|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА − Российский технологический университет»**  **РТУ МИРЭА** |

**Институт информационных технологий (ИИТ)**

**Кафедра инструментального и прикладного программного обеспечения (ИиППО)**

**ОТЧЁТ ПО ПРАКТИЧЕСКИМ РАБОТАМ №1-32**

**по дисциплине «Программирование на языке Джава»**

|  |  |  |  |
| --- | --- | --- | --- |
| Выполнил:  Студент группы Шмелев Д.В. | « » 2021г. | (подпись) | Шмелев Д.В. |
|  |  |  |  |
| Принял:  Ассистент кафедры ИиППО ИИТ | « » 2021г. | (подпись) | Волков М.Ю. |

Москва 2021 г.

Оглавление

[ПРАКТИЧЕСКАЯ РАБОТА №1. КЛАССЫ, КАК НОВЫЕ ТИПЫ ДАННЫХ. ПОЛЯ ДАННЫХ И МЕТОДЫ 8](#_Toc90615153)

[Цель работы 8](#_Toc90615154)

[Теоретическое введение 8](#_Toc90615155)

[Постановка задачи 9](#_Toc90615156)

[Программный код 10](#_Toc90615157)

[Вывод программы 11](#_Toc90615158)

[Вывод 11](#_Toc90615159)

[ПРАКТИЧЕСКАЯ РАБОТА №2. ИСПОЛЬЗОВАНИЕ UML ДИАГРАММ В ОБЪЕКТНООРИЕНТИРОВАННОМ ПРОГРАММИРОВАНИИ 12](#_Toc90615160)

[Цель работы 12](#_Toc90615161)

[Теоретическое введение 12](#_Toc90615162)

[Постановка задачи 13](#_Toc90615163)

[Программный код 14](#_Toc90615164)

[Вывод программы 15](#_Toc90615165)

[Вывод 15](#_Toc90615166)

[ПРАКТИЧЕСКАЯ РАБОТА № 3. НАСЛЕДОВАНИЕ. АБСТРАКТНЫЕ СУПЕРКЛАССЫ И ИХ ПОДКЛАССЫ В JAVA. 16](#_Toc90615167)

[Цель работы 16](#_Toc90615168)

[Теоретическое введение 16](#_Toc90615169)

[Постановка задачи 17](#_Toc90615170)

[Программный код 18](#_Toc90615171)

[Вывод программы 20](#_Toc90615172)

[Вывод 20](#_Toc90615173)

[ПРАКТИЧЕСКАЯ РАБОТА № 4. СОЗДАНИЕ GUI. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ В JAVA. 21](#_Toc90615174)

[Цель работы 21](#_Toc90615175)

[Теоретическое введение 21](#_Toc90615176)

[Постановка задачи 23](#_Toc90615177)

[Программный код 24](#_Toc90615178)

[Вывод программы 26](#_Toc90615179)

[Вывод 27](#_Toc90615180)

[ПРАКТИЧЕСКАЯ РАБОТА № 5. ПРОГРАММИРОВАНИЕ РЕКУРСИИ В JAVA 28](#_Toc90615181)

[Цель работы 28](#_Toc90615182)

[Теоретическое введение 28](#_Toc90615183)

[Постановка задачи 29](#_Toc90615184)

[Программный код 30](#_Toc90615185)

[Вывод программы 35](#_Toc90615186)

[Вывод 36](#_Toc90615187)

[ПРАКТИЧЕСКАЯ РАБОТА №6. ТЕХНИКИ СОРТИРОВКИ В JAVA 37](#_Toc90615188)

[Цель работы 37](#_Toc90615189)

[Теоретическое введение 37](#_Toc90615190)

[Постановка задачи 39](#_Toc90615191)

[Программный код 39](#_Toc90615192)

[Вывод программы 46](#_Toc90615193)

[Вывод 47](#_Toc90615194)

[ПРАКТИЧЕСКАЯ РАБОТА № 7. ИСПОЛЬЗОВАНИЕ СТАНДАРТНЫХ КОНТЕЙНЕРНЫХ КЛАССОВ ПРИ ПРОГРАММИРОВАНИИ НА JAVA 48](#_Toc90615195)

[Цель работы 48](#_Toc90615196)

[Теоретическое введение 48](#_Toc90615197)

[Постановка задачи 50](#_Toc90615198)

[Программный код 52](#_Toc90615199)

[Вывод программы 56](#_Toc90615200)

[Вывод 56](#_Toc90615201)

[ПРАКТИЧЕСКАЯ РАБОТА № 8 РАЗЛИЧНЫЕ ВИДЫ СПИСКОВ ОЖИДАНИЯ 57](#_Toc90615202)

[Постановка задачи 57](#_Toc90615203)

[Программный код 58](#_Toc90615204)

[Вывод программы 60](#_Toc90615205)

[Вывод 60](#_Toc90615206)

[ПРАКТИЧЕСКАЯ РАБОТА №9 СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ИСКЛЮЧЕНИЙ 61](#_Toc90615207)

[Цель работы 61](#_Toc90615208)

[Теоретическое введение 61](#_Toc90615209)

[Постановка задачи 64](#_Toc90615210)

[Программный код 66](#_Toc90615211)

[Вывод программы 69](#_Toc90615212)

[Вывод 69](#_Toc90615213)

[ПРАКТИЧЕСКАЯ РАБОТА № 10 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ. ПОРОЖДАЮЩИЕ ПАТТЕРНЫ: АБСТРАКТНАЯ ФАБРИКА, ФАБРИЧНЫЙ МЕТОД 70](#_Toc90615214)

[Цель работы 70](#_Toc90615215)

[Теоретическое введение 70](#_Toc90615216)

[Постановка задачи 70](#_Toc90615217)

[Программный код 72](#_Toc90615218)

[Вывод программы 76](#_Toc90615219)

[Вывод 76](#_Toc90615220)

[ПРАКТИЧЕСКАЯ РАБОТА №11, 12 ОБРАБОТКА СТРОК. 77](#_Toc90615221)

[Цель работы 77](#_Toc90615222)

[Теоретическое введение 77](#_Toc90615223)

[Постановка задачи 83](#_Toc90615224)

[Программный код 85](#_Toc90615225)

[Вывод программы 90](#_Toc90615226)

[Вывод 91](#_Toc90615227)

[ПРАКТИЧЕСКАЯ РАБОТА № 13, 14 РАБОТА С ДАТОЙ И ВРЕМЕНЕМ 92](#_Toc90615228)

[Цель работы 92](#_Toc90615229)

[Теоретическое введение 92](#_Toc90615230)

[Постановка задачи 95](#_Toc90615231)

[Программный код 96](#_Toc90615232)

[Вывод программы 102](#_Toc90615233)

[Вывод 103](#_Toc90615234)

[ПРАКТИЧЕСКАЯ РАБОТА № 15, 16 ИТОГОВАЯ РАБОТА 104](#_Toc90615235)

[Цель работы 104](#_Toc90615236)

[Теоретическое введение 104](#_Toc90615237)

[Постановка задачи 104](#_Toc90615238)

[Программный код 104](#_Toc90615239)

[Вывод программы 104](#_Toc90615240)

[Вывод 104](#_Toc90615241)

[ЛАБОРАТОРНАЯ РАБОТА №1 ЦИКЛЫ, УСЛОВИЯ, ПЕРЕМЕННЫЕ И МАССИВЫ В JAVA. 105](#_Toc90615242)

[Цель работы 105](#_Toc90615243)

[Постановка задачи 105](#_Toc90615244)

[Программный код 106](#_Toc90615245)

[Вывод программы 108](#_Toc90615246)

[Вывод 108](#_Toc90615247)

[ЛАБОРАТОРНАЯ РАБОТА № 2 ООП В JAVA. ПОНЯТИЕ КЛАССА. 109](#_Toc90615248)

[Цель работы 109](#_Toc90615249)

[Постановка задачи 109](#_Toc90615250)

[Программный код 110](#_Toc90615251)

[Вывод программы 111](#_Toc90615252)

[Вывод 111](#_Toc90615253)

[ЛАБОРАТОРНАЯ РАБОТА № 3 НАСЛЕДОВАНИЕ В JAVA 112](#_Toc90615254)

[Цель работы 112](#_Toc90615255)

[Постановка задачи 112](#_Toc90615256)

[Программный код 113](#_Toc90615257)

[Вывод программы 114](#_Toc90615258)

[Вывод 114](#_Toc90615259)

[ЛАБОРАТОРНАЯ РАБОТА № 4 ИНТЕРФЕЙСЫ В JAVA. 115](#_Toc90615260)

[Цель работы 115](#_Toc90615261)

[Постановка задачи 115](#_Toc90615262)

[Программный код 115](#_Toc90615263)

[Вывод программы 116](#_Toc90615264)

[Вывод 116](#_Toc90615265)

[ЛАБОРАТОРНАЯ РАБОТА № 5 СОЗДАНИЕ ПРОГРАМ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ НА JAVA 117](#_Toc90615266)

[Цель работы 117](#_Toc90615267)

[Постановка задачи 117](#_Toc90615268)

[Программный код 117](#_Toc90615269)

[Вывод программы 118](#_Toc90615270)

[Вывод 119](#_Toc90615271)

[ЛАБОРАТОРНАЯ РАБОТА № 6 ОБРАБОТКА СОБЫТИЙ В JAVA ПРОГРАММАХ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ 120](#_Toc90615272)

[Цель работы 120](#_Toc90615273)

