имя класса, а второй - возраст кота (странно, что всем котам будет по пять лет, но это лишь пример).

Здесь важно понять роль интерфейса. Мы лишь придумываем имена, а класс уже реализует нужную задачу. Для примера можно создать в интерфейсе метод **play()** для класса Пианино и класса Гитара, так как играть можно на обоих инструментах. Но код в методах будет отличаться, так как принцип игры на инструментах совершенно разный.

### Константы в интерфейсах

Интерфейсы можно использовать для импорта констант в несколько классов. Вы просто объявляете интерфейс, содержащий переменные с нужными значениями. При реализации интерфейса в классе имена переменных будут помещены в область констант. Поля для констант становятся открытыми и являются статическими и конечными (модификаторы **public static final**). При этом, если интерфейс не будет содержать никаких методов, то класс не будет ничего реализовывать. Хотя данный подход не рекомендуют использовать.

### Расширение интерфейсов

Интерфейс может наследоваться от другого интерфейса через ключевое слово **extends**.

### Методы обратного вызова

Интерфейсы часто используются для создания методов обратного вызова (callback). Рассмотрим такой пример. Создадим новый класс **SubClass** с интерфейсом **MyCallback**:

```
package ru.alexanderklimov.quickcourse;

public class SubClass {
    interface MyCallback{
        void callBackReturn();
    }

    MyCallback myCallback;

    void registerCallBack(MyCallback callback){
        this.myCallback = callback;
    }

    void doSomething(){
        // Здесь какой-то длительный код
        // Например, тянем кота за хвост

        // вызываем метод обратного вызова
```

```

        myCallback.callBackReturn();
    }
}

```

У интерфейса мы определили один метод **callBackReturn()**. Далее в классе мы создали объект интерфейса и инициализировали его в конструкторе класса. В классе также был создан метод **doSomething()**, в котором может содержаться какой-то сложный код. В конце метода вызывается метод интерфейса. В данном случае мы сами создали метод и знаем его код. Но во многих случаях, вы будете использовать готовый метод какого-то класса и вы не будете знать, что именно содержится в этом методе. Вам надо только знать, что такой метод существует, например, из документации и он выполняет конкретную задачу.

Переходим в код активности и подключаем интерфейс через ключевое слово **implements**:

```

public class MainActivity extends ActionBarActivity
implements SubClass.MyCallback {}

```

Среда разработки поможет вставить шаблон метода интерфейса.

```

@Override
public void callBackReturn() {

}

```

Теперь мы можем использовать метод обратного вызова **callBackReturn()** для решения своих задач. Допустим у нас есть текстовая метка и кнопка. При щелчке выполняется какой-то сложный код из класса **SubClass**. Когда он закончит работу, то сработает метод обратного вызова **callBackReturn()**, в котором пропишем нужные действия.

```

package ru.alexanderklimov.testapplication;

import ...

public class MainActivity extends ActionBarActivity
implements SubClass.MyCallback {

    private TextView mResultTextView;
    private SubClass mSubClass;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
    }
}

```



```

        mResultTextView
(TextView) findViewById(R.id.textviewResult);

        mSubClass = new SubClass();
        mSubClass.registerCallBack(this);
    }

    public void onClick(View v) {
        mSubClass.doSomething();
    }

    @Override
    public void callBackReturn() {
        mResultTextView.setText("Вызван
обратного вызова");
    }
}

```

Очень часто для интерфейса используют слово **Listener**, когда вы будете встречать это слово, то обратите внимание на логику работы кода. Вы узнаете знакомый код из этого примера.

Также интерфейсы часто используются при работе с фрагментами.

Требуется определённая практика и опыт, чтобы быстро разбираться в коде с использованием интерфейсов, так как приходится отслеживать цепочку вызовов из разных классов. Но бояться их не нужно.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

---

---

### **7. Контрольные вопросы и задания**

1. Какой класс называется абстрактным?
2. Как создать абстрактный класс?
3. Что такое интерфейс?
4. В чем отличие между абстрактным классом и интерфейсом?
5. Назначение и применение абстрактных методов.

### **8. Рекомендуемая литература**

55. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
56. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
57. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
58. Лафоре Р. Структуры данных и алгоритмы на Java – СПб: Питер, 2013. – 704 с.
59. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
60. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ с использованием пакетов»

Минск  
2018

## Лабораторная работа № 11

### Тема работы: «Разработка программ с использованием пакетов»

#### 1. Цель работы

Закрепить навык создания пакетов.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 2 лабораторной работы № 10 создать несколько пакетов, и классы по логике разнести па пакетом. Продумать видимость классов и их компонентов, внутри пакетов и за их пределами.

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Пакеты

Пакет является специальным библиотечным модулем, который содержит группу классов, объединённых в одном пространстве имён. Например, существует системная библиотека **android.widget**, в состав которой входит класс **Button**. Чтобы использовать класс в программе, можно привести его полное имя **android.widget.Button**. Но длинные имена классов не слишком удобно использовать в коде, поэтому можно использовать ключевое слово **import**.

```
import android.widget.Button;
```

Теперь к классу **Button** можно обращаться без указания полного имени.

Использование механизма импортирования обеспечивает возможность управления пространствами имён. Предположим, что вы создали класс **Cat** со своим набором методов, а кто-то другой тоже создал класс с подобным именем и со своим набором методов. Если вы захотите использовать свой и чужой класс в своей программе, то возникнет конфликт имён, так как Java не сможет понять, какой класс нужно использовать для вызова метода.

Файл с исходным кодом на Java является компилируемым модулем. Имя модуля имеет расширение *java*, а внутри него может находиться открытый (**public**) класс с именем файла без расширения. Модуль может содержать один открытый класс, остальные классы не должны быть открытыми и считаются вспомогательными по отношению к главному открытому классу.

Как уже говорилось, библиотека является набором файлов с классами. Директива **package** должна находиться в первой незакомментированной строке

файла. По правилам Java имена пакетов записываются только строчными буквами. Все классы, находящиеся внутри данного файла, будут принадлежать указанному пакету. Если оператор **package** не указан, то имена классов помещаются в специальный пакет без имени. Но вы должны всячески избегать подобных ситуаций.

Указывать один и тот же пакет можно в разных файлах, он просто указывает кому принадлежит класс. Поэтому, если три разных класса в трёх разных файлах указывают на один и тот же **package**, то это нормально.

Можно создавать иерархию пакетов через точечный оператор:

```
package pack1[.pack2[.pack3]];
// например
package cat.body.tail;
```

### Создание уникальных имён пакетов

Существует общепринятая схема, где первая часть имени пакета должна состоять из перевёрнутого доменного имени разработчика класса. Так как доменные имена в интернете уникальны, соблюдение этого правила обеспечивает уникальность имён пакетов и предотвратит конфликты. Если у вас нет собственного доменного имени, то придумайте свою уникальную комбинацию с малой вероятностью повторения.

### Доступ к членам класса

Модификаторы обеспечивают различные уровни доступа к членам класса. Пакеты также вносят свою лепту. Можно представить себе такую таблицу.

	<b>private</b>	<b>Модификатор не указан</b>	<b>protected</b>	<b>public</b>
В том же классе	Да	Да	Да	Да
Подкласс класса этого же пакета	Нет	Да	Да	Да
Класс из общего пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс класса другого пакета	Нет	Нет	Да	Да

---

Класс другого пакета, не являющийся подклассом класса данного пакета	Нет	Нет	Нет	Да
----------------------------------------------------------------------	-----	-----	-----	----

Любой компонент, объявленный как **public**, будет доступен из любого места. Компонент, объявленный как **private**, не виден для компонентов, расположенных вне его класса. Если модификатор явно не указан, он видим подклассам и другим классам в данном пакете. Это стандартное поведение по умолчанию. Если нужно, чтобы компонент был видим за пределами текущего пакета, но только классам, которые являются непосредственными подклассами данного класса, то используйте **protected**.

Это справедливо только для членов класса. У класса можно указать только два варианта: по умолчанию (не указан) и **public**.

Скорее всего в Android вам не придётся иметь дело с пакетами вплотную.

### Импорт

В начале статьи я говорил вам, что импорт позволяет сократить написание полного имени класса. Он создан для удобства программистов и программа может обойтись и без него. Но если не выпендриваться и использовать импорт, то вы сократите уменьшите объём вводимого кода.

Оператор **import** должен идти сразу после оператора **package** (если он есть). Кстати, имя класса можно указать явно или с помощью символа "звёздочка" (\*):

```
import java.io.*;
```

Но в Android такой способ категорически не рекомендуется использовать, так как ведёт к большому потреблению ресурсов. Да и сам я не разу не видел такой способ в примерах.

Таким образом, стандартный вариант:

```
public class MainActivity extends Activity {}
```

Можно заменить на вариант, удалив строчку кода из импорта:

```
public class MainActivity extends
android.app.Activity {}
```

Дополнительные сведения об импорте.

### Создание пакета

В студии создать пакет можно двумя способами. Первый - традиционный, щёлкаем правой кнопкой мыши на папке **java** или на существующем пакете и выбираем в меню команду **New | Package**.

Второй способ более хитрый - когда вы создаёте в студии новый класс, то указывая его имя можете использовать точечную нотацию,

например, **database.CatDB**. В этом случае пакет **database** будет создан автоматически и в нём будет находиться создаваемый класс.

### 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

### 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

### 7. Контрольные вопросы и задания

1. Что такое пакет?
2. Для чего нужны пакеты?
3. Видимость элементов классов в разных пакетах.

### 8. Рекомендуемая литература

61. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
62. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
63. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
64. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.

- 65.Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. –240с.
- 66.Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.



Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ обработки символов»

Минск  
2018

## Лабораторная работа № 12

### Тема работы: «Разработка программ обработки символов»

#### 1. Цель работы

Закрепление навыка работы с символьными переменными.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 11 создать методы для шифровки и дешифровки важной и секретной информации. Нужно реализовать один из методов шифрования (по варианту).

Номер варианта определяется последующей формуле:  $(N \% 11) + 1$ , где N – ваш номер по списку.

1. Аффинная система шифрования Цезаря
2. Биграммный шифр Плейфера
3. Двойная таблица перестановки
4. Криптосистема Хилла
5. Магический квадрат
6. Простая таблица перестановки
7. Система цезаря с ключевым словом
8. Система шифрования Вижинера
9. Система шифрования Цезаря
10. Шифр «двойной квадрат» Уинстона
11. Шифрующие таблицы Трисемуса

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

Описание типа Char для работы с символами. Для работы с символами в Java используется тип данных char. Объект этого типа является 16-битным символом Unicode. Его значение по умолчанию равно `'\u0000'`.

Присваивать значение переменной типа Char можно несколькими способами.

Пусть имеется переменная a1 типа char:

## Операция

```
char a1 = 'a';
```

Запишет в переменную a1 символ a.

Также мы можем получить символ по его номеру в Unicode-таблице. Пусть имеется число int i содержащее номер символа в таблице.

## Операция

```
char a2 = (char) i;
```

Запишет в переменную a2 символ находящийся в Unicode-таблице под номером i.

Также можно сравнивать переменные типа char. Пусть имеется переменная b1 типа Boolean и 2 объекта типа char – a1 и a2.

## Операция

```
b1 = (a1 < a2);
```

Запишет в переменную b1 значение true в случае если символ a1 встречается в Unicode-таблице раньше символа a2.

Как уже говорилось, char содержит лишь один символ, а вот о том как в java осуществляется работа со строками будет рассказано в следующем уроке.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

---

---

### **7. Контрольные вопросы и задания**

1. Как создать символьную переменную?
2. Назначение символьной переменной?
3. Применение символьной переменной?

### **8. Рекомендуемая литература**

67. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
68. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
69. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
70. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
71. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
72. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ разработки строк»

Минск  
2018

## Лабораторная работа № 13

### Тема работы: «Разработка программ разработки строк»

#### 1. Цель работы

Отработать навык работы с строковыми переменными реализованными с помощью различных классов.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

*Примечание: в задании начиная с 14 учащегося в списке необходимо определить свой вариант следующим образом:  $(N \% 13) + 1$ , где  $N$  – ваш номер по списку,  $\%$  - остаток от деления.*

1. Ввести  $n$  строк с консоли, найти самую короткую строку. Вывести эту строку и ее длину.
2. Ввести  $n$  строк с консоли. Упорядочить и вывести строки в порядке возрастания их длин, а также (второй приоритет) значений этих их длин.
3. Ввести  $n$  строк с консоли. Вывести на консоль те строки, длина которых меньше средней, также их длины.
4. В каждом слове текста  $k$ -ю букву заменить заданным символом. Если  $k$  больше длины слова, корректировку не выполнять.
5. В русском тексте каждую букву заменить ее номером в алфавите. В одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.
6. Из небольшого текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
7. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
8. В тексте найти все пары слов, из которых одно является обращением другого.
9. Найти и напечатать, сколько раз повторяется в тексте каждое слово.
10. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.
11. Выбрать три разные точки заданного на плоскости множества точек, составляющие треугольник наибольшего периметра.
12. Найти такую точку заданного на плоскости множества точек, сумма расстояний от которой до остальных минимальна.
13. Выпуклый многоугольник задан на плоскости перечислением координат вершин в порядке обхода его границы. Определить площадь многоугольника.

### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

### 4. Основные теоретические сведения

#### Класс **String**

Класс **String** очень часто используется программистами, поэтому его следует изучить очень хорошо.

Следует помнить, что объекты класса **String** являются неизменяемыми (immutable). Поэтому, когда вам кажется, что вы меняете строку, то на самом деле вы создаёте новую строку.

В Java есть специальные классы **StringBuffer** и **StringBuilder**, который допускают изменения в строке.

Классы **String**, **StringBuffer**, **StringBuilder** определены в пакете **java.lang** и доступны автоматически без объявления импорта. Все три класса реализуют интерфейс **CharSequence**.

Создать строку очень просто. Например, можно так:

```
String aboutCat = "Кот - это звучит гордо, а если наступить на хвост, то громко";
```

Можно создать массив строк:

```
String[] cats = {"Васька", "Барсик", "Мурзик"};
```

Можно создать пустой объект класса **String**:

```
String str = new String();
```

Можно создать строку через массив символов:

```
char[] chars = { 'c', 'a', 't' };
```

```
String str = new String(chars);
```

Есть ещё конструктор, позволяющий задать диапазон символьного массива. Вам нужно указать начало диапазона и количество символов для использования:

```
char[] chars = { 'c', 'a', 't', 'a', 'm', 'a', 'r', 'a', 'n' };
String str = new String(chars, 0, 3);
```

Можно создать объект класса **String** из объектов классов **StringBuffer** и **StringBuilder** при помощи следующих конструкторов:

```
String(StringBuffer объект_StrBuf)
String(StringBuilder объект_StrBuild)
```

## Операторы + и += для String

На языке Java знак плюс (+) означает конкатенацию строк (concatenation), иными словами - объединение строк.

```
String cat = "Кот";
String name = "Васька";
//складываем две строки и пробел между ними, чтобы
слова не слиплись
String fullname = cat + " " + name; // получится Кот
Васька
```

Если один из операндов в выражении содержит строку, то другие операнды также должны быть строками. Поэтому Java сама может привести переменные к строковому представлению, даже если они не являются строками.

```
int digit = 4;
String paws = " лапы";
String aboutcat = digit + paws; // хотя мы складываем
число и строку, но все равно получим строку
За кулисами Java за нас преобразовало число 4 в строку "4".
```

## Строковой ресурс

Строки желательно хранить в ресурсах (о ресурсах есть отдельная статья). Программно доступ к строковому ресурсу делается так:

```
String catName =
getResources().getString(R.string.barsik);
```

## Извлечь строки из строковых массивов в ресурсах

Предположим, у вас есть строковый массив, определенный в файле **strings.xml** под именем **cats\_array**. Тогда получить доступ к строкам из ресурсов можно так:

```
Resources res = getResources();
String[] cats =
res.getStringArray(R.array.cats_array);
```

## Методы

### **public char charAt (int index)**

Возвращает символ с указанным смещением в этой строке. Отсчет идет от 0. Не надо использовать отрицательные и несуществующие значения, будьте серьезнее. Для извлечения нескольких символов используйте `getChars()`.



```
String testString = "Котёнок";
char myChar = testString.charAt(2);
tv.setText(Character.toString(myChar));    //    выводит
третий символ - т
```

### **public int codePointAt(int index)**

Возвращает Unicode-символ в заданном индексе

```
String testString = "Котёнок";
int myChar = testString.codePointAt(3);
tv.setText(String.valueOf(myChar));    //    возвращает
1105
```

### **public int codePointBefore(int index)**

Возвращает Unicode-символ, который предшествует данному индексу

```
String testString = "Котёнок";
int myChar = testString.codePointBefore(4);
tv.setText(String.valueOf(myChar));    //    возвращает
1105
```

### **public int codePointCount(int start, int end)**

Вычисляет количество Unicode-символов между позициями **start** и **end**

```
String testString = "Котёнок";
int myChar = testString.codePointCount(0, 3);
tv.setText(String.valueOf(myChar)); // возвращает 3
```

### **public int compareTo(String string)**

Сравнивает указанную строку, используя значения символов Unicode и вычисляет, какая из строк меньше, равна или больше следующей. Может использоваться при сортировке. Регистр учитывается. Если строки совпадают, то возвращается 0, если меньше нуля, то вызывающая строка меньше строки **string**, если больше нуля, то вызывающая строка больше строки **string**. Слова с большим регистром стоят выше слова с нижним регистром.

```
String testString = "Котёнок";

if (testString.compareTo("котёнок") == 0) {
    tvInfo.setText("Строки равны");
} else {
    tvInfo.setText("Строки не равны. Возвращено"
        + testString.compareTo("котёнок"));    //
возвращает -32
}
```

Отсортируем массив строк через пузырьковую сортировку.

```
String[] poem = { "Мы", "везём", "с", "собой", "кота"
};

for (int j = 0; j < poem.length; j++) {
    for (int i = j + 1; i < poem.length; i++) {
        if (poem[i].compareTo(poem[j]) < 0) {
            String temp = poem[j];
            poem[j] = poem[i];
            poem[i] = temp;
        }
    }
    System.out.println(poem[j]);
}
```

В результате мы получим:

```
Мы
везём
кота
с
собой
```

### **public int compareToIgnoreCase (String string)**

Сравнивает указанную строку, используя значения символов Unicode, без учета регистра.

```
String testString = "Котёнок";

if (testString.compareToIgnoreCase("котёнок") == 0) {
    tv.setText("Строки равны"); // слова одинаковы,
если не учитывать регистр
} else {
    tv.setText("Строки не равны. Возвращено"
        + testString.compareTo("котёнок"));
}
```

### **public String concat (String string)**

Объединяет строку с указанной строкой. Возвращается новая строка, которая содержит объединение двух строк. Обратите внимание, что само имя метода содержит кота!

```
String testString = "Сук";
String newString = testString.concat("кот");
tv.setText(newString);
```

Метод выполняет ту же функцию, что и оператор + и можно было написать **Сук + кот**. Но настоящий кошатник будет использовать "кошачий" метод.

### **public boolean contains(CharSequence cs)**

Определяет, содержит ли строка последовательность символов в `CharSequence`

```
String testString = "котёнок";

if(testString.contains("кот")){
    infoTextView.setText("В слове котёнок содержится слово кот!");
}
```

### **public boolean contentEquals(CharSequence cs)**

Сравнивает `CharSequence` с этой строкой.

### **public boolean endsWith(String suffix)**

Проверяет, заканчивается ли строка символами *suffix*.

```
String str1 = "Суккот";

if(str1.endsWith("кот"))
    infoTextView.setText("Слово заканчивается на котике");
else
    infoTextView.setText("Плохое слово. Нет смысла его использовать");
```

### **public boolean equals(Object string)**

Сравнивает указанный объект и строку и возвращает *true*, если сравниваемые строки равны, т.е. содержит те же символы и в том же порядке с учётом регистра.

```
String str1 = "Кот";
String str2 = "Кошка";

if(str1.equals(str2))
    infoTextView.setText("Строки совпадают");
else
    infoTextView.setText("Строки не совпадают");
```

Не путать метод с оператором `==`, который сравнивает две ссылки на объекты и определяет, ссылаются ли они на один и тот же экземпляр. Смотри пункт Сравнение строк: `equals()` или `==`?

### **public boolean equalsIgnoreCase(String string)**

Сравнивает указанную строку с исходной строкой без учёта регистра и возвращает *true*, если они равны. Диапазон A-Z считается равным диапазону a-z.

```
String str1 = "Кот";
```

```
String str2 = "кот";
```

```
if(str1.equalsIgnoreCase(str2))  
    infoTextView.setText("Строки совпадают");
```

```
else
```

```
    infoTextView.setText("Строки не совпадают");
```

**public static String format(Locale locale, String format, Object... args)**

Возвращает отформатированную строку, используя прилагаемый формат и аргументы, локализованных в данной области. Например дату или время

```
// выводим число типа float с двумя знаками после запятой
```

```
String.format("%.2f", floatValue);
```

Склеиваем два слова, которые выводятся с новой строки. При этом второе слово выводится в верхнем регистре.

```
String str1 = "Кот";
```

```
String str2 = "васька";
```

```
String strResult = String.format("%s\n%S", str1,  
str2);
```

```
// выводим результат в TextView
```

```
infoTextView.setText(strResult);
```

Конвертируем число в восьмеричную систему.

```
String str1 = "8";
```

```
int inInt = Integer.parseInt(str1); // конвертируем  
строку в число
```

```
String strResult = String.format("(Восьмеричное  
значение): %o\n", inInt);
```

```
infoTextView.setText(strResult);
```

По аналогии выводим в шестнадцатеричной системе

```
String str1 = "255";
```

```
int inInt = Integer.parseInt(str1);
```

```
String strResult = String.format("(Шестнадцатеричное  
значение): %x\n", inInt);
```

```
// число 255 будет выведено как ff
```

```
infoTextView.setText(strResult);
```

Для верхнего регистра используйте **%X**, тогда вместо **ff** будет **FF**.

Для десятичной системы используйте **%d**.

Дату тоже можно выводить по разному.

```
Date now = new Date();
```

```

Locale locale = Locale.getDefault();
infoTextView.setText(
    String.format(locale, "%tD\n", now) + //
(MM/DD/YY)
    String.format(locale, "%tF\n", now) + //
(YYYY-MM-DD)
    String.format(locale, "%tr\n", now) + //
Full 12-hour time
    String.format(locale, "%tz\n", now) + //
Time zone GMT offset
    String.format(locale, "%tZ\n", now)); //
Localized time zone abbreviation

```

### **public void getChars(int start, int end, char[] buffer, int index)**

Метод для извлечения нескольких символов из строки. Вам надо указать индекс начала подстроки (start), индекс символа, следующего за концом извлекаемой подстроки (end). Массив, который принимает выделенные символы находится в параметре **buffer**. Индекс в массиве, начиная с которого будет записываться подстрока, передаётся в параметре **index**. Следите, чтобы массив был достаточно размера, чтобы в нём поместились все символы указанной подстроки.

```

String unusualCat = "Котёнок по имени Гав";
int start = 5;
int end = 12;
char[] buf = new char[end - start];
unusualCat.getChars(start, end, buf, 0);
infoTextView.setText(new String(buf));

```

### **public int indexOf(int c)**

Возвращает номер первой встречной позиции с указанным индексом **c**.

```

String testString = "котёнок";
// символ ё встречается в четвёртой позиции (index =
3)
infoTextView.setText(String.valueOf(testString.indexO
f("ё")));

```

### **public int indexOf (int c, int start)**

Ищет индекс **c**, начиная с позиции **start**

```

String testString = "котёнок";
// вернёт -1, так как после 5 символа буквы ё нет
infoTextView.setText(String.valueOf(testString.indexO
f("ё", 4)));

```

### **public int indexOf (String string)**

Ищет цепочку символов **subString**

```
String testString = "У окошка";
infoTextView.setText(String.valueOf(testString.indexOf("кошка")));
```

**public int indexOf (String subString, int start)**

Ищет цепочку символов **subString**, начиная с позиции **start**

```
String testString = "У окошка";
infoTextView.setText(String.valueOf(testString.indexOf("кошка", 2)));
```

**public boolean isEmpty ()**

Проверяет, является ли строка пустой

```
if (catname.isEmpty()) {
    // здесь ваш код
}
```

Данный метод появился в API 9 (Android 2.1). Для старых устройств используйте **String.length() == 0**

**public int lastIndexOf (String string)** и другие перегруженные версии

Возвращает номер последней встречной позиции с указанным индексом.

Например, получить имя файла без расширения можно так:

```
filename.toString().substring(0,
filename.getString().lastIndexOf("."));
```

В этом примере мы получаем позицию последней точки и получаем подстроку до неё.

**public int length()**

Возвращает длину строки

```
String testString = "котёнок";
```

```
infoTextView.setText(String.valueOf(testString.length())); // возвращает 7 (семь символов)
```

**public String replace(CharSequence target, CharSequence replacement)** и

другие перегруженные версии

Меняет символ или последовательность символов **target** на **replacement**

```
String testString = "КИТ";
// меняем и на о
infoTextView.setText(testString.replace("и", "о"));
// возвращается кот
```

**public String replaceAll (String regularExpression, String replacement)**

```

        int i =
Integer.parseInt("kitty123".replaceAll("[\\D]", ""));
        int j =
Integer.parseInt("123kitty".replaceAll("[\\D]", ""));
        int k =
Integer.parseInt("1k2it3ty".replaceAll("[\\D]", ""));

```

// Вернет

i = 123;

j = 123;

k = 123;

**public String replaceFirst (String regularExpression, String replacement)**

Удаляет первые символы при помощи регулярного выражения.