[Теоретическое введение 120](#_Toc90615274)

[Постановка задачи 125](#_Toc90615275)

[Программный код 126](#_Toc90615276)

[Вывод программы 128](#_Toc90615277)

[Вывод 130](#_Toc90615278)

[ЛАБОРАТОРНАЯ РАБОТА № 7 КОЛЛЕКЦИИ, ОЧЕРЕДИ, СПИСКИ В JAVA 131](#_Toc90615279)

[Цель работы 131](#_Toc90615280)

[Постановка задачи 131](#_Toc90615281)

[Программный код 132](#_Toc90615282)

[Вывод программы 133](#_Toc90615283)

[Вывод 133](#_Toc90615284)

[ЛАБОРАТОРНАЯ РАБОТА №8 РАБОТА С ФАЙЛАМИ 134](#_Toc90615285)

[Цель работы 134](#_Toc90615286)

[Теоретическое введение 134](#_Toc90615287)

[Постановка задачи 134](#_Toc90615288)

[Программный код 134](#_Toc90615289)

[Вывод программы 135](#_Toc90615290)

[Вывод 135](#_Toc90615291)

[ЛАБОРАТОРНАЯ РАБОТА № 9 РАБОТА С ИСКЛЮЧЕНИЯМИ 136](#_Toc90615292)

[Цель работы 136](#_Toc90615293)

[Теоретическое введение 136](#_Toc90615294)

[Постановка задачи 138](#_Toc90615295)

[Программный код 145](#_Toc90615296)

[Вывод программы 147](#_Toc90615297)

[Вывод 149](#_Toc90615298)

[ЛАБОРАТОРНАЯ РАБОТА № 10 РАБОТА С ДЖЕНЕРИКАМИ. СТИРАНИЕ ТИПОВ 151](#_Toc90615299)

[Цель работы 151](#_Toc90615300)

[Теоретические сведения 151](#_Toc90615301)

[Постановка задачи 155](#_Toc90615302)

[Программный код 156](#_Toc90615303)

[Вывод программы 158](#_Toc90615304)

[Вывод 158](#_Toc90615305)

[ЛАБОРАТОРНАЯ РАБОТА №11 ДЖЕНЕРИК MAP 159](#_Toc90615306)

[Цель работы 159](#_Toc90615307)

[Постановка задачи 159](#_Toc90615308)

[Программный код 159](#_Toc90615309)

[Вывод программы 160](#_Toc90615310)

[Вывод 160](#_Toc90615311)

[ЛАБОРАТОРНАЯ РАБОТА №12 ОБРАБОТКА СТРОК. ИСПОЛЬЗОВАНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ В JAVA-ПРИЛОЖЕНИЯХ 161](#_Toc90615312)

[Цель работы 161](#_Toc90615313)

[Теоретическое введение 161](#_Toc90615314)

[Постановка задачи 164](#_Toc90615315)

[Программный код 166](#_Toc90615316)

[Вывод программы 168](#_Toc90615317)

[Вывод 168](#_Toc90615318)

[ЛАБОРАТОРНАЯ РАБОТА № 169](#_Toc90615319)

[Цель работы 169](#_Toc90615320)

[Постановка задачи 169](#_Toc90615321)

[Программный код 169](#_Toc90615322)

[Вывод программы 169](#_Toc90615323)

[Вывод 169](#_Toc90615324)

# ПРАКТИЧЕСКАЯ РАБОТА №1. КЛАССЫ, КАК НОВЫЕ ТИПЫ ДАННЫХ. ПОЛЯ ДАННЫХ И МЕТОДЫ

## Цель работы

Цель данной практической работы – освоить на практике работу с классами на Java.

## Теоретическое введение

В Java, класс является определением объектов одного и того же вида. Другими словами, класс — это тип данных, создаваемый программистом для решения задач. Он представляет из себя шаблон, или прототип, который определяет и описывает статические свойства и динамическое поведение, общие для всех объектов одного и того же вида.

Экземпляр класса - реализация конкретного объекта типа класс. Другими словами, экземпляр экземпляра класса. Все экземпляры класса имеют аналогичные свойства, как задано в определении класса. Например, вы можете определить класс с именем "Студент " и создать три экземпляра класса "Студент": " Петр", " Павел" и " Полина ". Термин "Объект " обычно относится к экземпляру класса. Но он часто используется свободно, которые могут относиться к классу или экземпляру.

Графически можно представить класс в виде UML диаграммы как прямоугольник в виде как трех секций, в котором присутствует секция наименования класса, секция инкапсуляции данных и методов (функций или операций) класса.

Рассмотрим подробнее диаграмму класса. Имя (или сущность): определяет класс.

Переменные (или атрибуты, состояние, поля данных класса): содержит статические атрибуты класса, или описывают свойства класса (сущности предметной области).

Методы (или поведение, функции, работа c данными): описывают динамическое поведение класса. Другими словами, класс инкапсулирует статические свойства (данные) и динамические модели поведения (операции, которые работают с данными) в одном месте (“контейнере” или “боксе”), представленном на рисунке в виде прямоугольника.

1) Класс, тип данных, определяемый программистом, абстрактный тип данных, повторно-используемый программный объект, который имитирует реальные сущности предметной области. Класс можно представить графически в виде контейнера на UML диаграмме, который состоит из трех условных частей и содержит имя, переменные и методы.

2) Класс инкапсулирует статическое состояние объекта, его атрибуты или свойства данных в виде переменных класса и поведение объекта в виде методов, которые могут реализовывать определенные алгоритмы.

3) Значения переменных или поля данные составляют его состояние. Методы создает свои модели поведения.

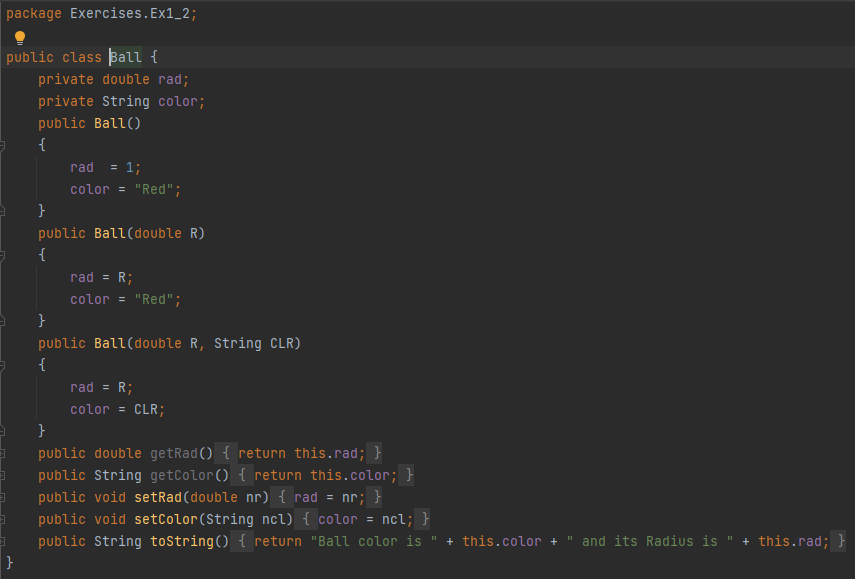
Экземпляр класса — это представление (или реализация) конкретного представителя класса.

## Постановка задачи

Необходимо реализовать простейший класс на языке программирования Java. Не забудьте добавить метод toString() к вашему классу. Так-же в программе необходимо предусмотреть класс-тестер для тестирования класса и вывода информации об объекте.

Упражнение 2. Реализуйте простейший класс «Мяч»

## Программный код



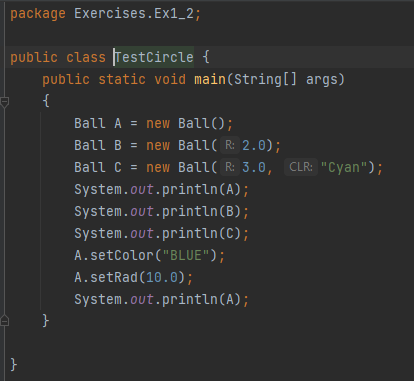


Рис. Код программы

## Вывод программы

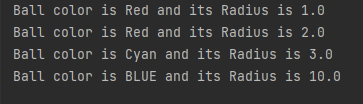


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы смогли освоить на практике работу с классами на Java, научились писать классы, методы, переменные и взаимодействовать с ними.

# ПРАКТИЧЕСКАЯ РАБОТА №2. ИСПОЛЬЗОВАНИЕ UML ДИАГРАММ В ОБЪЕКТНООРИЕНТИРОВАННОМ ПРОГРАММИРОВАНИИ

## Цель работы

Цель данной работы – это научиться работать с UML-диаграммами классов.

## Теоретическое введение

Язык моделирования Unified Modeling Language (UML) является стандартом де-факто с 1998 года для проектирования и документирования объектно-ориентированных программ. Средствами UML в виде диаграмм можно графически изобразить класс и экземпляр класса.

Графически представляем класс в виде прямоугольника, разделенного на три области – область именования класса, область инкапсуляции данных и область операций (методы).

Имя (или сущность) : определяет класс.

Переменные (или атрибуты, состояние, поля данных класса): содержит статические атрибуты класса, или описывают свойства класса (сущности предметной области).

Методы (или поведение, функции, работа c данными): описывают динамическое поведение класса. Другими словами, класс инкапсулирует статические свойства (данные) и динамические модели поведения (операции, которые работают с данными) в одном месте (“коробке” или прямоугольнике).