Например, если нужно удалить нули в начале чисел 001, 007, 000024, то можно использовать такой вызов.

```

String s = "001234-cat";
String s = s.replaceFirst ("^0*", ""); // останется
1234-cat

```

**public String[] split (String regularExpression) и другие перегруженные версии**

Разбивает строку на массив из слов. Например, есть строка **Васька Рыжик Мурзик Барсик** и мы хотим получить массив имён котиков:

```

String catnames = "Васька Рыжик Мурзик Барсик";
String aCats[] = catnames.split(" "); // по пробелу

```

Получим:

```

aCats[0] = Васька
aCats[1] = Рыжик
aCats[2] = Мурзик
aCats[3] = Барсик

```

**public boolean startsWith (String prefix)**

Проверяет, начинается ли строка символами *prefix* с начала строки

```
String str1 = "котлета";
```

```
if (str1.startsWith("кот"))
```

```
    infoTextView.setText("Слово содержит кота");
```

```
else
```

```
    infoTextView.setText("Плохое слово. Нет смысла
его использовать");
```

**public boolean startsWith (String prefix, int start)**

Проверяет, начинается ли заданная строка символами *prefix* с указанной позиции.

```
String str1 = "Суккот";

if(str1.startsWith("кот", 3))
    infoTextView.setText("Слово содержит кота");
else
    infoTextView.setText("Плохое слово. Нет смысла
его использовать");
```

### **public String substring(int start) и другие перегруженные версии.**

Создаёт новую последовательность/строку с символами из данной строки начиная с позиции **start** до конца строки/заканчивая символом с позиции **end**. Новая строка содержит символы от **start** до **end** - 1, поэтому берём на один символ больше.

```
String testString = "скотина";

infoTextView.setText(testString.substring(1, 4)); //
возвращается кот
```

### **public char[] toCharArray()**

Копирует символы в этой строке в массив символов. Тот же результат можно получить через метод `getChars()`. Документация не рекомендует использовать данный метод, предлагая метод `charAt()`.

```
String unusualCat = "Котёнок по имени Гав";
```

```
char[] yomoe = unusualCat.toCharArray();
infoTextView.setText(String.valueOf(yomoe[3]));
```

### **public String toLowerCase() и другие перегруженные версии**

Преобразовывает строку в нижний регистр. Преобразованием управляет заданный по умолчанию региональный язык.

```
String cat = "Кот";
String lower = cat.toLowerCase();
infoTextView.setText(lower);
```

### **public String toString()**

Возвращает строку. Для самой строки, которая сама уже является строкой, возвращать строку бессмысленно (о, как я загнул). Но на самом деле этот метод очень полезен для других классов.

### **public String toUpperCase()**

Преобразовывает строку в верхний регистр. Преобразованием управляет заданный по умолчанию региональный язык.



```
String cat = "Кот";
String upper = cat.toUpperCase();
infoTextView.setText(upper);
```

### **public String trim()**

Удаляет пробелы в начале и в конце строки.

```
String str = "    Hello Kitty    ".trim();
infoTextView.setText(str);
```

### **public static String.valueOf(long value) и другие перегруженные версии**

Конвертирует содержимое (числа, объекты, символы, массивы символов) в строку.

```
int catAge = 7; // это число

infoTextView.setText(String.valueOf(catAge)); //
преобразовано в строку
```

### **Генерируем случайную строку**

Допустим, нам нужна случайная строка из заданных символов.

```
private static final String mCHAR =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
private static final int STR_LENGTH = 9; // длина
генерируемой строки
Random random = new Random();

public void onClick(View view) {
    TextView infoTextView = (TextView)
findViewById(R.id.textviewInfo);
    infoTextView.setText(createRandomString());
}

public String createRandomString() {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < STR_LENGTH; i++) {
        int number = random.nextInt(mCHAR.length());
        char ch = mCHAR.charAt(number);
        builder.append(ch);
    }
    return builder.toString();
}
```

### **Сравнение строк: equals() или ==?**

Рассмотрим пример.

```
String str1 = "Мурзик";
String str2 = new String(str1);
boolean isCat = str1 == str2;
```

```
infoTextView.setText(str1 + " == " + str2 + " -> " +
isCat);
```

Хотя в двух переменных содержится одно и то же слово, мы имеем дело с двумя разными объектами и оператор `==` вернёт *false*.

Однажды, когда деревья были большими, мне понадобилось сравнить две строки из разных источников. Хотя строки выглядели совершенно одинаково, сравнение при помощи оператора `==` возвращало *false* и путало мне все карты. И только потом я узнал, что нужно использовать метод **`equals()`**. Строка в Java - это отдельный объект, который может не совпадать с другим объектом, хотя на экране результат выводимой строки может выглядеть одинаково. Просто Java в случае с логическим оператором `==` (а также `!=`) сравнивает ссылки на объекты (при работе с примитивами такой проблемы нет):

```
String s1 = "hello";
String s2 = "hello";
String s3 = s1;
String s4 = "h" + "e" + "l" + "l" + "o";
String s5 = new String("hello");
String s6 = new String(new char[]{'h', 'e', 'l', 'l',
'o'});

infoTextView.setText(s1 + " == " + s2 + ": " + (s1 ==
s2));
// попробуйте и другие варианты
// infoTextView.setText(s1 + " equals " + s2 + ": " +
(s1.equals(s2)));
// infoTextView.setText(s1 + " == " + s3 + ": " + (s1
== s3));
// infoTextView.setText(s1 + " equals " + s3 + ": " +
(s1.equals(s3)));
// infoTextView.setText(s1 + " == " + s4 + ": " + (s1
== s4));
// infoTextView.setText(s1 + " equals " + s4 + ": " +
(s1.equals(s4)));
// infoTextView.setText(s1 + " == " + s5 + ": " + (s1
== s5)); // false
// infoTextView.setText(s1 + " equals " + s5 + ": " +
(s1.equals(s5)));
// infoTextView.setText(s1 + " == " + s6 + ": " + (s1
== s6)); // false
```

```
// infoTextView.setText(s1 + " equals " + s6 + ": " +  
(s1.equals(s6)));
```

## Классы **StringBuffer** и **StringBuilder**

Класс **String** представляет собой неизменяемые последовательности символов постоянной длины и частое использование объектов класса занимают много места в памяти. Класс **StringBuffer** представляет расширяемые и доступные для изменений последовательности символов, позволяя вставлять символы и подстроки в существующую строку и в любом месте. Данный класс гораздо экономичнее в плане потребления памяти и настоятельно рекомендуется к использованию.

Существует четыре конструктора класса:

1. **StringBuffer()** - резервирует место под 16 символов без перераспределения памяти
2. **StringBuffer(int capacity)** - явно устанавливает размер буфера
3. **StringBuffer(String string)** - устанавливает начальное содержимое и резервирует 16 символов без повторного резервирования
4. **StringBuffer(CharSequence cs)** - создаёт объект, содержащий последовательность символов и резервирует место ещё под 16 символов

## Методы класса **StringBuffer**

### **length()**

Метод позволяет получить текущую длину объекта.

```
StringBuffer sb = new StringBuffer("Котэ");  
tvInfo.setText("Длина: " + sb.length());
```

### **capacity()**

Метод позволяет получить текущий объём выделенной памяти.

```
StringBuffer sb = new StringBuffer("Котэ");  
tvInfo.setText("Объём памяти: " + sb.capacity()); //  
вернёт 20
```

Обратите внимание, что хотя слово состоит из четырёх символов, в памяти выделено запасное пространство для дополнительных 16 символов. Для такой простейшей операции выигрыша нет, но в сложных примерах, когда приходится на лету соединять множество строк, производительность резко возрастает.

### **ensureCapacity()**

Можно предварительно выделить место для определённого количества символов, если собираетесь добавлять большое количество маленьких строк.

### **setLength(int length)**

Устанавливает длину строки. Значение должно быть неотрицательным.

### **charAt(int index) и setCharAt(int index, char ch)**

Можно извлечь значение отдельного символа с помощью метода **charAt()** и установить новое значение символа с помощью метода **setCharAt()**, указав индекс символа и его значение.

```
StringBuffer sb = new StringBuffer("Кит");  
sb.setCharAt(1, 'о');  
tvInfo.setText(sb.toString());
```

### **getChars()**

Позволяет скопировать подстроку из объекта класса **StringBuffer** в массив. Необходимо позаботиться, чтобы массив был достаточного размера для приёма нужного количества символов указанной подстроки.

### **append()**

Метод соединяет представление любого другого типа данных. Есть несколько перегруженных версий.

```
StringBuffer append(String string)  
StringBuffer append(int number)  
StringBuffer append(Object object)
```

Строковое представление каждого параметра за кулисами получают через метод **String.valueOf()** и затем полученные строки склеиваются в итоговую строку.

```
String str1 = "У кота ";  
String str2 = " лапы";  
int paws = 4;  
StringBuffer sb = new StringBuffer(20);  
sb.append(str1).append(paws).append(str2);  
tvInfo.setText(sb.toString());
```

### **insert()**

Вставляет одну строку в другую. Также можно вставлять значения других типов, которые будут автоматически преобразованы в строки. Вам надо указать индекс позиции, куда будет вставляться строка.

```
StringBuffer sb = new StringBuffer("Я Котов");  
sb.insert(2, "Люблю ");  
tvInfo.setText(sb.toString());
```

### **reverse()**

Позволяет изменить порядок символов на обратный.

```
StringBuffer sb = new StringBuffer("МОКНЕТ ОКСАНА С  
КОТЕНКОМ");  
sb.reverse();  
tvInfo.setText(sb.toString());
```

### **delete() и deleteCharAt()**

Метод **delete()** удаляет последовательность символов, вам надо задать индекс первого символа, который надо удалить, а также индекс символа, следующего за последним из удаляемых. Метод **deleteCharAt()** удаляет один символ из указанной позиции.

### **replace()**

Позволяет заменить один набор символов на другой. Нужно указать начальный и конечный индекс и строку замены.

### **substring()**

Позволяет получить часть содержимого. Есть две формы метода. В первом случае нужно указать индекс начала позиции, с которой нужно получить подстроку. Во втором варианте указывается начальный индекс и конечный индекс, если нужно получить текст из середины строки.

## **StringBuilder**

Класс **StringBuilder** идентичен классу **StringBuffer** и обладает большей производительностью. Однако он не синхронизирован, поэтому его не нужно использовать в тех случаях, когда к изменяемой строке обращаются несколько потоков.

Создадим новый объект.

```
StringBuilder builder = new StringBuilder();
```

Добавляем новый фрагмент в существующую строку:

```
builder.append(ch); // можно добавить один символ
builder.append(sometext); // а можно добавить готовую
строку
String completedString = builder.toString(); //
результатирующая строка
```

Соединять строки можно цепочкой.

```
new
StringBuilder().append(s1).append(s2).append(s3).toString
();
```

Данный код работает чуть быстрее, чем вызов **append()** по отдельности. Но это будет заметно при очень больших объёмах.

## **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;

4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Как создать строку с помощью класса String?
2. Как создать строку с помощью класса StringBuffer?
3. Как создать строку с помощью класса StringBuilder?
4. В чем различия между строками String, StringBuffer, StringBuilder?

## **8. Рекомендуемая литература**

73. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
74. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
75. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
76. Лафоре Р. Структуры данных и алгоритмы на Java – СПб: Питер, 2013. – 704 с.
77. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
78. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Обработка исключительных ситуаций»

Минск  
2018

## **Лабораторная работа № 14**

### **Тема работы: «Обработка исключительных ситуаций»**

#### **1. Цель работы**

Отрабатывает навык генерирования и отслеживания исключительных ситуаций.

#### **2. Задание**

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 12 реализовать обработку исключительных ситуаций, при этом организовать:

- ✓ завершение работы программы, с выводом сообщения об ошибке;
- ✓ вывода сообщения об ошибке, и организовать безопасное продолжение работы программ;
- ✓ генерирование собственного исключения.

#### **3. Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### **4. Основные теоретические сведения**

##### **Исключения**

Исключение - это нештатная ситуация, ошибка во время выполнения программы. Самый простой пример - деление на ноль. Можно вручную отслеживать возникновение подобных ошибок, а можно воспользоваться специальным механизмом исключений, который упрощает создание больших надёжных программ, уменьшает объём необходимого кода и повышает уверенность в том, что в приложении не будет необработанной ошибки.

В методе, в котором происходит ошибка, создаётся и передаётся специальный объект. Метод может либо обработать исключение самостоятельно, либо пропустить его. В любом случае исключение ловится и обрабатывается. Исключение может появиться благодаря самой системе, либо вы сами можете создать его вручную. Системные исключения возникают при неправильном использовании языка Java или запрещённых приёмов доступа к системе. Ваши собственные исключения обрабатывают специфические ошибки вашей программы.



Вернёмся к примеру с делением. Деление на ноль может предотвратить проверкой соответствующего условия. Но что делать, если знаменатель оказался нулём? Возможно, в контексте вашей задачи известно, как следует поступить в такой ситуации. Но, если нулевой знаменатель возник неожиданно, деление в принципе невозможно, и тогда необходимо возбудить исключение, а не продолжать исполнение программы.

Существует пять ключевых слов, используемых в исключениях: **try**, **catch**, **throw**, **throws**, **finally**. Порядок обработки исключений следующий.