Приведенные выше диаграммы классов описаны в соответствии с UML нотацией. Класс представляется в этой нотации как прямоугольник, разделенный на три области, одна содержит название, две вторых содержат переменные (поля данных класса) и методы класса, соответственно. Имя класса выделено жирным шрифтом и находится посредине. Экземпляр (объект класса) также представляется в виде прямоугольника, разделенного на три части, в первой части помещается надпись с именем экземпляра, например в instanceName:Classname и выделенная подчеркиванием ( название\_экземпляра : имя\_класса).

## Постановка задачи

По UML диаграмме класса, представленной на рисунке 2.5. написать программу, которая состоит из двух классов. Один из них Ball должен реализовывать сущность мяч, а другой с названием TestBall тестировать работу созданного класса. Класс Ball должен содержать реализацию методов, представленных на UML. Диаграмма на рисунке описывает сущность Мяч написать программу

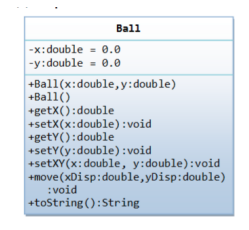


Рис. Пример UML диаграммы

Класс Ball моделирует движущийся мяч. В состав класса входят:

• Две переменные с модификатором private (поля данных класса): х, у, которые описывают положение мяча на поле.

• Конструкторы, public методы получения и записи значений для private переменных.

• Метод setXY (), который задает положение мяча и метод setXYSpeed(), чтобы задать скорость мяча

• Метод move() , позволяет переместить мяч, так что что увеличивает х и у на данном участке на xDisp и yDisp, соответственно.

• Метод toString(), который возвращает "Ball @ (х , у) " .

## Программный код

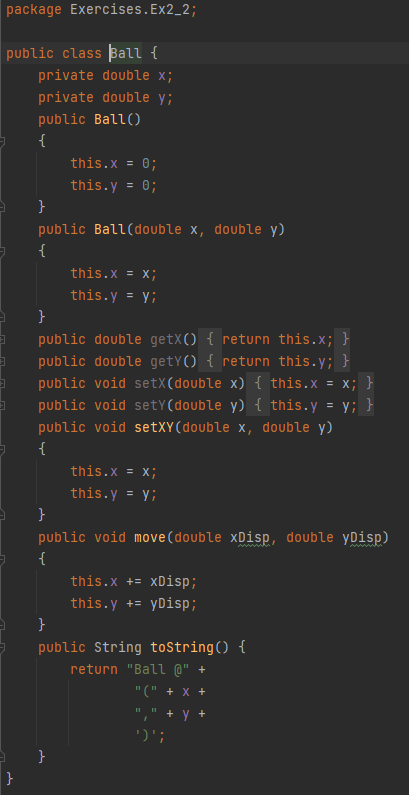




Рис. Код программы

## Вывод программы

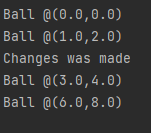


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы, мы научились создавать классы, основываясь на UML диаграммах

# ПРАКТИЧЕСКАЯ РАБОТА № 3. НАСЛЕДОВАНИЕ. АБСТРАКТНЫЕ СУПЕРКЛАССЫ И ИХ ПОДКЛАССЫ В JAVA.

## Цель работы

Цель работы: освоить на практике работу с абстрактными классами и наследованием на Java.

## Теоретическое введение

Класс, содержащий абстрактные методы, называется абстрактным классом. Такие классы при определении помечаются ключевым словом abstract.

Абстрактный метод внутри абстрактного класса не имеет тела, только прототип. Он состоит только из объявления и не имеет тела.

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.

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.

## Постановка задачи

Напишите два класса MovablePoint и MovableCircle - которые реализуют интерфейс Movable.

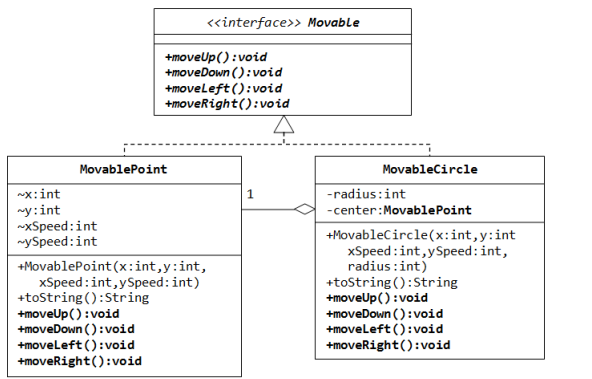


Рис. UML диаграмма задание

## Программный код

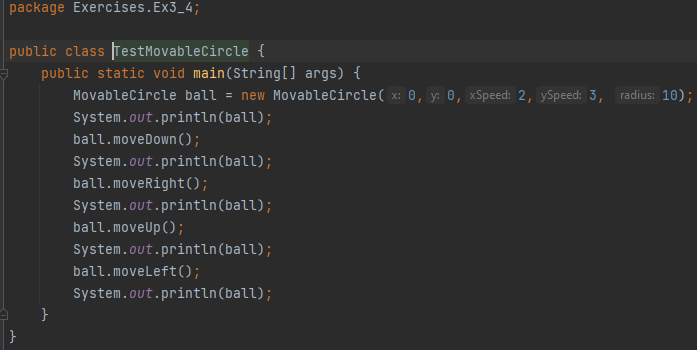
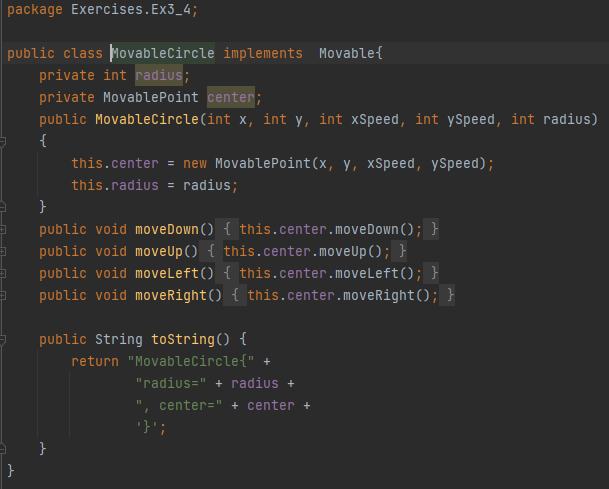
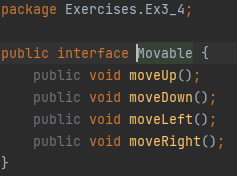


Рис. Код программы

## Вывод программы

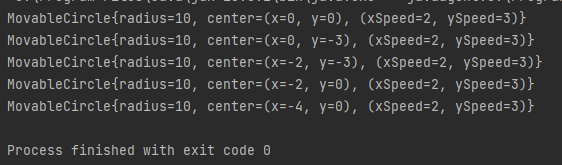


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы, мы научились работать и создавать абстрактные классы, изучили наследование, и научились создавать и реализовывать интерфейсы.

# ПРАКТИЧЕСКАЯ РАБОТА № 4. СОЗДАНИЕ GUI. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ В JAVA.

## Цель работы

Цель работы: введение в событийное программирование на языке Java.

## Теоретическое введение

Данная практическая работа посвящена закреплению практических навыков по созданию приложений на Java c использованием следующих элементов GUI:

• Текстовые поля и области ввода текста; • Менеджеры компоновки компонентов;

• Слушатель мыши;

• Создание меню.

Text Fields - текстовое поле или поля для ввода текста (можно ввести только одну строку). Примерами текстовых полей являются поля для ввода логина и пароля, например, используемые, при входе в электронную почту.

Пример создания объекта класса JTextField:

JTextField jta = new JTextField (10);

В параметрах конструктора задано число 10, это количество символов, которые могут быть видны в текстовом поле. Текст веденный в поле JText может быть возвращен с помощью метода getText(). Также в поле можно записать новое значение с помощью метода setText(String s).

Как и у других компонентов, мы можем изменять цвет и шрифт текста в текстовом поле.

Ответственность за выполнение проверки на наличие ошибок в коде лежит полностью на программисте, например, чтобы проверить произойдет ли ошибка, когда в качестве входных данных в JTextField ожидается ввод числа. Компилятор не будет ловить такого рода ошибку, поэтому ее необходимо обрабатывать пользовательским кодом.

Компонент TextArea похож на TextField, но в него можно вводить более одной строки. В качестве примера TextArea можно рассмотреть текст, который мы набираем в теле сообщения электронной почты.

Менеджер BorderLayout: разделяет компонент на пять областей (WEST, EAST, NOTH, SOUTH and Center). Другие компоненты могут быть добавлены в любой из этих компонентов пятерками.

Обратите внимание, что мы можем, например, добавить панели JPanel в эти области и затем добавлять компоненты этих панелей. Мы можем установить расположение этих JPanel используя другие менеджеры.

С помощью менеджера GridLayout компонент может принимать форму таблицы, где можно задать число строк и столбцов.

Если компоненту GridLayout задать 3 строки и 4 столбца, то компоненты будут принимать форму таблицы, показанной выше, и будут всегда будут добавляться в порядке их появления

Иногда бывает нужно изменить размер и расположение компонента в контейнере. Таким образом, мы должны указать программе не использовать никакой менеджер компоновки, то есть (setLayout (нуль)).

Добавление меню в программе Java проста. Java определяет три компонента для обработки:

• JMenuBar: который представляет собой компонент, который содержит меню.

• JMenu: который представляет меню элементов для выбора.

• JMenuItem: представляет собой элемент, который можно кликнуть из меню.

Подобно компоненту Button (на самом деле MenuItems являются подклассами класса AbstractButton). Мы можем добавить ActionListener к ним так же, как мы делали с кнопками

## Постановка задачи

Напишите интерактивную программу с использованием GUI имитирует таблицу результатов матчей между командами Милан и Мадрид. Создайте JFrame приложение, у которого есть следующие компоненты GUI: • одна кнопка JButton labeled “AC Milan”

• другая JButton подписана “Real Madrid” • надпись JLabel содержит текст “Result: 0 X 0”

• надпись JLabel содержит текст “Last Scorer: N/A”

• надпись Label содержит текст “Winner: DRAW”;

Всякий раз, когда пользователь нажимает на кнопку AC Milan, результат будет увеличиваться для Милана, сначала 1 X 0, затем 2 X 0 и так далее. Last Scorer означает последнюю забившую команду. В этом случае: AC Milan. Если пользователь нажимает кнопку для команды Мадрид, то счет приписывается ей. Победителем становится команда, которая имеет больше кликов кнопку на соответствующую, чем другая.

## Программный код



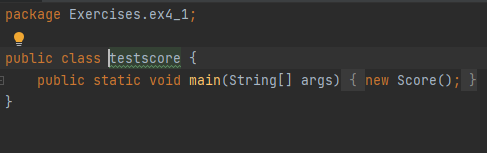


Рис. Код программы

## Вывод программы



Рис. Приложение сделанное в коде

## Вывод

В результате выполнения данной практической работы, мы научились создавать базовые элементы приложения на Java, используя JPannel, JButton, Border Layout и другие.

# ПРАКТИЧЕСКАЯ РАБОТА № 5. ПРОГРАММИРОВАНИЕ РЕКУРСИИ В JAVA

## Цель работы

Цель работы: разработка и программирование рекурсивных алгоритмов на языке Java.

## Теоретическое введение

В контексте языка программирования рекурсия — это некий активный метод (или подпрограмма) вызываемый сам по себе непосредственно, или вызываемой другим методом (или подпрограммой) косвенно. В первую очередь надо понимать, что рекурсия — это своего рода перебор. Вообще говоря, всё то, что решается итеративно можно решить рекурсивно, то есть с использованием рекурсивной функции.

Так же, как и у перебора (цикла) у рекурсии должно быть условие остановки — базовый случай (иначе также, как и цикл, рекурсия будет работать вечно — infinite). Это условие и является тем случаем, к которому рекурсия идет (шаг рекурсии). При каждом шаге вызывается рекурсивная функция до тех пор, пока при следующем вызове не сработает базовое условие и не произойдет остановка рекурсии (а точнее возврат к последнему вызову функции). Всё решение сводится к поиску решения для базового случая. В случае, когда рекурсивная функция вызывается для решения сложной задачи (не базового случая) выполняется некоторое количество рекурсивных вызовов или шагов, с целью сведения задачи к более простой. И так до тех пор, пока не получим базовое решение.

Итак, рекурсивная функция состоит из:

• условие остановки или же базового случая или условия;

• условие продолжения или шага рекурсии — способ сведения сложной задачи к более простым подзадачам

## Постановка задачи

12. Вывести нечетные числа последовательности. Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Выведите все нечетные числа из этой последовательности, сохраняя их порядок. В этой задаче нельзя использовать глобальные переменные и передавать какие-либо параметры в рекурсивную функцию. Функция получает данные, считывая их с клавиатуры. Функция не возвращает значение, а сразу же выводит результат на экран. Основная программа должна состоять только из вызова этой функции.

13. Вывести члены последовательности с нечетными номерами Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Выведите первое, третье, пятое и т.д. из введенных чисел. Завершающий ноль выводить не надо. В этой задаче нельзя использовать глобальные переменные и передавать какие-либо параметры в рекурсивную функцию. Функция получает данные, считывая их с клавиатуры. Функция не возвращает значение, а сразу же выводит результат на экран. Основная программа должна состоять только из вызова этой функции.

14. Цифры числа слева направо Дано натуральное число N. Выведите все его цифры по одной, в обычном порядке, разделяя их пробелами или новыми строками. При решении этой задачи нельзя использовать строки, списки, массивы (ну и циклы, разумеется). Разрешена только рекурсия и целочисленная арифметика

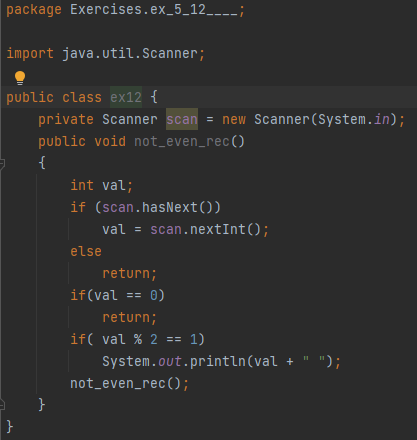
15. Цифры числа справа налево Дано натуральное число N. Выведите все его цифры по одной, в обратном порядке, разделяя их пробелами или новыми строками. При решении этой задачи нельзя использовать строки, списки, массивы (ну и циклы, разумеется). Разрешена только рекурсия и целочисленная арифметика.

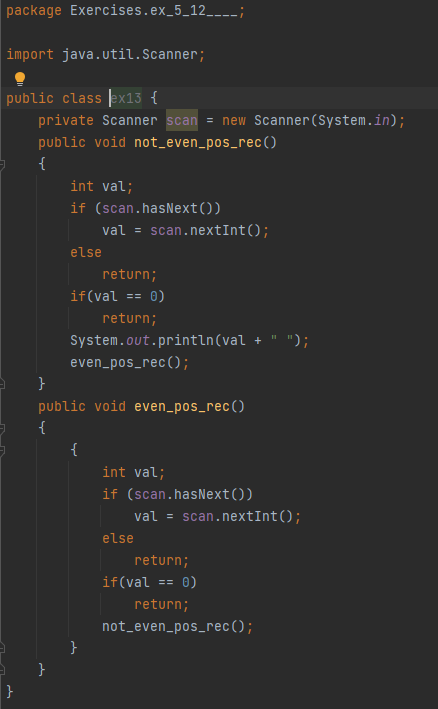
16. Количество элементов, равных максимуму Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Определите, какое количество элементов этой последовательности, равны ее наибольшему элементу.

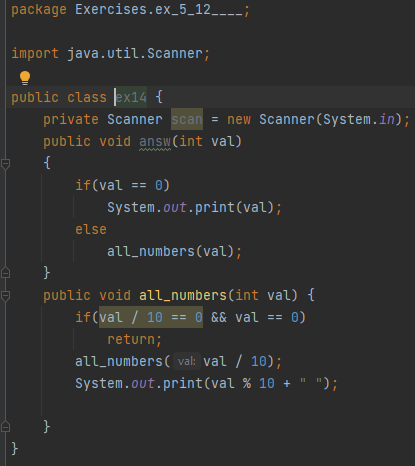
В этой задаче нельзя использовать глобальные переменные. Функция получает данные, считывая их с клавиатуры, а не получая их в виде параметра. В программе на языке Python функция возвращает результат в виде кортежа из нескольких чисел, и функция вообще не получает никаких параметров. В программе на языке C++ результат записывается в переменные, которые передаются в функцию по ссылке. Других параметров, кроме как используемых для возврата значения, функция не получает.

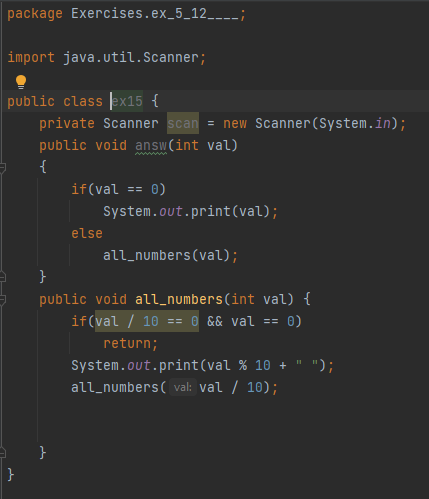
Гарантируется, что последовательность содержит хотя бы одно число (кроме нуля)

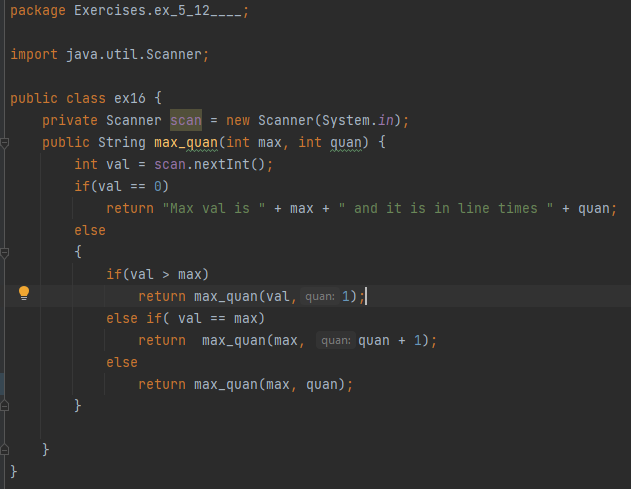
## Программный код











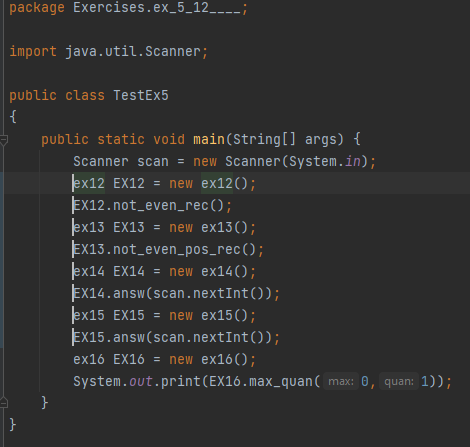
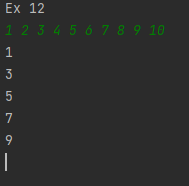
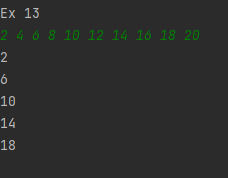
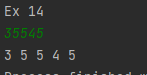


Рис. Код программы

## Вывод программы







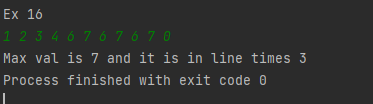


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы, мы научились программировать рекурсивные функции в Java. Узнали, что такое глубина, шаг и условие рекурсии. Также написали задачи на рекурсивные функции.