Операторы программы, которые вы хотите отслеживать, помещаются в блок **try**. Если исключение произошло, то оно создаётся и передаётся дальше. Ваш код может перехватить исключение при помощи блока **catch** и обработать его. Системные исключения автоматически передаются самой системой. Чтобы передать исключение вручную, используется **throw**. Любое исключение, созданное и передаваемое внутри метода, должно быть указано в его интерфейсе ключевым словом **throws**. Любой код, который следует выполнить обязательно после завершения блока **try**, помещается в блок **finally**

Схематически код выглядит так:

```
try {  
    // блок кода, где отслеживаются ошибки  
}  
catch (тип_исключения_1 exceptionObject) {  
    // обрабатываем ошибку  
}  
catch (тип_исключения_2 exceptionObject) {  
    // обрабатываем ошибку  
}  
finally {  
    // код, который нужно выполнить после завершения  
    блока try  
}
```

Существует специальный класс для исключений **Trowable**. В него входят два класса **Exception** и **Error**.

Класс **Exception** используется для обработки исключений вашей программой. Вы можете наследоваться от него для создания собственных типов исключений. Для распространённых ошибок уже существует класс **RuntimeException**, который может обрабатывать деление на ноль или определять ошибочную индексацию массива.

Класс **Error** служит для обработки ошибок в самом языке **Java** и на практике вам не придётся иметь с ним дело.

Прежде чем научиться обрабатывать исключения, нам (как и нормальному любопытному коту) хочется посмотреть, а что происходит, если

ошибку не обработать. Давайте разделим число котов в вашей квартире на ноль, хотя мы и знаем, что котов на ноль делить нельзя!

```
int catNumber;  
int zero;  
catNumber = 1; // у меня один кот  
zero = 0; // ноль, он и в Африке ноль  
int result = catNumber / zero;
```

Я поместил код в обработчик щелчка кнопки. Когда система времени выполнения Java обнаруживает попытку деления на ноль, она создаёт объект исключения и передаёт его. Да вот незадача, никто не перехватывает его, хотя это должны были сделать вы. Видя вашу бездеятельность, объект перехватывает стандартный системный обработчик Java, который отличается вредным характером. Он останавливает вашу программу и выводит сообщение об ошибке, которое можно увидеть в журнале LogCat:

```
Caused by: java.lang.ArithmeticException: divide by  
zero at  
ru.alexanderklimov.test.MainActivity.onClick(MainActivity  
.java:79)
```

Как видно, созданный объект исключения принадлежит к классу **ArithmeticException**, далее системный обработчик любезно вывел краткое описание ошибки и место возникновения.

Вряд ли пользователи вашей программы будут довольны, если вы так и оставите обработку ошибки системе. Если программа будет завершаться с такой ошибкой, то скорее всего вашу программу просто удалят. Посмотрим, как мы можем исправить ситуацию.

Поместим проблемный код в блок **try**, а в блоке **catch** обработаем исключение.

```
int catNumber;  
int zero;  
  
try { // мониторим код  
    catNumber = 1; // у меня один кот  
    zero = 0; // ноль, он и в Африке ноль  
    int result = catNumber / zero;  
    Toast.makeText(this, "Не увидите это сообщение!",  
Toast.LENGTH_LONG).show();  
} catch (ArithmeticException e) {  
    Toast.makeText(this, "Нельзя котов делить на  
ноль!", Toast.LENGTH_LONG).show();  
}  
    Toast.makeText(this, "Жизнь продолжается",  
Toast.LENGTH_LONG).show();
```

Теперь программа аварийно не закрывается, так как мы обрабатываем ситуацию с делением на ноль.

В данном случае мы уже знали, к какому классу принадлежит получаемая ошибка, поэтому в блоке **catch** сразу указали конкретный тип. Обратите внимание, что последний оператор в блоке **try** не срабатывает, так как ошибка происходит раньше строчкой выше. Далее выполнение передаётся в блок **catch**, далее выполняются следующие операторы в обычном порядке.

Операторы **try** и **catch** работают совместно в паре. Хотя возможны ситуации, когда **catch** может обрабатывать несколько вложенных операторов **try**.

Если вы хотите увидеть описание ошибки, то параметр **e** и поможет увидеть её.

```
catch (ArithmeticException e) {  
    Toast.makeText(this, e + ": Нельзя котов делить  
на ноль!", Toast.LENGTH_LONG).show();  
}
```

По умолчанию, класс **Trowable**, к которому относится **ArithmeticException** возвращает строку, содержащую описание исключения. Но вы можете и явно указать метод **e.toString**.

### Несколько исключений

Фрагмент кода может содержать несколько проблемных мест. Например, кроме деления на ноль, возможна ошибка индексации массива. В таком случае вам нужно создать два или более операторов **catch** для каждого типа исключения. Причём они проверяются по порядку. Если исключение будет обнаружено у первого блока обработки, то он будет выполнен, а остальные проверки пропускаются и выполнение программы продолжается с места, который следует за блоком **try/catch**.

```
int catNumber;  
int zero;  
  
try { // мониторим код  
    catNumber = 1; // у меня один кот  
    zero = 1; // ноль, он и в Африке ноль  
    int result = catNumber / zero;  
    // Создадим массив из трёх котов  
    String[] catNames = {"Васька", "Барсик",  
"Мурзик"};  
    catNames[3] = "Рыжик";  
    Toast.makeText(this, "Не увидите это сообщение!",  
Toast.LENGTH_LONG).show();  
} catch (ArithmeticException e) {
```

```

        Toast.makeText(this, e.toString() + ": Нельзя
котов делить на ноль!", Toast.LENGTH_LONG).show();
    }
    catch (ArrayIndexOutOfBoundsException e) {
        Toast.makeText(this, "Ошибка: " + e.toString(),
Toast.LENGTH_LONG).show();
    }
    Toast.makeText(this, "Жизнь продолжается",
Toast.LENGTH_LONG).show();

```

В примере мы добавили массив с тремя элементами, но обращаемся к четвёртому элементу, так как забыли, что отсчёт у массива начинается с нуля. Если оставить значение переменной **zero** равным нулю, то сработает обработка первого исключения деления на ноль, и мы даже не узнаем о существовании второй ошибки. Но допустим, что в результате каких-то вычислений значение переменной стало равно единице. Тогда наше исключение **ArithmeticException** не сработает. Но сработает новое добавленное исключение **ArrayIndexOutOfBoundsException**. А дальше всё пойдёт как раньше.

Тут всегда нужно помнить одну особенность. При использовании множественных операторов **catch** обработчики подклассов исключений должны находиться выше, чем обработчики их суперклассов. Иначе, суперкласс будет перехватывать все исключения, имея большую область перехвата. Иными словами, **Exception** не должен находиться выше **ArithmeticException** и **ArrayIndexOutOfBoundsException**. К счастью, среда разработки сама замечает непорядок и предупреждает вас, что такой порядок не годится. Увидев такую ошибку, попробуйте перенести блок обработки исключений ниже.

### Вложенные операторы try

Операторы **try** могут быть вложенными. Если вложенный оператор **try** не имеет своего обработчика **catch** для определения исключения, то идёт поиск обработчика **catch** у внешнего блока **try** и т.д. Если подходящий **catch** не будет найден, то исключение обработает сама система (что никуда не годится).

### Оператор throw

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора **throw**. Код выглядит так:

```
throw экземпляр_Throwable
```

Вам нужно создать экземпляр класса **Throwable** или его наследников. Получить объект класса **Throwable** можно в операторе **catch** или стандартным способом через оператор **new**.

Мы могли бы написать такой код для кнопки:

```

Cat cat;

public void onClick(View view) {
    if(cat == null){
        throw new NullPointerException("Котик не
инициализирован");
    }
}

```

Мы объявили объект класса **Cat**, но забыли его проинициализировать, например, в **onCreate()**. Теперь нажатие кнопки вызовет исключение, которое обработает система, а в логах мы можем прочесть сообщение об ошибке. Возможно, вы захотите использовать другое исключение, например, **throw new UnsupportedOperationException("Котик не инициализирован");**.

В любом случае мы передали обработку ошибки системе. В реальном приложении вам нужно обработать ошибку самостоятельно.

Поток выполнения останавливается непосредственно после оператора **throw** и другие операторы не выполняются. При этом ищется ближайший блок **try/catch** соответствующего исключению типа.

Перепишем пример с обработкой ошибки.

```

public void onClick(View view) {
    if (cat == null) {
        try {
            throw new NullPointerException("Кота не
существует");
        } catch (NullPointerException e) {
            Toast.makeText(this, e.getMessage(),
Toast.LENGTH_LONG).show();
        }
    }
}

```

Мы создали новый объект класса **NullPointerException**. Многие классы исключений кроме стандартного конструктора по умолчанию с пустыми скобками имеют второй конструктор с строковым параметром, в котором можно разместить подходящую информацию об исключении. Получить текст из него можно через метод **getMessage()**, что мы и сделали в блоке **catch**.

Теперь программа не закроется аварийно, а будет просто выводить сообщения в всплывающих **Toast**.

### Оператор throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен задать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Для этого к объявлению метода

добавляется конструкция **throws**, которая перечисляет типы исключений (кроме исключений `Error` и `RuntimeException` и их подклассов).

Общая форма объявления метода с оператором **throws**:

```
тип            имя_метода (список_параметров)            throws
список_исключений {
    // код внутри метода
}
```

В фрагменте *список\_исключений* можно указать список исключений через запятую.

Создадим метод, который может породить исключение, но не обрабатывает его. А в щелчке кнопки вызовем его.

```
// Метод без обработки исключения
public void createCat() {
    Toast.makeText(this, "Вы создали котёнка",
Toast.LENGTH_LONG).show();
    throw new NullPointerException("Кота не
существует");
}
```

```
// Щелчок кнопки
public void onClick(View v) {
    createCat();
}
```

Если вы запустите пример, то получите ошибку. Исправим код.

```
// Без изменений
public void createCat() throws NullPointerException {
    Toast.makeText(this, "Вы создали котёнка",
Toast.LENGTH_LONG).show();
    throw new NullPointerException("Кота не
существует");
}
```

```
// Щелчок кнопки
public void onClick(View v) {
    try {
        createCat();
    } catch (NullPointerException e) {
        // TODO: handle exception
        Toast.makeText(this, e.getMessage(),
Toast.LENGTH_LONG).show();
    }
}
```

}

Мы поместили вызов метода в блок **try** и вызвали блок **catch** с нужным типом исключения. Теперь ошибки не будет.

### Оператор **finally**

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм **finally**.

Ключевое слово **finally** создаёт блок кода, который будет выполнен после завершения блока **try/catch**, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет. Оператор **finally** не обязателен, однако каждый оператор **try** требует наличия либо **catch**, либо **finally**.

### Встроенные исключения Java

Существуют несколько готовых системных исключений. Большинство из них являются подклассами типа **RuntimeException** и их не нужно включать в список **throws**. Вот небольшой список непроверяемых исключений.

- ✓ **ArithmeticException** - арифметическая ошибка, например, деление на ноль
- ✓ **ArrayIndexOutOfBoundsException** - выход индекса за границу массива
- ✓ **ArrayStoreException** - присваивание элементу массива объекта несовместимого типа
- ✓ **ClassCastException** - неверное приведение
- ✓ **EnumConstantNotPresentException** - попытка использования неопределённого значения перечисления
- ✓ **IllegalArgumentException** - неверный аргумент при вызове метода
- ✓ **IllegalMonitorStateException** - неверная операция мониторинга
- ✓ **IllegalStateException** - некорректное состояние приложения
- ✓ **IllegalThreadStateException** - запрашиваемая операция несовместима с текущим потоком
- ✓ **IndexOutOfBoundsException** - тип индекса вышел за допустимые пределы
- ✓ **NegativeArraySizeException** - создан массив отрицательного размера
- ✓ **NullPointerException** - неверное использование пустой ссылки
- ✓ **NumberFormatException** - неверное преобразование строки в числовой формат
- ✓ **SecurityException** - попытка нарушения безопасности
- ✓ **StringIndexOutOfBoundsException** - попытка использования индекса за пределами строки
- ✓ **TypeNotPresentException** - тип не найден

- ✓ `UnsupportedOperationException` - обнаружена неподдерживаемая операция

Список проверяемых системных исключений, которые можно включать в список **throws**.

- ✓ `ClassNotFoundException` - класс не найден
- ✓ `CloneNotSupportedException` - попытка клонировать объект, который не реализует интерфейс **Cloneable**
- ✓ `IllegalAccessException` - запрещен доступ к классу
- ✓ `InstantiationException` - попытка создать объект абстрактного класса или интерфейса
- ✓ `InterruptedException` - поток прерван другим потоком
- ✓ `NoSuchFieldException` - запрашиваемое поле не существует
- ✓ `NoSuchMethodException` - запрашиваемый метод не существует
- ✓ `ReflectiveOperationException` - исключение, связанное с рефлексией

### Создание собственных классов исключений

Система не может предусмотреть все исключения, иногда вам придётся создать собственный тип исключения для вашего приложения. Вам нужно наследоваться от **Exception** (напомню, что этот класс наследуется от **Throwable**) и переопределить нужные методы класса **Throwable**. Либо вы можете унаследоваться от уже существующего типа, который наиболее близок по логике с вашим исключением.