# ПРАКТИЧЕСКАЯ РАБОТА №6. ТЕХНИКИ СОРТИРОВКИ В JAVA

## Цель работы

Цель работы: освоение на практике методов сортировки с использованием приемов программирования на объектно-ориентированном языке Java.

## Теоретическое введение

Сортировка — это процесс упорядочивания списка элементов (организация в определенном порядке) исходного списка элементов, который возможно организован в виде контейнера или храниться в виде коллекции.

Процесс сортировки основан на упорядочивании конкретных значений, например:

• сортировка списка результатов экзаменов баллов в порядке возрастания результата;

• сортировка списка людей в алфавитном порядке по фамилии.

Есть много алгоритмов для сортировки списка элементов, которые различаются по эффективности.

Алгоритм сортировки вставками.

Работа метода сортировки состоит из следующих шагов:

• выбрать любой элемент из списка элементов и вставить его в надлежащее место в отсортированный подсписок;

• повторять предыдущий шаг, до тех пор, пока все элементы не будут вставлены. Более детально:

• рассматриваем первый элемент списка как отсортированный подсписок (то есть первый элемент списка);

• вставим второй элемент в отсортированный подсписок, сдвигая первый элемент по мере необходимости, чтобы освободить место для вставки нового элемента;

• вставим третий элемент в отсортированный подсписок (из двух элементов), сдвигая элементы по мере необходимости;

• повторяем до тех пор, пока все значения не будут вставлены на свои соответствующие позиции.

Алгоритм быстрой сортировки (Quick Sort).

Состоит из последовательного выполнения двух шагов:

• массив A[1..n] разбивается на два непустых подмассивов по отношению к "опорному элементу”;

• два подмассива сортируются рекурсивно посредством Quick Sort.

Алгоритм сортировка слиянием (Merge Sort).

Состоит из последовательного выполнения трех шагов:

• разделить массив A[1..n] на 2 равные части;

• провести сортировку слиянием двух подмассивов (рекурсивно);

• объединить (соединить) два отсортированных подмассива.

Использование полиморфизма в сортировке.

Техника программирования сортировок в Java отличается от написания алгоритмов на процедурных языках программирования. При написании кода большим преимуществом является использование основного принципа ООП – полиморфизма. Напомним, что класс, который реализует интерфейс Comparable определяет метод compareTo(), чтобы определить относительный порядок своих объектов.

Таким образом мы можем использовать полиморфизм, чтобы разработать обобщенную сортировку для любого набора Comparable объектов.

При разработке класса, реализующего метод сортировки, нужно помнить, что метод принимает в качестве параметра массив объектов типа Comparable или фактически полиморфных ссылок.

Таким образом, один метод может быть использован для сортировки любых объектов, например: People (людей), Books (книг), или любой каких-либо других объектов.

Методу сортировки все-равно, что именно он будет сортировать, ему только необходимо иметь возможность вызвать метод compareTo().

Это обеспечивается использованием в качестве типа формального параметра интерфейса Comparable или интерфейсной ссылки.

Кроме того, таким образом каждый класс “для себя” решает, что означает для одного объекта, быть меньше, чем другой.

## Постановка задачи

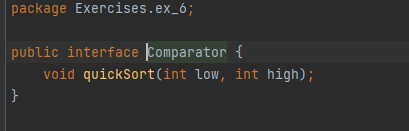
Упражнение 2.

Напишите класс SortingStudentsByGPA который реализует интерфейс Comparator таким образом, чтобы сортировать список студентов по их итоговым баллам в порядке убывания с использованием алгоритма быстрой сортировки.

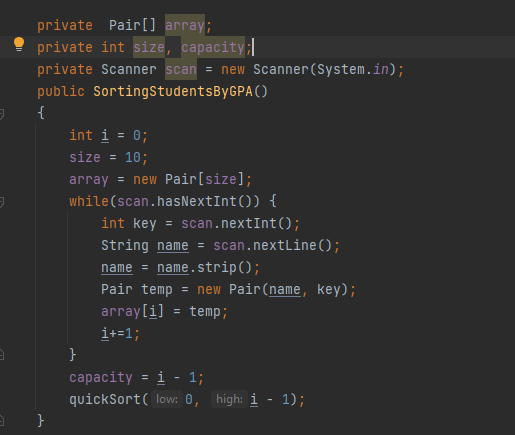
Упражнение 3.

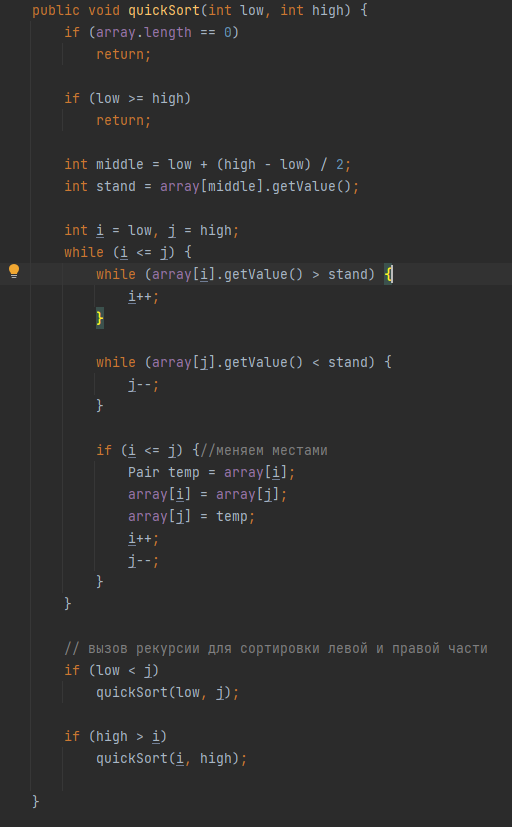
Напишите программу, которая объединяет два списка данных о студентах в один отсортированный списках с использованием алгоритма сортировки слиянием

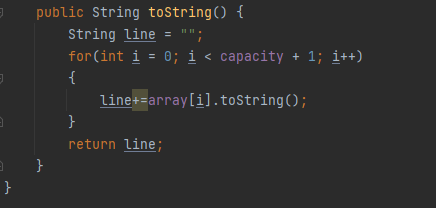
## Программный код

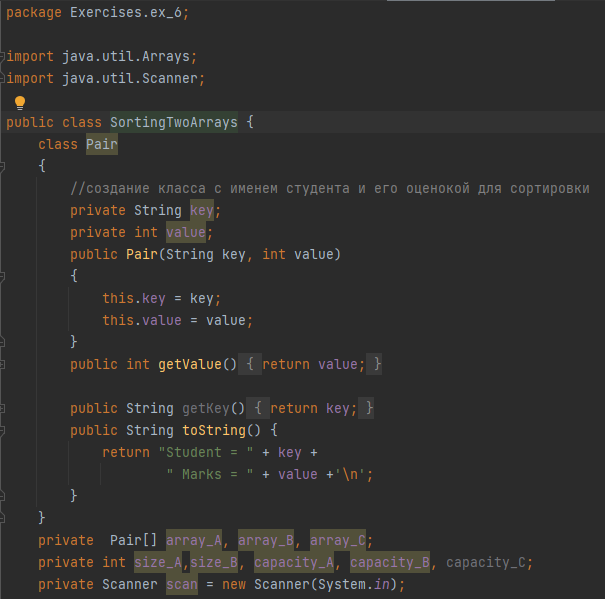




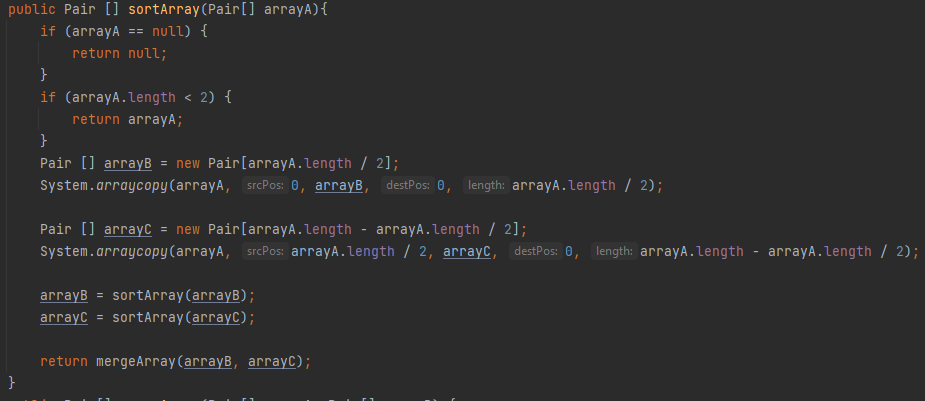


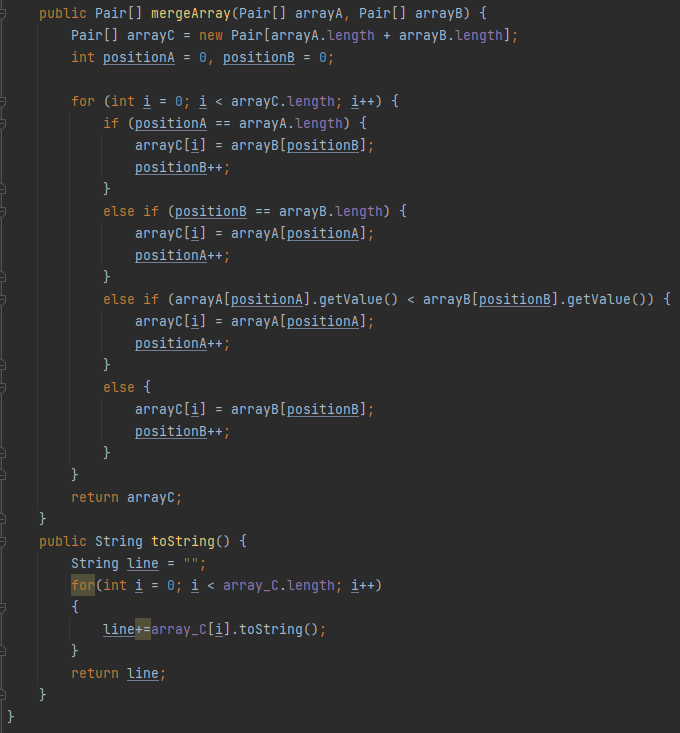












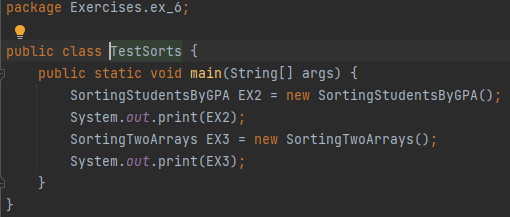
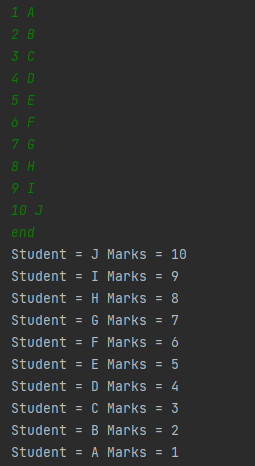


Рис. Код программы

## Вывод программы



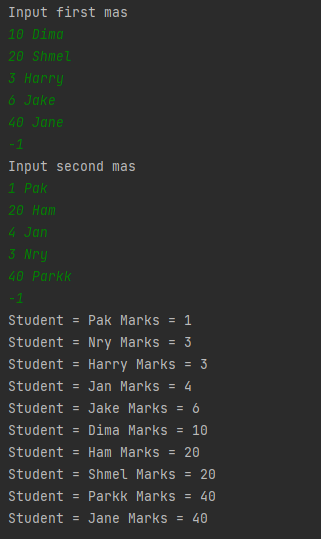


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы изучили и научились разным алгоритмам сортировки. А также научились реализовывать из в Java.

# ПРАКТИЧЕСКАЯ РАБОТА № 7. ИСПОЛЬЗОВАНИЕ СТАНДАРТНЫХ КОНТЕЙНЕРНЫХ КЛАССОВ ПРИ ПРОГРАММИРОВАНИИ НА JAVA

## Цель работы

Цель работы: изучение на практике приемов работы со стандартными контейнерными классами Java Collection Framework.

## Теоретическое введение

Java Collections Framework — это набор связанных классов и интерфейсов, реализующих широко используемые структуры данных — коллекции. На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие в фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).

Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но, как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — ArrayList.

Stack — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например ArrayDeque.

ArrayList — как и Vector является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предполагается частое обращение к элементам по индексу. Из-за особенностей реализации обращение к элементам по индексу, которое выполняется за константное время O(1). Использование данной коллекции рекомендуется избегать, если требуется частое удаление/добавление элементов в середине коллекции.

LinkedList — вид реализации List. Позволяет хранить любые данные, включая null. Данная коллекция реализована на основе двунаправленного связного списка (каждый элемент списка имеет ссылки на предыдущий и следующий). Добавление и удаление элемента из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное время O(1). Ввиду реализации, данную коллекцию можно использовать как абстрактный тип данных — стек или очередь. Для этого в ней реализованы соответствующие методы

Интерфейс Set.

Представляет собой неупорядоченную коллекцию, которая не может содержать одинаковые элементы и является программной моделью математического понятия «множество».

Интерфейс Queue.

Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь.

PriorityQueue — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null.

Как заявлено в документации, эта коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO коллекция

## Постановка задачи

Напишите программу в виде консольного приложения, которая моделирует карточную игру «пьяница» и определяет, кто выигрывает. В игре участвует 10 карт, имеющих значения от 0 до 9, большая карта побеждает меньшую; карта «0» побеждает карту «9».

Карточная игра “ В пьяницу”. В этой игре карточная колода раздается поровну двум игрокам. Далее они открывают по одной верхней карте, и тот, чья карта старше, забирает себе обе открытые карты, которые кладутся под низ его колоды. Тот, кто остается без карт, - проигрывает.

Для простоты будем считать, что все карты различны по номиналу и что самая младшая карта побеждает самую старшую карту (“шестерка берет туз”).

Игрок, который забирает себе карты, сначала кладет под низ своей колоды карту первого игрока, затем карту второго игрока (то есть карта второго игрока оказывается внизу колоды).

Входные данные

Программа получает на вход две строки: первая строка содержит 5 карт первого игрока, вторая - 5 карт второго игрока. Карты перечислены сверху вниз, то есть каждая строка начинается с той карты, которая будет открыта первой.

Выходные данные

Программа должна определить, кто выигрывает при данной раздаче, и вывести слово first или second, после чего вывести количество ходов, сделанных до выигрыша. Если на протяжении 106 ходов игра не заканчивается, программа должна вывести слово botva.

Пример ввода

1 3 5 7 9

2 4 6 8 0

Пример вывода

second 5

Упражнение 1

Используйте для организации хранения структуру данных Stack.

Упражнение 2

Используйте для организации хранения структуру данных Queue.

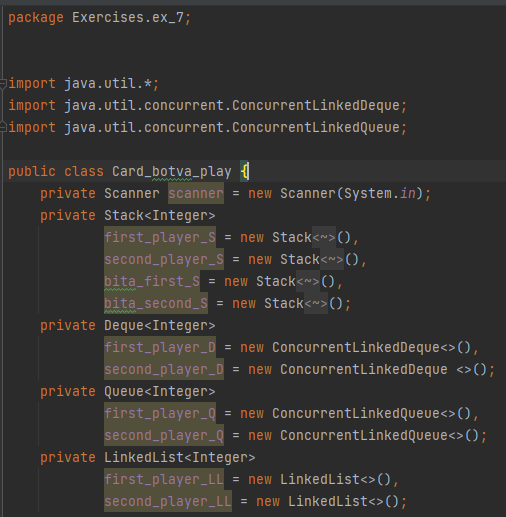
Упражнение 3

Используйте для организации хранения структуру данных Dequeue

Упражнение 4

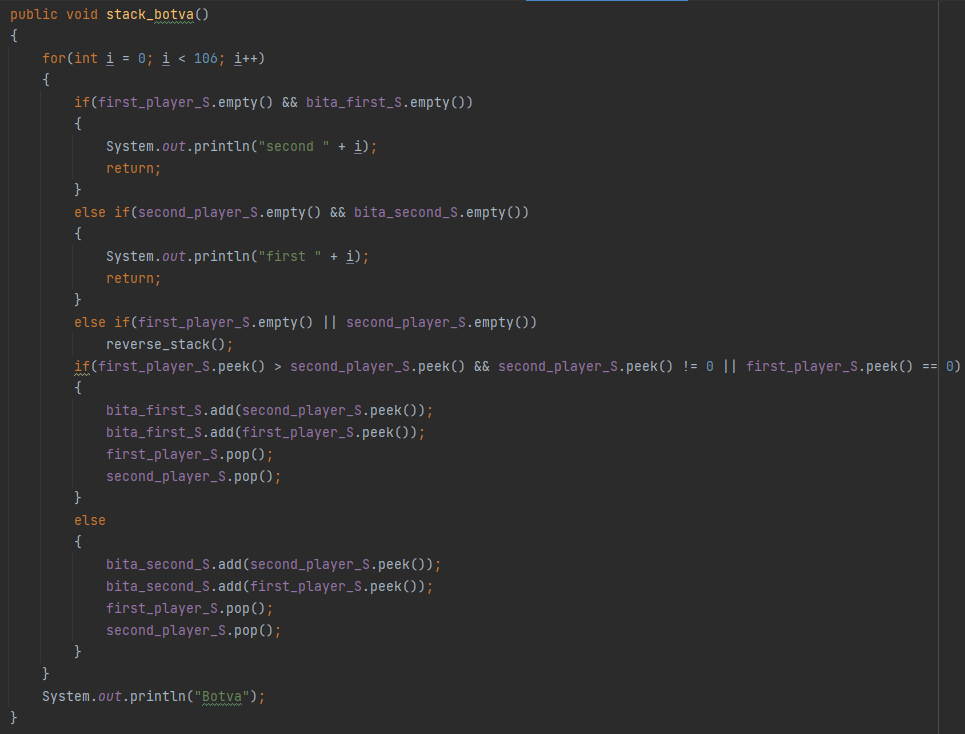
Используйте для организации хранения структуру данных DoubleList.