- ✓ `final void addSuppressed(Throwable exception)` - добавляет исключение в список подавляемых исключений (JDK 7)
- ✓ `Throwable fillInStackTrace()` - возвращает объект класса **Throwable**, содержащий полную трассировку стека.
- ✓ `Throwable getCause()` - возвращает исключение, лежащее под текущим исключением или `null`
- ✓ `String getLocalizedMessage()` - возвращает локализованное описание исключения
- ✓ `String getMessage()` - возвращает описание исключения
- ✓ `StackTraceElement[] getStackTrace()` - возвращает массив, содержащий трассировку стека и состояний из элементов класса **StackTraceElement**
- ✓ `final Throwable[] getSuppressed()` - получает подавленные исключения (JDK 7)
- ✓ `Throwable initCause(Throwable exception)` - ассоциирует исключение с вызывающим исключением. Возвращает ссылку на исключение.
- ✓ `void printStackTrace()` - отображает трассировку стека
- ✓ `void printStackTrace(PrintStream stream)` - посылает трассировку стека в заданный поток
- ✓ `void printStackTrace(PrintWriter stream)` - посылает трассировку стека в заданный поток



- ✓ `void setStackTrace(StackTraceElement elements[])` - устанавливает трассировку стека для элементов (для специализированных приложений)
- ✓ `String toString()` - возвращает объект класса **String**, содержащий описание исключения.

Самый простой способ - создать класс с конструктором по умолчанию.

```
// Если этот код работает, его написал Александр
Климов,
// а если нет, то не знаю, кто его писал.
```

```
package ru.alexanderklimov.exception;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void testMethod() throws
HungryCatException{
        System.out.println("Возбуждаем
HungryCatException из метода testMethod()");
        throw new HungryCatException(); //
конструктор по умолчанию
    }

    public void onClick(View view) {
        try {
            testMethod();
        } catch (HungryCatException e) {
            e.printStackTrace();
            System.out.println("Наше исключение
перехвачено");
        }
    }
}
```

```

        class HungryCatException extends Exception{
        }
    }

```

Мы создали собственный класс **HungryCatException**, в методе **testMethod()** его возбуждаем, а по нажатию кнопки вызываем этот метод. В результате наше исключение сработает.

Создать класс исключения с конструктором, который получает аргумент-строку, также просто.

```

// Если этот код работает, его написал Александр
КЛИМОВ,
// а если нет, то не знаю, кто его писал.

```

```

package ru.alexanderklimov.exception;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void testMethod() throws
HungryCatException {
        System.out.println("Возбуждаем
HungryCatException из метода testMethod()");
        throw new HungryCatException(); //
конструктор по умолчанию
    }

    public void testMethod2() throws
HungryCatException {
        System.out.println("Возбуждаем
HungryCatException из метода testMethod2()");
        throw new HungryCatException("Создано во
втором методе");
    }
}

```

```

        public void onClick(View view) {
            try {
                testMethod();
            } catch (HungryCatException e) {
                e.printStackTrace();
                System.out.println("Наше исключение
перехвачено");
            }

            try {
                testMethod2();
            } catch (HungryCatException e) {
                e.printStackTrace();
            }
        }

        class HungryCatException extends Exception {
            HungryCatException() {
            }

            HungryCatException(String msg) {
                super(msg);
            }
        }
    }

```

Теперь класс содержит два конструктора. Во втором конструкторе используется конструктор родительского класса с аргументом **String**, вызываемый ключевым словом **super**.

### Перехват произвольных исключений

Можно создать универсальный обработчик, перехватывающий любые типы исключения. Осуществляется это перехватом базового класса всех исключений **Exception**:

```

    catch(Exception e) {
        Log.w("Log", "Перехвачено исключение");
    }

```

Подобная конструкция не упустит ни одного исключения, поэтому её следует размещать в самом конце списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

### Основные правила обработки исключений

Используйте исключения для того, чтобы:

- ✓ обработать ошибку на текущем уровне (избегайте перехватывать исключения, если не знаете, как с ними поступить)

- ✓ исправить проблему и снова вызвать метод, возбудивший исключение
- ✓ предпринять все необходимые действия и продолжить выполнение без повторного вызова действия
- ✓ попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод
- ✓ сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень
- ✓ сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень
- ✓ завершить работу программы
- ✓ упростить программу (если используемая схема обработки исключений делает все только сложнее, значит, она никуда не годится)
- ✓ добавить вашей библиотеке и программе безопасности

## **5. Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## **6. Форма отчета о работе**

*Лабораторная работа № \_\_\_\_*

*Номер учебной группы* \_\_\_\_\_

*Фамилия, инициалы учащегося* \_\_\_\_\_

*Дата выполнения работы* \_\_\_\_\_

*Тема работы:* \_\_\_\_\_

*Цель работы:* \_\_\_\_\_

*Оснащение работы:* \_\_\_\_\_

*Результат выполнения работы:* \_\_\_\_\_

\_\_\_\_\_

## **7. Контрольные вопросы и задания**

1. Что такое исключительная ситуация?
2. Каких видов существуют исключительные ситуации?
3. Как отловить исключительную ситуацию?
4. Как сгенерировать исключительную ситуацию?

## **8. Рекомендуемая литература**

- 79. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
- 80. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
- 81. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
- 82. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
- 83. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
- 84. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Поточная модель»

Минск  
2018

## Лабораторная работа № 15

### Тема работы: «Поточная модель»

#### 1. Цель работы

Закрепляет навыки создания и управления потоками.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 14 реализовать многопоточную работу приложения (потоки не синхронизированные).

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Потоки. Класс **Thread** и интерфейс **Runnable**

В русской терминологии за термином **Thread** укрепился перевод "Поток". Хотя это слово также можно перевести как "Нить". Иногда в зарубежных учебных материалах понятие потока объясняется именно на нитях. Продолжим логический ряд - там где нити, там и клубок. А где клубок, там и кот. Сразу видно, что у переводчиков не было котов. Так и возникла путаница. Тем более что существуют другие потоки под термином **Stream**. Переводчики, вообще странный народ.

Когда запускается любое приложение, то начинает выполняться поток, называемый *главным потоком* (**main**). От него порождаются дочерние потоки. Главный поток, как правило, является последним потоком, завершающим выполнение программы.

Несмотря на то, что главный поток создаётся автоматически, им можно управлять через объект класса **Thread**. Для этого нужно вызвать метод **currentThread()**, после чего можно управлять потоком.

Класс **Thread** содержит несколько методов для управления потоками.

- ✓ **getName()** - получить имя потока
- ✓ **getPriority()** - получить приоритет потока
- ✓ **isAlive()** - определить, выполняется ли поток
- ✓ **join()** - ожидать завершения потока
- ✓ **run()** - запуск потока. В нём пишете свой код
- ✓ **sleep()** - приостановить поток на заданное время

✓ **start()** - запустить поток

Получим информацию о главном потоке и поменяем его имя.

```
Thread mainThread = Thread.currentThread();
mInfoTextView.setText("Текущий поток: " +
mainThread.getName());
```

```
// Меняем имя и выводим в текстовом поле
mainThread.setName("CatThread");
mInfoTextView.append("\nНовое имя потока: " +
mainThread.getName());
```

Имя у главного потока по умолчанию **main**, которое мы заменили на **CatThread**.

Вызовем информацию о названии потока без указания метода.

```
Thread mainThread = Thread.currentThread();
mInfoTextView.setText("Текущий поток: " +
mainThread);
```

В этом случае можно увидеть строчку **Thread[main,5,main]** - имя потока, его приоритет и имя его группы.

### Создание собственного потока

Создать собственный поток не сложно. Достаточно наследоваться от класса **Thread**.

Объявим внутри нашего класса внутренний класс и вызовем его по щелчку, вызвав метод **start()**.

```
public class MyThread extends Thread {
    public void run() {
        Log.d(TAG, "Мой поток запущен...");
    }
}
```

```
public void onClick(View view) {
    MyThread myThread = new MyThread();
    myThread.start();
}
```

Как вариант, перенести вызов метода **start()** в конструктор.

```
public void onClick(View view) {
    MyThread myThread = new MyThread();
}
```

```
public class MyThread extends Thread {
```



```

// Конструктор
MyThread() {
    // Создаём новый поток
    super("Второй поток");
    Log.i(TAG, "Создан второй поток " + this);
    start(); // Запускаем поток
}

public void run() {
    Log.d(TAG, "Мой поток запущен...");

    try {
        for (int i = 5; i > 0; i--) {
            Log.i(TAG, "Второй поток: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        Log.i(TAG, "Второй поток прерван");
    }
}
}

```

### Создание потока с интерфейсом Runnable

Есть более сложный вариант создания потока. Для создания нового потока нужно реализовать интерфейс **Runnable**. Вы можете создать поток из любого объекта, реализующего интерфейс **Runnable** и объявить метод **run()**.

Внутри метода **run()** вы размещаете код для нового потока. Этот поток завершится, когда метод вернёт управление.

Когда вы объявите новый класс с интерфейсом **Runnable**, вам нужно использовать конструктор:

```
Thread(Runnable объект_потока, String имя_потока)
```

В первом параметре указывается экземпляр класса, реализующего интерфейс. Он определяет, где начнётся выполнение потока. Во втором параметре передаётся имя потока.

После создания нового потока, его нужно запустить с помощью метода **start()**, который, по сути, выполняет вызов метода **run()**.

Создадим новый поток внутри учебного проекта в виде вложенного класса и запустим его.

```

package ru.alexanderklimov.expresscourse;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

```

```

import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import static
ru.alexanderklimov.expresscourse.R.id.textViewInfo;

public class MainActivity extends AppCompatActivity {

    final String TAG = "ExpressCourse";

    private Button mButton;
    private EditText mResultEditText;
    private TextView mInfoTextView;

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mButton = (Button)
findViewById(R.id.buttonGetResult);
        mResultEditText = (EditText)
findViewById(R.id.editText);
        mInfoTextView = (TextView)
findViewById(textViewInfo);
    }

    public void onClick(View view) {
        new MyRunnable(); // создаём новый поток

        try {
            for (int i = 5; i > 0; i--) {
                Log.i(TAG, "Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            Log.i(TAG, "Главный поток прерван");
        }
    }
}

```

```

class MyRunnable implements Runnable {
    Thread thread;

    // Конструктор
    MyRunnable() {
        // Создаём новый второй поток
        thread = new Thread(this, "Поток для
примера");
        Log.i(TAG, "Создан второй поток " +
thread);
        thread.start(); // Запускаем поток
    }

    // Обязательный метод для интерфейса Runnable
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                Log.i(TAG, "Второй поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            Log.i(TAG, "Второй поток прерван");
        }
    }
}

```

Внутри конструктора **MyRunnable()** мы создаём новый объект класса **Thread**

```
thread = new Thread(this, "Поток для примера");
```

В первом параметре использовался объект **this**, что означает желание вызвать метод **run()** этого объекта. Далее вызывается метод **start()**, в результате чего запускается выполнение потока, начиная с метода **run()**. В свою очередь метод запускает цикл для нашего потока. После вызова метода **start()**, конструктор **MyRunnable()** возвращает управление приложению. Когда главный поток продолжает свою работу, он входит в свой цикл. После этого оба потока выполняются параллельно.

Можно запускать несколько потоков, а не только второй поток в дополнение к первому. Это может привести к проблемам, когда два потока пытаются работать с одной переменной одновременно.

### Ключевое слово **synchronized** - синхронизированные методы

Для решения проблемы с потоками, которые могут внести путаницу, используется синхронизация.

Метод может иметь модификатор **synchronized**. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его в том же экземпляре, должны ожидать. Это позволяет исключить путаницу, когда несколько потоков пытаются вызвать метод.

```
synchronized void meow(String msg);
```

Кроме того, ключевое слово **synchronized** можно использовать в качестве оператора. Вы можете заключить в блок **synchronized** вызовы методов какого-нибудь класса:

```
synchronized(объект) {  
    // операторы, требующие синхронизации  
}
```

### Looper

Поток имеет в своём составе сущности **Looper**, **Handler**, **MessageQueue**.

Каждый поток имеет один уникальный **Looper** и может иметь много **Handler**.

Считайте **Looper** вспомогательным объектом потока, который управляет им. Он обрабатывает входящие сообщения, а также даёт указание потоку завершиться в нужный момент.