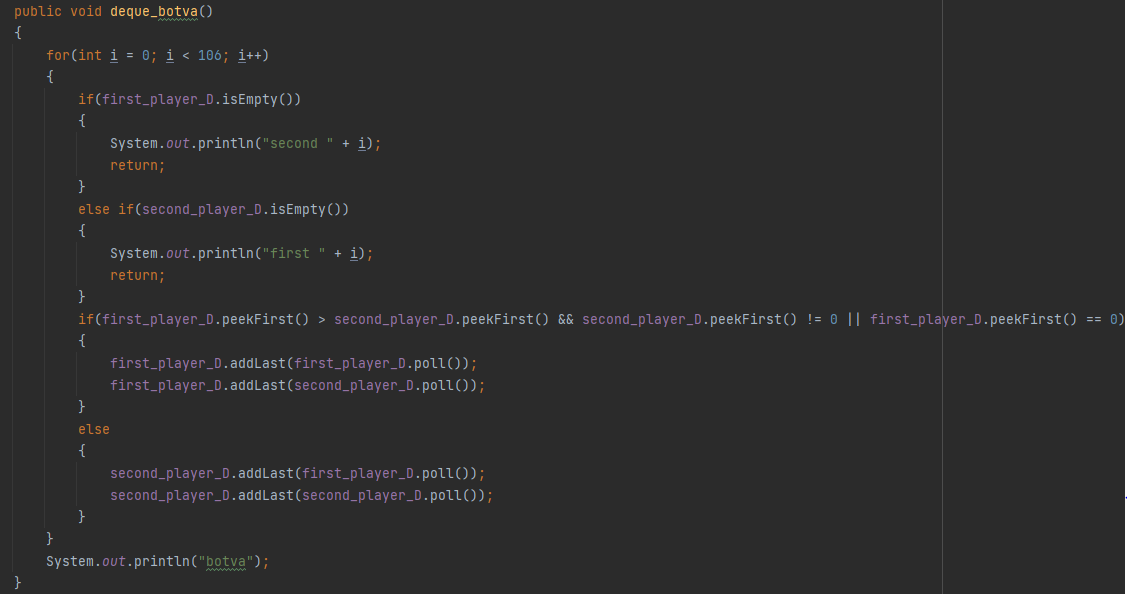
## Программный код



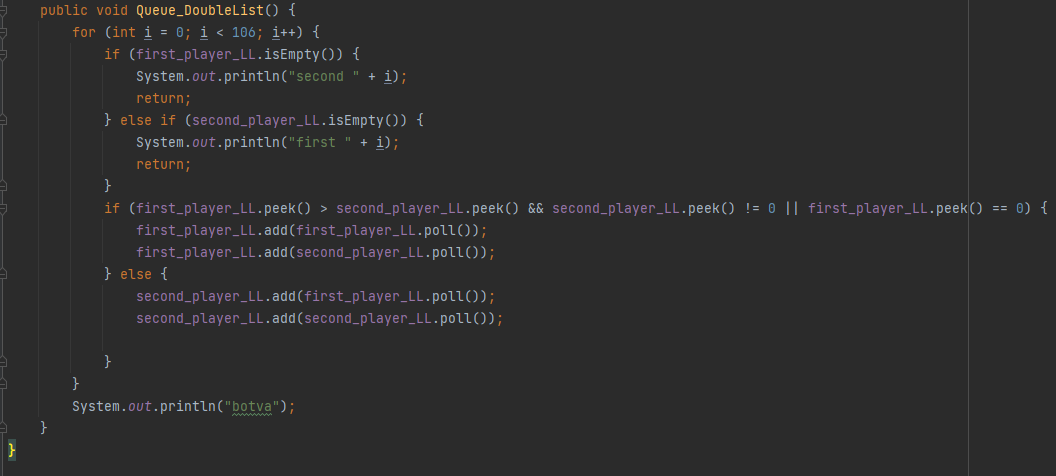












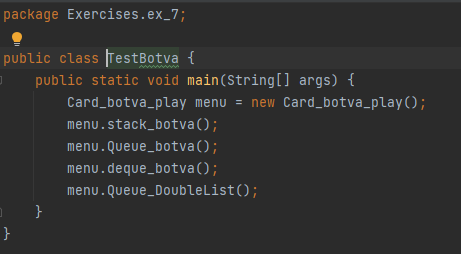


Рис. Код программы

## Вывод программы

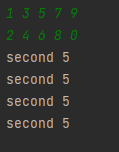


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы изучили контейнеры в Java, как их реализовывать и использовать. Узнали их различия и процесс работы.

# ПРАКТИЧЕСКАЯ РАБОТА № 8 РАЗЛИЧНЫЕ ВИДЫ СПИСКОВ ОЖИДАНИЯ

## Постановка задачи

Вначале, рассмотрите класс, представленный на диаграмме - общий класс список ожидания. Допустим мы решили, что нам понадобиться еще два вида списков ожидания:

• BoundedWaitList: Этот список ожидания имеет ограниченную емкость, указываемую в момент создания. Он не принимает более элементов, чем заранее задано (возможное количество потенциальных элементов в списке ожидания).

• UnfairWaitList: В этом списке ожидания, можно удалить элемент, который не является первым в очереди - и помните он не может вернуться обратно! (Возможны различные реализации, но в вашей реализации необходимо удалить первое вхождение данного элемента.) Также возможно, чтобы например, первый элемент будет отправлен обратно в конец списка.

После описания всей задачи в целом, мы сможем решить, что мы нам нужен интерфейс IWaitList, и затем нужно создать три разных класса для трех списков ожидания. Также предполагается, что один из списков ожидания должен быть супер классом для двух других списков ожидания.

Задание

Исследуйте UML диаграмму классов на рисунке и понаблюдайте, как она выражает то, что мы говорили выше в словах. Убедитесь, что вы понимаете все аспекты диаграммы.

Расширить и модифицировать исходный код WaitList, как необходимо, чтобы полностью реализовать всю схему UML. Включить комментарии Javadoc. Обратите внимание на переключение ролей после реализации каждого интерфейса / класса!

Изучение работу метода main(), которая использует ваши новые классы и интерфейс.

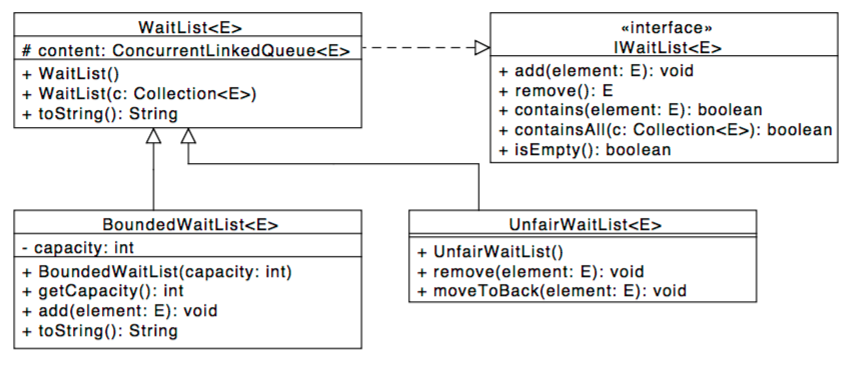
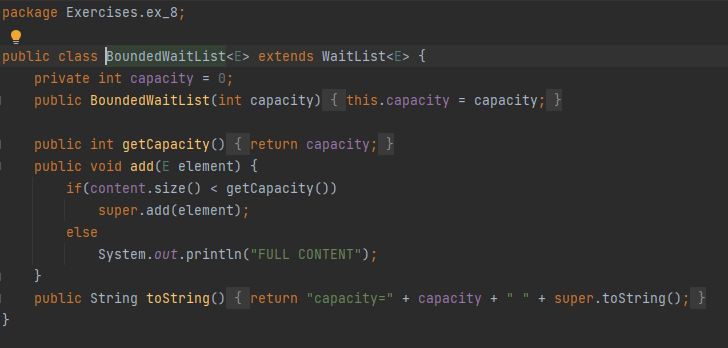
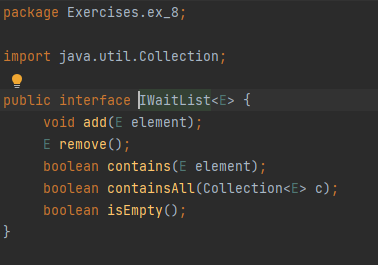
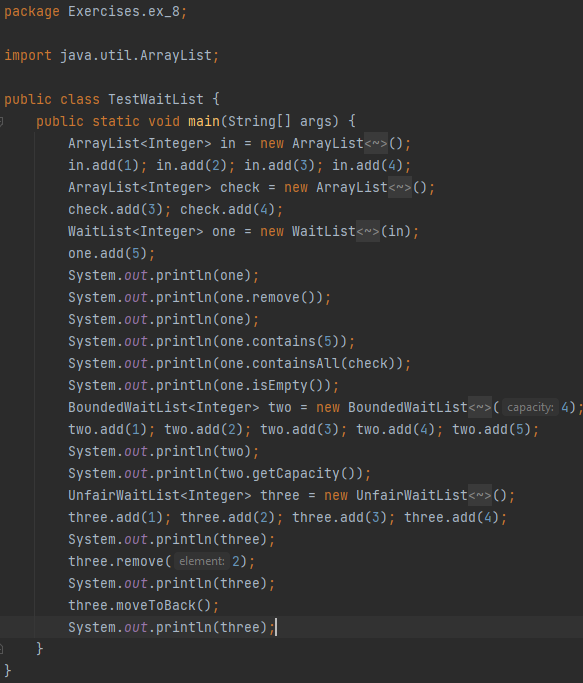