Поток получает свой **Looper** и **MessageQueue** через метод **Looper.prepare()** после запуска. **Looper.prepare()** идентифицирует вызывающий потк, создаёт **Looper** и **MessageQueue** и связывает поток с ними в хранилище **ThreadLocal**. Метод **Looper.loop()** следует вызывать для запуска **Looper**. Завершить его работу можно через метод **looper.quit()**.

```
class LooperThread extends Thread {  
    public Handler mHandler;  
  
    public void run() {  
        Looper.prepare();  
  
        mHandler = new Handler() {  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
            }  
        };  
        Looper.loop();  
    }  
}
```

Используйте статический метод **getMainLooper()** для доступа к **Looper** главного потока:

```
Looper mainLooper = Looper.getMainLooper();
```

Создадим два потока. Один запустим в основном потоке, а второй отдельно от основного. Нам будет достаточно двух кнопок и метки.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android
"

xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:id="@+id/button_start"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="start()" />
    <Button
        android:id="@+id/button_run"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="run()" />
    <TextView
        android:id="@+id/textview_info"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Обратите внимание, как запускаются потоки. Первый поток запускается с помощью метода **start()**, а второй - **run()**. Затем проверяем, в каком потоке мы находимся.

```
package ru.alexanderklimov.as23;

import android.os.Bundle;
import android.os.Looper;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
```

```

public class MainActivity extends AppCompatActivity {

    TextView mInfoTextView;

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button startButton = (Button)
findViewById(R.id.button_start);
        Button runButton = (Button)
findViewById(R.id.button_run);
        mInfoTextView = (TextView)
findViewById(R.id.textview_info);

        startButton.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Thread thread = new Thread(new
MyRunnable());
                thread.start(); //в фоновом потоке
            }
        });

        runButton.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Thread thread = new Thread(new
MyRunnable());
                thread.run(); //в текущем потоке
            }
        });
    }

    private class MyRunnable implements Runnable {

        @Override
        public void run() {
            // Проверяем, в каком потоке находимся

```

```

        if (Looper.getMainLooper().getThread() ==
Thread.currentThread()) {
            mInfoTextView.setText("В основном
потоке");
        } else {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    mInfoTextView.setText("В
фоновом потоке");
                }
            });
        }
    }
}

```

Эта тема достаточно сложная и для большинства не представляет интереса и необходимости изучать.

В Android потоки в чистом виде используются всё реже и реже, у системы есть собственные способы.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

## 7. Контрольные вопросы и задания

1. Что такое паток?
2. Как создать несколько потоков?
3. Как запустить потоки?
4. Как синхронизировать потоки?
5. Для чего необходимо синхронизировать потоки?

## **8. Рекомендуемая литература**

85. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
86. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
87. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
88. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
89. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
90. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.



Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка многопоточных приложений»

Минск  
2018

## Лабораторная работа № 16

### Тема работы: «Разработка многопоточных приложений»

#### 1. Цель работы

Отрабатывает навык создания многопоточных приложений.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 16 реализовать многопоточную работу приложения (все потоки синхронизировать).

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Потоки. Класс **Thread** и интерфейс **Runnable**

В русской терминологии за термином **Thread** укрепился перевод "Поток". Хотя это слово также можно перевести как "Нить". Иногда в зарубежных учебных материалах понятие потока объясняется именно на нитях. Продолжим логический ряд - там где нити, там и клубок. А где клубок, там и кот. Сразу видно, что у переводчиков не было котов. Так и возникла путаница. Тем более что существуют другие потоки под термином **Stream**. Переводчики, вообще странный народ.

Когда запускается любое приложение, то начинает выполняться поток, называемый *главным потоком* (**main**). От него порождаются дочерние потоки. Главный поток, как правило, является последним потоком, завершающим выполнение программы.

Несмотря на то, что главный поток создаётся автоматически, им можно управлять через объект класса **Thread**. Для этого нужно вызвать метод **currentThread()**, после чего можно управлять потоком.

Класс **Thread** содержит несколько методов для управления потоками.

- ✓ **getName()** - получить имя потока
- ✓ **getPriority()** - получить приоритет потока
- ✓ **isAlive()** - определить, выполняется ли поток
- ✓ **join()** - ожидать завершения потока
- ✓ **run()** - запуск потока. В нём пишете свой код
- ✓ **sleep()** - приостановить поток на заданное время

✓ **start()** - запустить поток

Получим информацию о главном потоке и поменяем его имя.

```
Thread mainThread = Thread.currentThread();
mInfoTextView.setText("Текущий поток: " +
mainThread.getName());
```

```
// Меняем имя и выводим в текстовом поле
mainThread.setName("CatThread");
mInfoTextView.append("\nНовое имя потока: " +
mainThread.getName());
```

Имя у главного потока по умолчанию **main**, которое мы заменили на **CatThread**.

Вызовем информацию о названии потока без указания метода.

```
Thread mainThread = Thread.currentThread();
mInfoTextView.setText("Текущий поток: " +
mainThread);
```

В этом случае можно увидеть строчку **Thread[main,5,main]** - имя потока, его приоритет и имя его группы.

### Создание собственного потока

Создать собственный поток не сложно. Достаточно наследоваться от класса **Thread**.

Объявим внутри нашего класса внутренний класс и вызовем его по щелчку, вызвав метод **start()**.

```
public class MyThread extends Thread {
    public void run() {
        Log.d(TAG, "Мой поток запущен...");
    }
}
```

```
public void onClick(View view) {
    MyThread myThread = new MyThread();
    myThread.start();
}
```

Как вариант, перенести вызов метода **start()** в конструктор.

```
public void onClick(View view) {
    MyThread myThread = new MyThread();
}
```

```
public class MyThread extends Thread {
```

```

// Конструктор
MyThread() {
    // Создаём новый поток
    super("Второй поток");
    Log.i(TAG, "Создан второй поток " + this);
    start(); // Запускаем поток
}

public void run() {
    Log.d(TAG, "Мой поток запущен...");

    try {
        for (int i = 5; i > 0; i--) {
            Log.i(TAG, "Второй поток: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        Log.i(TAG, "Второй поток прерван");
    }
}
}

```

### Создание потока с интерфейсом Runnable

Есть более сложный вариант создания потока. Для создания нового потока нужно реализовать интерфейс **Runnable**. Вы можете создать поток из любого объекта, реализующего интерфейс **Runnable** и объявить метод **run()**.

Внутри метода **run()** вы размещаете код для нового потока. Этот поток завершится, когда метод вернёт управление.

Когда вы объявите новый класс с интерфейсом **Runnable**, вам нужно использовать конструктор:

```
Thread(Runnable объект_потока, String имя_потока)
```

В первом параметре указывается экземпляр класса, реализующего интерфейс. Он определяет, где начнётся выполнение потока. Во втором параметре передаётся имя потока.

После создания нового потока, его нужно запустить с помощью метода **start()**, который, по сути, выполняет вызов метода **run()**.

Создадим новый поток внутри учебного проекта в виде вложенного класса и запустим его.

```

package ru.alexanderklimov.expresscourse;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

```

```

import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import static
ru.alexanderklimov.expresscourse.R.id.textViewInfo;

public class MainActivity extends AppCompatActivity {

    final String TAG = "ExpressCourse";

    private Button mButton;
    private EditText mResultEditText;
    private TextView mInfoTextView;

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mButton = (Button)
findViewById(R.id.buttonGetResult);
        mResultEditText = (EditText)
findViewById(R.id.editText);
        mInfoTextView = (TextView)
findViewById(textViewInfo);
    }

    public void onClick(View view) {
        new MyRunnable(); // создаём новый поток

        try {
            for (int i = 5; i > 0; i--) {
                Log.i(TAG, "Главный поток: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            Log.i(TAG, "Главный поток прерван");
        }
    }
}

```

```

class MyRunnable implements Runnable {
    Thread thread;

    // Конструктор
    MyRunnable() {
        // Создаём новый второй поток
        thread = new Thread(this, "Поток для
примера");
        Log.i(TAG, "Создан второй поток " +
thread);
        thread.start(); // Запускаем поток
    }

    // Обязательный метод для интерфейса Runnable
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                Log.i(TAG, "Второй поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            Log.i(TAG, "Второй поток прерван");
        }
    }
}

```

Внутри конструктора **MyRunnable()** мы создаём новый объект класса **Thread**

```
thread = new Thread(this, "Поток для примера");
```

В первом параметре использовался объект **this**, что означает желание вызвать метод **run()** этого объекта. Далее вызывается метод **start()**, в результате чего запускается выполнение потока, начиная с метода **run()**. В свою очередь метод запускает цикл для нашего потока. После вызова метода **start()**, конструктор **MyRunnable()** возвращает управление приложению. Когда главный поток продолжает свою работу, он входит в свой цикл. После этого оба потока выполняются параллельно.

Можно запускать несколько потоков, а не только второй поток в дополнение к первому. Это может привести к проблемам, когда два потока пытаются работать с одной переменной одновременно.

### Ключевое слово **synchronized** - синхронизированные методы

Для решения проблемы с потоками, которые могут внести путаницу, используется синхронизация.

Метод может иметь модификатор **synchronized**. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его в том же экземпляре, должны ожидать. Это позволяет исключить путаницу, когда несколько потоков пытаются вызвать метод.

```
synchronized void meow(String msg);
```

Кроме того, ключевое слово **synchronized** можно использовать в качестве оператора. Вы можете заключить в блок **synchronized** вызовы методов какого-нибудь класса:

```
synchronized(объект) {  
    // операторы, требующие синхронизации  
}
```

### Looper

Поток имеет в своём составе сущности **Looper**, **Handler**, **MessageQueue**.

Каждый поток имеет один уникальный **Looper** и может иметь много **Handler**.

Считайте **Looper** вспомогательным объектом потока, который управляет им. Он обрабатывает входящие сообщения, а также даёт указание потоку завершиться в нужный момент.

Поток получает свой **Looper** и **MessageQueue** через метод **Looper.prepare()** после запуска. **Looper.prepare()** идентифицирует вызывающий потк, создаёт **Looper** и **MessageQueue** и связывает поток с ними в хранилище **ThreadLocal**. Метод **Looper.loop()** следует вызывать для запуска **Looper**. Завершить его работу можно через метод **looper.quit()**.

```
class LooperThread extends Thread {  
    public Handler mHandler;  
  
    public void run() {  
        Looper.prepare();  
  
        mHandler = new Handler() {  
            public void handleMessage(Message msg) {  
                // process incoming messages here  
            }  
        };  
        Looper.loop();  
    }  
}
```

Используйте статический метод **getMainLooper()** для доступа к **Looper** главного потока:

```
Looper mainLooper = Looper.getMainLooper();
```

Создадим два потока. Один запустим в основном потоке, а второй отдельно от основного. Нам будет достаточно двух кнопок и метки.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android
"

xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:id="@+id/button_start"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="start()" />
    <Button
        android:id="@+id/button_run"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="run()" />
    <TextView
        android:id="@+id/textview_info"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Обратите внимание, как запускаются потоки. Первый поток запускается с помощью метода **start()**, а второй - **run()**. Затем проверяем, в каком потоке мы находимся.

```
package ru.alexanderklimov.as23;

import android.os.Bundle;
import android.os.Looper;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
```



```

public class MainActivity extends AppCompatActivity {

    TextView mInfoTextView;

    @Override
    protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button startButton = (Button)
findViewById(R.id.button_start);
        Button runButton = (Button)
findViewById(R.id.button_run);
        mInfoTextView = (TextView)
findViewById(R.id.textview_info);

        startButton.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Thread thread = new Thread(new
MyRunnable());
                thread.start(); //в фоновом потоке
            }
        });

        runButton.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Thread thread = new Thread(new
MyRunnable());
                thread.run(); //в текущем потоке
            }
        });
    }

    private class MyRunnable implements Runnable {

        @Override
        public void run() {
            // Проверяем, в каком потоке находимся

```

```

        if (Looper.getMainLooper().getThread() ==
Thread.currentThread()) {
            mInfoTextView.setText("В основном
потоке");
        } else {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    mInfoTextView.setText("В
фоновом потоке");
                }
            });
        }
    }
}

```

Эта тема достаточно сложная и для большинства не представляет интереса и необходимости изучать.

В Android потоки в чистом виде используются всё реже и реже, у системы есть собственные способы.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

## 7. Контрольные вопросы и задания

6. Что такое паток?
7. Как создать несколько потоков?
8. Как запустить потоки?
9. Как синхронизировать потоки?
10. Для чего необходимо синхронизировать потоки?

## **8. Рекомендуемая литература**

91. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
92. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
93. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
94. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
95. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
96. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ создания файлов»

Минск  
2018

## Лабораторная работа № 17

### Тема работы: «Разработка программ создания файлов»

#### 1. Цель работы

Закрепить навык создания, открытия и чтения файлов.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 16 реализовать запись всех данных в файлы, для каждого объекта создать свой файл (продумайте логическое название файлов и папок где они будут храниться). Каждый файл должен в себе хранить:

- ✓ имя объекта;
- ✓ название класс (названия всех классов);
- ✓ названия и содержимое всех полей.

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Система ввода/вывода

Java имеет в своём составе множество классов, связанных с вводом/выводом данных. Рассмотрим некоторые из них.

##### Класс File

В отличие от большинства классов ввода/вывода, класс **File** работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

##### Поток

При работе с данными ввода/вывода вам будет часто попадаться термин **Поток** (Stream). Поток - это абстрактное значение источника или приёмника данных, которые способны обрабатывать информацию. Вы в реальности не видите, как действительно идёт обработка данных в устройствах ввода/вывода, так как это сложно и вам это не нужно. Это как с телевизором -

вы не знаете, как сигнал из кабеля превращается в картинку на экране, но вполне можете переключаться между каналами через пульт.

Есть два типа потоков: байтовые и символьные. В некоторых ситуациях символьные потоки более эффективны, чем байтовые.

За ввод и вывод отвечают разные классы Java. Классы, производные от базовых классов **InputStream** или **Reader**, имеют методы с именами **read()** для чтения отдельных байтов или массива байтов (отвечают за ввод данных). Классы, производные от классов **OutputStream** или **Writer**, имеют методы с именами **write()** для записи одиночных байтов или массива байтов (отвечают за вывод данных).

## Класс OutputStream

Класс **OutputStream** - это абстрактный класс, определяющий потоковый байтовый вывод.

В этой категории находятся классы, определяющие, куда направляются ваши данные: в массив байтов (но не напрямую в String; предполагается что вы сможете создать их из массива байтов), в файл или канал.

### **BufferedOutputStream**

Буферизированный выходной поток

### **ByteArrayOutputStream**

Создает буфер в памяти. Все данные, посылаемые в этот поток, размещаются в созданном буфере

### **DataOutputStream**

Выходной поток, включающий методы для записи стандартных типов данных Java

### **FileOutputStream**

Отправка данных в файл на диске. Реализация класса OutputStream

### **ObjectOutputStream**

Выходной поток для объектов

### **PipedOutputStream**

Реализует понятие выходного канала.

### **FilterOutputStream**

Абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства.

Методы класса:

- ✓ **int close()** - закрывает выходной поток. Следующие попытки записи передадут исключение **IOException**
- ✓ **void flush()** - финализирует выходное состояние, очищая все буферы вывода
- ✓ **abstract void write (int oneByte)** - записывает единственный байт в выходной поток
- ✓ **void write (byte[] buffer)** - записывает полный массив байтов в выходной поток
- ✓ **void write (byte[] buffer, int offset, int count)** - записывает диапазон из *count* байт из массива, начиная с смещения *offset*

## BufferedOutputStream

Класс **BufferedOutputStream** не сильно отличается от класса **OutputStream**, за исключением дополнительного метода **flush()**, используемого для обеспечения записи данных в буферизируемый поток. Буферы вывода нужны для повышения производительности.

### **ByteArrayOutputStream**

Класс **ByteArrayOutputStream** использует байтовый массив в выходном потоке. Метод **close()** можно не вызывать.

### **DataOutputStream**

Класс **DataOutputStream** позволяет писать элементарные данные в поток через интерфейс **DataOutput**, который определяет методы, преобразующие элементарные значения в форму последовательности байтов. Такие потоки облегчают сохранение в файле двоичных данных.

Класс **DataOutputStream** расширяет класс **FilterOutputStream**, который в свою очередь, расширяет класс **OutputStream**.

Методы интерфейса **DataOutput**:

- ✓ **writeDouble(double value)**
- ✓ **writeBoolean(boolean value)**
- ✓ **writeInt(int value)**

### **FileOutputStream**

Класс **FileOutputStream** создаёт объект класса **OutputStream**, который можно использовать для записи байтов в файл. Создание нового объекта не зависит от того, существует ли заданный файл, так как он создаёт его перед открытием. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

### **Классы символьных потоков**

Символьные потоки имеют два основных абстрактных класса **Reader** и **Writer**, управляющие потоками символов Unicode.

### **Reader**

Методы класса **Reader**:

- ✓ **abstract void close()** - закрывает входной поток. Последующие попытки чтения передадут исключение **IOException**
- ✓ **void mark(int readLimit)** - помещает метку в текущую позицию во входном потоке
- ✓ **boolean markSupported()** - возвращает *true*, если поток поддерживает методы **mark()** и **reset()**
- ✓ **int read()** - возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает значение -1. Есть и другие перегруженные версии метода

- ✓ `boolean ready()` - возвращает значение *true*, если следующий запрос не будет ожидать.
- ✓ `void reset()` - сбрасывает указатель ввода в ранее установленную позицию метки
- ✓ `long skip(long charCount)` - пропускает указанное число символов ввода, возвращая количество действительно пропущенных символов

## **BufferedReader**

Буферизированный входной символьный поток

### **CharArrayReader**

Входной поток, который читает из символьного массива

### **FileReader**

Входной поток, читающий файл

### **FilterReader**

Фильтрующий читатель

### **InputStreamReader**

Входной поток, транслирующий байты в символы

### **LineNumberReader**

Входной поток, подсчитывающий строки

### **PipedReader**

Входной канал

### **PushbackReader**

Входной поток, позволяющий возвращать символы обратно в поток

### **Reader**

Абстрактный класс, описывающий символьный ввод

### **StringReader**

Входной поток, читающий из строки

## **Класс BufferedReader**

Класс **BufferedReader** увеличивает производительность за счёт буферизации ввода.

## **Класс CharArrayReader**

Класс **CharArrayReader** использует символьный массив в качестве источника.

## **Класс FileReader**

Класс **FileReader**, производный от класса **Reader**, можно использовать для чтения содержимого файла. В конструкторе класса нужно указать либо путь к файлу, либо объект типа **File**.

## **Writer**

Класс **Writer** - абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение **IOException**.

Методы класса:

- ✓ `Writer append(char c)` - добавляет символ в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- ✓ `Writer append(CharSequence csq)` - добавляет символы в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток



- ✓ **Writer append(CharSequence csq, int start, int end)** - добавляет диапазон символов в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- ✓ **abstract void close()** - закрывает вызывающий поток
- ✓ **abstract void flush()** - финализирует выходное состояние так, что все буферы очищаются
- ✓ **void write(int oneChar)** - записывает единственный символ в вызывающий выходной поток. Есть и другие перегруженные версии метода

#### **BufferedWriter**

Буферизированный выходной символьный поток

#### **CharArrayWriter**

Выходной поток, который пишет в символьный массив

#### **FileWriter**

Выходной поток, пишущий в файл

#### **FilterWriter**

Фильтрующий писатель

#### **OutputStreamWriter**

Выходной поток, транслирующий байты в символы

#### **PipedWriter**

Выходной канал

#### **PrintWriter**

Выходной поток, включающий методы print() и println()

#### **StringWriter**

Выходной поток, пишущий в строку

#### **Writer**

Абстрактный класс, описывающий символьный вывод

### **Класс BufferedWriter**

Класс **BufferedWriter** - это класс, производный от класса **Writer**, который буферизует вывод. С его помощью можно повысить производительность за счёт снижения количества операций физической записи в выходное устройство.

### **Класс CharArrayWriter**

Класс **CharArrayWriter** использует массив для выходного потока.

### **Класс FileWriter**

Класс **FileWriter** создаёт объект класса, производного от класса **Writer**, который вы можете применять для записи файла. Есть конструкторы, которые позволяют добавить вывод в конец файла. Создание объекта не зависит от наличия файла, он будет создан в случае необходимости. Если файл существует и он доступен только для чтения, то передаётся исключение **IOException**.

## **Чтение и запись файлов**

Существует множество классов и методов для чтения и записи файлов. Наиболее распространённые из них - классы **FileInputStream** и **FileOutputStream**, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, нужно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора.

```
        FileInputStream(String      filename)      throws
FileNotFoundException
        FileOutputStream(String      filename)      throws
FileNotFoundException
```

В **filename** нужно указать имя файла, который вы хотите открыть. Если при создании входного потока файл не существует, передаётся исключение **FileNotFoundException**. Аналогично для выходных потоков, если файл не может быть открыт или создан, также передаётся исключение. Сам класс исключения происходит от класса **IOException**. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

После завершения работы с файлом, его необходимо закрыть с помощью метода **close()** для освобождения системных ресурсов. Незакрытый файл приводит к утечке памяти.

В JDK 7 метод **close()** определяется интерфейсом **AutoCloseable** и можно явно не закрывать файл, а использовать новый оператор **try-c-ресурсами**, что для Android пока не слишком актуально.

Чтобы читать файл, нужно вызвать метод **read()**. Когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. Когда будет достигнут конец файла, то метод вернёт значение -1. Примеры использования методов есть в различных статьях на сайте.

Иногда используют вариант, когда метод **close()** помещается в блок **finally**. При таком подходе все методы, которые получают доступ к файлу, содержатся в пределах блока **try**, а блок **finally** используется для закрытия файла. Таким образом, независимо от того, как закончится блок **try**, файл будет закрыт.

Так как исключение **FileNotFoundException** является подклассом **IOException**, то не обязательно обрабатывать два исключения отдельно, а оставить только **IOException**, если вам не нужно отдельно обрабатывать разные причины неудачного открытия файла. Например, если пользователь вводит вручную имя файла, то более конкретное исключение будет к месту.

Для записи в файл используется метод **write()**.

```
void write(int value) throws IOException
```

Метод пишет в файл байт, переданный параметром **value**. Хотя параметр объявлена как целочисленный, в файл записываются только младшие восемь бит. При ошибке записи передаётся исключение.

В JDK 7 есть способ автоматического управления ресурсами:

```
try (спецификация_ресурса) {
    // использование ресурса
}
```

## Буферизированное чтение из файла - **BufferedReader**

Чтобы открыть файл для посимвольного чтения, используется класс **FileInputReader**; имя файла задаётся в виде строки (String) или объекта **File**. Ускорить процесс чтения помогает буферизация ввода, для этого полученная ссылка передаётся в конструктор класса **BufferedReader**. Так как в интерфейсе класса имеется метод **readLine()**, все необходимое для чтения имеется в вашем распоряжении. При достижении конца файла метод **readLine()** возвращает ссылку **null**.

### Вывод в файл - **FileWriter**

Объект **FileWriter** записывает данные в файл. При вводе/выводе практически всегда применяется буферизация, поэтому используется **BufferedWriter**.

Когда данные входного потока исчерпываются, метод **readLine()** возвращает **null**. Для потока явно вызывается метод **close()**; если не вызвать его для всех выходных файловых потоков, в буферах могут остаться данные, и файл получится неполным.

### Сохранение и восстановление данных - **PrintWriter**

**PrintWriter** форматирует данные так, чтобы их мог прочитать человек. Однако для вывода информации, предназначенной для другого потока, следует использовать классы **DataOutputStream** для записи данных и **DataInputStream** для чтения данных.

Единственным надёжным способом записать в поток **DataOutputStream** строку так, чтобы ее можно было потом правильно считать потоком **DataInputStream**, является кодирование UTF-8, реализуемое методами **readUTF()** и **writeUTF()**. Эти методы позволяют смешивать строки и другие типы данных, записываемые потоком **DataOutputStream**, так как вы знаете, что строки будут правильно сохранены в Юникоде и их будет просто воспроизвести потоком **DataInputStream**.

Метод **writeDouble()** записывает число **double** в поток, а соответствующий ему метод **readDouble()** затем восстанавливает его (для других типов также существуют подобные методы).

### **RandomAccessFile** - Чтение/запись файлов с произвольным доступом

Работа с классом **RandomAccessFile** напоминает использование совмещенных в одном классе потоков **DataInputStream** и **DataOutputStream** (они реализуют те же интерфейсы **DataInput** и **DataOutput**). Кроме того, метод **seek()** позволяет переместиться к определенной позиции и изменить хранящееся там значение.

При использовании **RandomAccessFile** необходимо знать структуру файла. Класс **RandomAccessFile** содержит методы для чтения и записи примитивов и строк UTF-8.

**RandomAccessFile** может открываться в режиме чтения ("r") или чтения/записи ("rw"). Также есть режим "rws", когда файл открывается для

операций чтения-записи и каждое изменение данных файла немедленно записывается на физическое устройство.

### Исключения ввода/вывода

В большинстве случаев у классов ввода/вывода используется исключение **IOException**. Второе исключение **FileNotFoundException** передаётся в тех случаях, когда файл не может быть открыт. Данное исключение происходит от **IOException**, поэтому оба исключения можно обрабатывать в одном блоке **catch**, если у вас нет нужды обрабатывать их по отдельности.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

## 7. Контрольные вопросы и задания

1. Как создать файл?
2. Как открыть файл для чтения?
3. Как открыть файл для записи?
4. Расскажите о исключениях ввода/вывода.

## **8. Рекомендуемая литература**

97. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с
98. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
99. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
100. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
101. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. – 240с.
102. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»  
Филиал  
«Минский радиотехнический колледж»

Учебная дисциплина «Разработка приложений для мобильных устройств»

**Инструкция**  
по выполнению лабораторной работы  
«Разработка программ обработки файлов»

Минск  
2018

## Лабораторная работа № 18

### Тема работы: «Разработка программ обработки файлов»

#### 1. Цель работы

Отработать навык решения задач с использованием различных видов файлов.

#### 2. Задание

Номер варианта соответствует вашему номеру по списку.

Необходимо в задании 1 лабораторной работы № 17 реализовать чтение данных из файлов при:

- ✓ созданию объектов, конкретных классов;
- ✓ обработке определённой информации в методах.

#### 3. Оснащение работы

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

#### 4. Основные теоретические сведения

##### Система ввода/вывода

Java имеет в своём составе множество классов, связанных с вводом/выводом данных. Рассмотрим некоторые из них.

##### Класс File

В отличие от большинства классов ввода/вывода, класс **File** работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

##### Поток

При работе с данными ввода/вывода вам будет часто попадаться термин **Поток** (Stream). Поток - это абстрактное значение источника или приёмника данных, которые способны обрабатывать информацию. Вы в реальности не видите, как действительно идёт обработка данных в устройствах ввода/вывода, так как это сложно и вам это не нужно. Это как с телевизором - вы не знаете, как сигнал из кабеля превращается в картинку на экране, но вполне можете переключаться между каналами через пульт.

Есть два типа потоков: байтовые и символьные. В некоторых ситуациях символьные потоки более эффективны, чем байтовые.

За ввод и вывод отвечают разные классы Java. Классы, производные от базовых классов **InputStream** или **Reader**, имеют методы с именами **read()** для чтения отдельных байтов или массива байтов (отвечают за ввод данных). Классы, производные от классов **OutputStream** или **Writer**, имеют методы с именами **write()** для записи одиночных байтов или массива байтов (отвечают за вывод данных).

## Класс OutputStream

Класс **OutputStream** - это абстрактный класс, определяющий потоковый байтовый вывод.

В этой категории находятся классы, определяющие, куда направляются ваши данные: в массив байтов (но не напрямую в String; предполагается что вы сможете создать их из массива байтов), в файл или канал.

### **BufferedOutputStream**

Буферизированный выходной поток

### **ByteArrayOutputStream**

Создает буфер в памяти. Все данные, посылаемые в этот поток, размещаются в созданном буфере

### **DataOutputStream**

Выходной поток, включающий методы для записи стандартных типов данных Java

### **FileOutputStream**

Отправка данных в файл на диске. Реализация класса OutputStream

### **ObjectOutputStream**

Выходной поток для объектов

### **PipedOutputStream**

Реализует понятие выходного канала.

### **FilterOutputStream**

Абстрактный класс, предоставляющий интерфейс для классов-надстроек, которые добавляют к существующим потокам полезные свойства.

Методы класса:

- ✓ **int close()** - закрывает выходной поток. Следующие попытки записи передадут исключение **IOException**
- ✓ **void flush()** - финализирует выходное состояние, очищая все буферы вывода
- ✓ **abstract void write (int oneByte)** - записывает единственный байт в выходной поток
- ✓ **void write (byte[] buffer)** - записывает полный массив байтов в выходной поток
- ✓ **void write (byte[] buffer, int offset, int count)** - записывает диапазон из *count* байт из массива, начиная с смещения *offset*

## BufferedOutputStream

Класс **BufferedOutputStream** не сильно отличается от класса **OutputStream**, за исключением дополнительного метода **flush()**, используемого для обеспечения записи данных в буферизируемый поток. Буферы вывода нужны для повышения производительности.



## ByteArrayOutputStream

Класс **ByteArrayOutputStream** использует байтовый массив в выходном потоке. Метод **close()** можно не вызывать.

## DataOutputStream

Класс **DataOutputStream** позволяет писать элементарные данные в поток через интерфейс **DataOutput**, который определяет методы, преобразующие элементарные значения в форму последовательности байтов. Такие потоки облегчают сохранение в файле двоичных данных.

Класс **DataOutputStream** расширяет класс **FilterOutputStream**, который в свою очередь, расширяет класс **OutputStream**.

Методы интерфейса **DataOutput**:

- ✓ **writeDouble(double value)**
- ✓ **writeBoolean(boolean value)**
- ✓ **writeInt(int value)**

## FileOutputStream

Класс **FileOutputStream** создаёт объект класса **OutputStream**, который можно использовать для записи байтов в файл. Создание нового объекта не зависит от того, существует ли заданный файл, так как он создаёт его перед открытием. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

## Классы символьных потоков

Символьные потоки имеют два основных абстрактных класса **Reader** и **Writer**, управляющие потоками символов Unicode.

## Reader

Методы класса **Reader**:

- ✓ **abstract void close()** - закрывает входной поток. Последующие попытки чтения передадут исключение **IOException**
- ✓ **void mark(int readLimit)** - помещает метку в текущую позицию во входном потоке
- ✓ **boolean markSupported()** - возвращает *true*, если поток поддерживает методы **mark()** и **reset()**
- ✓ **int read()** - возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает значение -1. Есть и другие перегруженные версии метода
- ✓ **boolean ready()** - возвращает значение *true*, если следующий запрос не будет ожидать.
- ✓ **void reset()** - сбрасывает указатель ввода в ранее установленную позицию метки

- ✓ `logn skip(long charCount)` - пропускает указанное число символов ввода, возвращая количество действительно пропущенных символов

### **BufferedReader**

Буферизированный входной символьный поток

### **CharArrayReader**

Входной поток, который читает из символьного массива

### **FileReader**

Входной поток, читающий файл

### **FilterReader**

Фильтрующий читатель

### **InputStreamReader**

Входной поток, транслирующий байты в символы

### **LineNumberReader**

Входной поток, подсчитывающий строки

### **PipedReader**

Входной канал

### **PushbackReader**

Входной поток, позволяющий возвращать символы обратно в поток

### **Reader**

Абстрактный класс, описывающий символьный ввод

### **StringReader**

Входной поток, читающий из строки

### **Класс BufferedReader**

Класс **BufferedReader** увеличивает производительность за счёт буферизации ввода.

### **Класс CharArrayReader**

Класс **CharArrayReader** использует символьный массив в качестве источника.

### **Класс FileReader**

Класс **FileReader**, производный от класса **Reader**, можно использовать для чтения содержимого файла. В конструкторе класса нужно указать либо путь к файлу, либо объект типа **File**.

## **Writer**

Класс **Writer** - абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение **IOException**.

Методы класса:

- ✓ `Writer append(char c)` - добавляет символ в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- ✓ `Writer append(CharSequence csq)` - добавляет символы в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- ✓ `Writer append(CharSequence csq, int start, int end)` - добавляет диапазон символов в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток
- ✓ `abstract void close()` - закрывает вызывающий поток
- ✓ `abstract void flush()` - финализирует выходное состояние так, что все буферы очищаются

- ✓ `void write(int oneChar)` - записывает единственный символ в вызывающий выходной поток. Есть и другие перегруженные версии метода

#### **BufferedWriter**

Буферизированный выходной символьный поток

#### **CharArrayWriter**

Выходной поток, который пишет в символьный массив

#### **FileWriter**

Выходной поток, пишущий в файл

#### **FilterWriter**

Фильтрующий писатель

#### **OutputStreamWriter**

Выходной поток, транслирующий байты в символы

#### **PipedWriter**

Выходной канал

#### **PrintWriter**

Выходной поток, включающий методы `print()` и `println()`

#### **StringWriter**

Выходной поток, пишущий в строку

#### **Writer**

Абстрактный класс, описывающий символьный вывод

### **Класс BufferedWriter**

Класс **BufferedWriter** - это класс, производный от класса **Writer**, который буферизует вывод. С его помощью можно повысить производительность за счёт снижения количества операций физической записи в выходное устройство.

### **Класс CharArrayWriter**

Класс **CharArrayWriter** использует массив для выходного потока.

### **Класс FileWriter**

Класс **FileWriter** создаёт объект класса, производного от класса **Writer**, который вы можете применять для записи файла. Есть конструкторы, которые позволяют добавить вывод в конец файла. Создание объекта не зависит от наличия файла, он будет создан в случае необходимости. Если файл существует и он доступен только для чтения, то передаётся исключение **IOException**.

## **Чтение и запись файлов**

Существует множество классов и методов для чтения и записи файлов. Наиболее распространённые из них - классы **FileInputStream** и **FileOutputStream**, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, нужно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора.

```
FileInputStream(String filename) throws
FileNotFoundException
FileOutputStream(String filename) throws
FileNotFoundException
```

В **filename** нужно указать имя файла, который вы хотите открыть. Если при создании входного потока файл не существует, передаётся

исключение **FileNotFoundException**. Аналогично для выходных потоков, если файл не может быть открыт или создан, также передаётся исключение. Сам класс исключения происходит от класса **IOException**. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

После завершения работы с файлом, его необходимо закрыть с помощью метода **close()** для освобождения системных ресурсов. Незакрытый файл приводит к утечке памяти.

В JDK 7 метод **close()** определяется интерфейсом **AutoCloseable** и можно явно не закрывать файл, а использовать новый оператор **try-c-ресурсами**, что для Android пока не слишком актуально.

Чтобы читать файл, нужно вызвать метод **read()**. Когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. Когда будет достигнут конец файла, то метод вернёт значение -1. Примеры использования методов есть в различных статьях на сайте.

Иногда используют вариант, когда метод **close()** помещается в блок **finally**. При таком подходе все методы, которые получают доступ к файлу, содержатся в пределах блока **try**, а блок **finally** используется для закрытия файла. Таким образом, независимо от того, как закончится блок **try**, файл будет закрыт.

Так как исключение **FileNotFoundException** является подклассом **IOException**, то не обязательно обрабатывать два исключения отдельно, а оставить только **IOException**, если вам не нужно отдельно обрабатывать разные причины неудачного открытия файла. Например, если пользователь вводит вручную имя файла, то более конкретное исключение будет к месту.

Для записи в файл используется метод **write()**.

```
void write(int value) throws IOException
```

Метод пишет в файл байт, переданный параметром **value**. Хотя параметр объявлена как целочисленный, в файл записываются только младшие восемь бит. При ошибке записи передаётся исключение.

В JDK 7 есть способ автоматического управления ресурсами:

```
try (спецификация_ресурса) {  
    // использование ресурса  
}
```

### Буферизированное чтение из файла - **BufferedReader**

Чтобы открыть файл для посимвольного чтения, используется класс **FileInputStream**; имя файла задаётся в виде строки (String) или объекта **File**. Ускорить процесс чтения помогает буферизация ввода, для этого полученная ссылка передаётся в конструктор класса **BufferedReader**. Так как в интерфейсе класса имеется метод **readLine()**, все необходимое для чтения

имеется в вашем распоряжении. При достижении конца файла метод **readLine()** возвращает ссылку **null**.

### Вывод в файл - **FileWriter**

Объект **FileWriter** записывает данные в файл. При вводе/выводе практически всегда применяется буферизация, поэтому используется **BufferedWriter**.

Когда данные входного потока исчерпываются, метод **readLine()** возвращает **null**. Для потока явно вызывается метод **close()**; если не вызвать его для всех выходных файловых потоков, в буферах могут остаться данные, и файл получится неполным.

### Сохранение и восстановление данных - **PrintWriter**

**PrintWriter** форматирует данные так, чтобы их мог прочитать человек. Однако для вывода информации, предназначенной для другого потока, следует использовать классы **DataOutputStream** для записи данных и **DataInputStream** для чтения данных.

Единственным надежным способом записать в поток **DataOutputStream** строку так, чтобы ее можно было потом правильно считать потоком **DataInputStream**, является кодирование UTF-8, реализуемое методами **readUTF()** и **writeUTF()**. Эти методы позволяют смешивать строки и другие типы данных, записываемые потоком **DataOutputStream**, так как вы знаете, что строки будут правильно сохранены в Юникоде и их будет просто воспроизвести потоком **DataInputStream**.

Метод **writeDouble()** записывает число **double** в поток, а соответствующий ему метод **readDouble()** затем восстанавливает его (для других типов также существуют подобные методы).

### **RandomAccessFile** - Чтение/запись файлов с произвольным доступом

Работа с классом **RandomAccessFile** напоминает использование совмещенных в одном классе потоков **DataInputStream** и **DataOutputStream** (они реализуют те же интерфейсы **DataInput** и **DataOutput**). Кроме того, метод **seek()** позволяет переместиться к определенной позиции и изменить хранящееся там значение.

При использовании **RandomAccessFile** необходимо знать структуру файла. Класс **RandomAccessFile** содержит методы для чтения и записи примитивов и строк UTF-8.

**RandomAccessFile** может открываться в режиме чтения ("r") или чтения/записи ("rw"). Также есть режим "rws", когда файл открывается для операций чтения-записи и каждое изменение данных файла немедленно записывается на физическое устройство.

### Исключения ввода/вывода

В большинстве случаев у классов ввода/вывода используется исключение **IOException**. Второе исключение **FileNotFoundException** передаётся в тех случаях, когда файл не может быть открыт. Данное исключение происходит от **IOException**, поэтому оба исключения можно обрабатывать в одном блоке **catch**, если у вас нет нужды обрабатывать их по отдельности.

## 5. Порядок выполнения работы

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.

## 6. Форма отчета о работе

Лабораторная работа № \_\_\_\_

Номер учебной группы \_\_\_\_\_

Фамилия, инициалы учащегося \_\_\_\_\_

Дата выполнения работы \_\_\_\_\_

Тема работы: \_\_\_\_\_

Цель работы: \_\_\_\_\_

Оснащение работы: \_\_\_\_\_

Результат выполнения работы: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## 7. Контрольные вопросы и задания

5. Как создать файл?
6. Как открыть файл для чтения?
7. Как открыть файл для записи?
8. Расскажите о исключениях ввода/вывода.

## 8. Рекомендуемая литература

103. Android для разработчиков/ П. Дейтел [и др.]; под общ. ред. П. Дейтел. – СПб.: Питер, 2015. – 384с

104. Farrell J. Java Programming – Course Technology, 2017. – 1026 с.
105. Head First. Программирование для Android/ Д. Гриффитс [и др.]; под общ. ред. Д. Гриффитс. – СПб.: Питер, 2016. – 704с
106. Лафоре Р. Структуры данных и алгоритмы наJava – СПб: Питер, 2013. – 704 с.
107. Сеттер Р. Изучаем Java на примерах и задачах – М: Наука и техника, 2016. –240с.
108. Хорстманн К. Java. Библиотека профессионала / Хорстманн К., Корнелл Г. - М.: Вильямс, 2016. – 866 с.