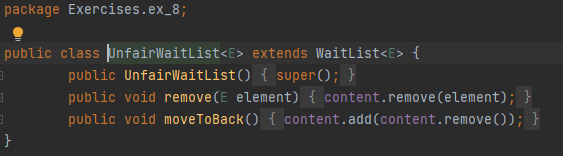
Рис. UML диаграмма задания

## Программный код









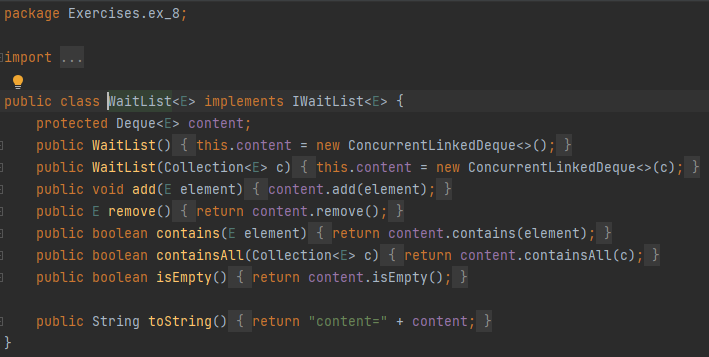


Рис. Код программы

## Вывод программы

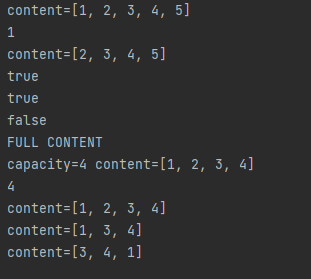


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы познакомились с WaitList также научились описывать собственные структуры, в которые мы можем записывать и извлекать данные, с описанными нами условиями.

# ПРАКТИЧЕСКАЯ РАБОТА №9 СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ИСКЛЮЧЕНИЙ

## Цель работы

Цель данной практической работы – научиться создавать собственные исключения.

## Теоретическое введение

Язык Java предоставляет исчерпывающий набор классов исключений, но иногда при разработке программ вам потребуется создавать новые – свои собственные исключения, которые являются специфическими для потребностей именно вашего приложения. В этой практической работе вы научитесь создавать свои собственные пользовательские классы исключений. Как вы уже знаете, в Java есть два вида исключений- проверяемы и непроверяемые. Для начала рассмотрим создание пользовательских проверяемых исключений.

Создание проверяемых пользовательских исключений

Проверяемые исключения — это исключения, которые необходимо обрабатывать явно. Рассмотрим пример кода:

try (Scanner file = new Scanner(new File(fileName)))

if (file.hasNextLine()) return file.nextLine(); }

catch(FileNotFoundException e) {

// Logging, etc

}

Приведенный выше код является классическим способом обработки проверяемых исключений на Java. Хотя код выдает исключение FileNotFoundException, но в целом неясно, какова точная причина ошибки – не такого файла нет или же имя файла является недопустимым.

Чтобы создать собственное пользовательское исключение, мы будем наследоваться от класса java.lang.Exception. Давайте рассмотрим пример как это реализуется на практике и создадим собственный класс для проверяемого исключения с именем BadFileNameException:

public class BadFileNameException extends Exception {

public BadFileNameException(String errorMessage) {

super(errorMessage);

}

}

Обратите внимание, что мы также должны написать конструктор в нашем классе, который принимает параметр типа String в качестве сообщения об ошибке, в котором вызывается конструктор родительского класса. Фактически это все, что нам нужно сделать, чтобы определить свое собственное пользовательское исключение.

Далее, давайте посмотрим, как мы можем использовать пользовательское исключение в программе:

try (Scanner file = new Scanner(new File(fileName))) {

if (file.hasNextLine())

return file.nextLine();

}

catch (FileNotFoundException e) {

if (!isCorrectFileName(fileName))

{

throw new BadFileNameException("Bad filename : " + fileName );

} //...

}

Мы создали и использовали свое собственное пользовательское исключение, теперь в случае ошибки, можно понять, что произошло, и какое именно исключение сработало. Как вы думаете, этого достаточно? Если ваш ответ да, то мы не узнаем основную причину, по которой сработало исключения. Как исправить программу. Для этого мы также можем добавить параметр java.lang.Throwable в конструктор. Таким образом, мы можем передать родительское исключение во время вызова метода:

public BadFileNameException(String errorMessage, Throwable err) {

super(errorMessage, err);

}

Теперь мы связали BadFileNameException с основной причиной возникновения данного исключения, например:

try (Scanner file = new Scanner(new File(fileName))) {

if (file.hasNextLine())

{

return file.nextLine();

}

}

catch (FileNotFoundException err) {

if (!isCorrectFileName(fileName)) {

throw new BadFileNameException( "Bad filename: " + fileName, err);

}

// ...

}

Мы рассмотрели, как мы можем использовать пользовательские исключения в программах, учитывая их связь с причинами по которым они могут возникать.

Создание непроверяемых пользовательских исключений

В том же примере, который мы рассматривали выше предположим, что нам нужно такое пользовательское исключение, в котором обрабатывается ошибка, если файла не содержит расширения.

В этом случае нам как раз понадобится создать пользовательское непроверяемое исключение, похожее на предыдущее, потому что данная ошибка будет обнаружена только во время выполнения участка кода. Чтобы создать собственное непроверяемое исключение, нам нужно наследоваться от класса java.lang.RuntimeException:

public class BadFileExtensionException extends RuntimeException { public BadFileExtensionException(String errorMessage, Throwable err) { super(errorMessage, err); } }

Теперь, мы можем использовать это нестандартное исключение в рассматриваемом нами выше примере:

try (Scanner file = new Scanner(new File(fileName))) { if (file.hasNextLine()) { return file.nextLine(); } else { throw new IllegalArgumentException("Non readable file"); } } catch (FileNotFoundException err) { if (!isCorrectFileName(fileName)) { throw new BadFileNameException( "Bad filename: " + fileName , err); } //... } catch(IllegalArgumentException err) { if(!containsExtension(fileName)) { throw new BadFileExtensionException( "Filename does not contain extension: " + fileName, err); } //... }

Заключение В приведенных выше примерах мы рассмотрели основные особенности обработки исключений.

## Постановка задачи

Клиент совершает покупку онлайн. При оформлении заказа у пользователя запрашивается фио и номер ИНН. В программе проверяется, действителен ли номер ИНН для такого клиента. Исключение будет выдано в том случае, если введен недействительный ИНН.

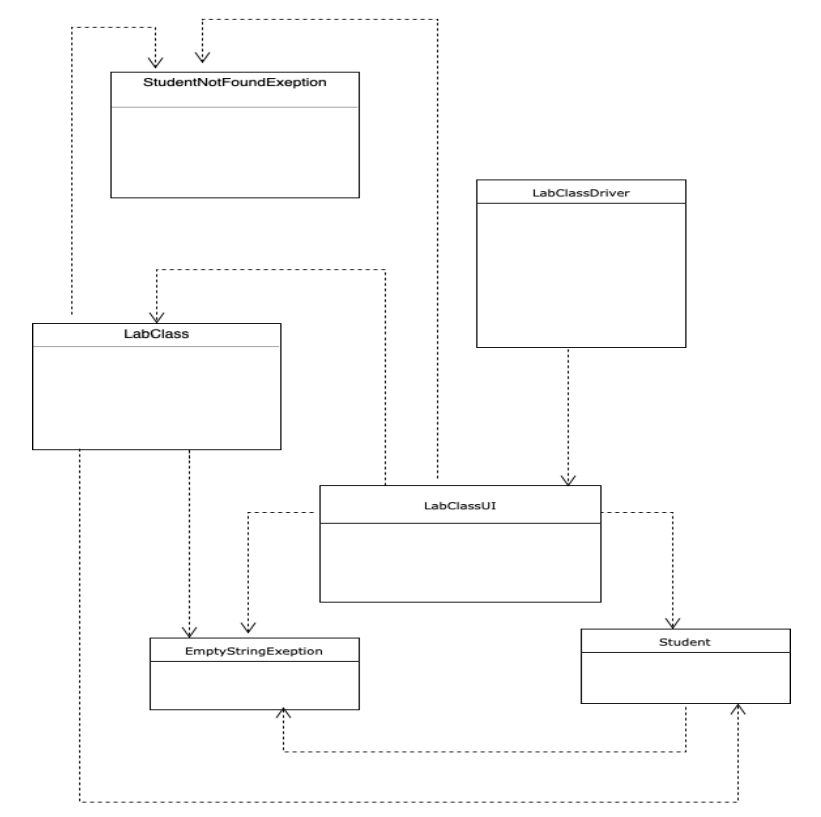
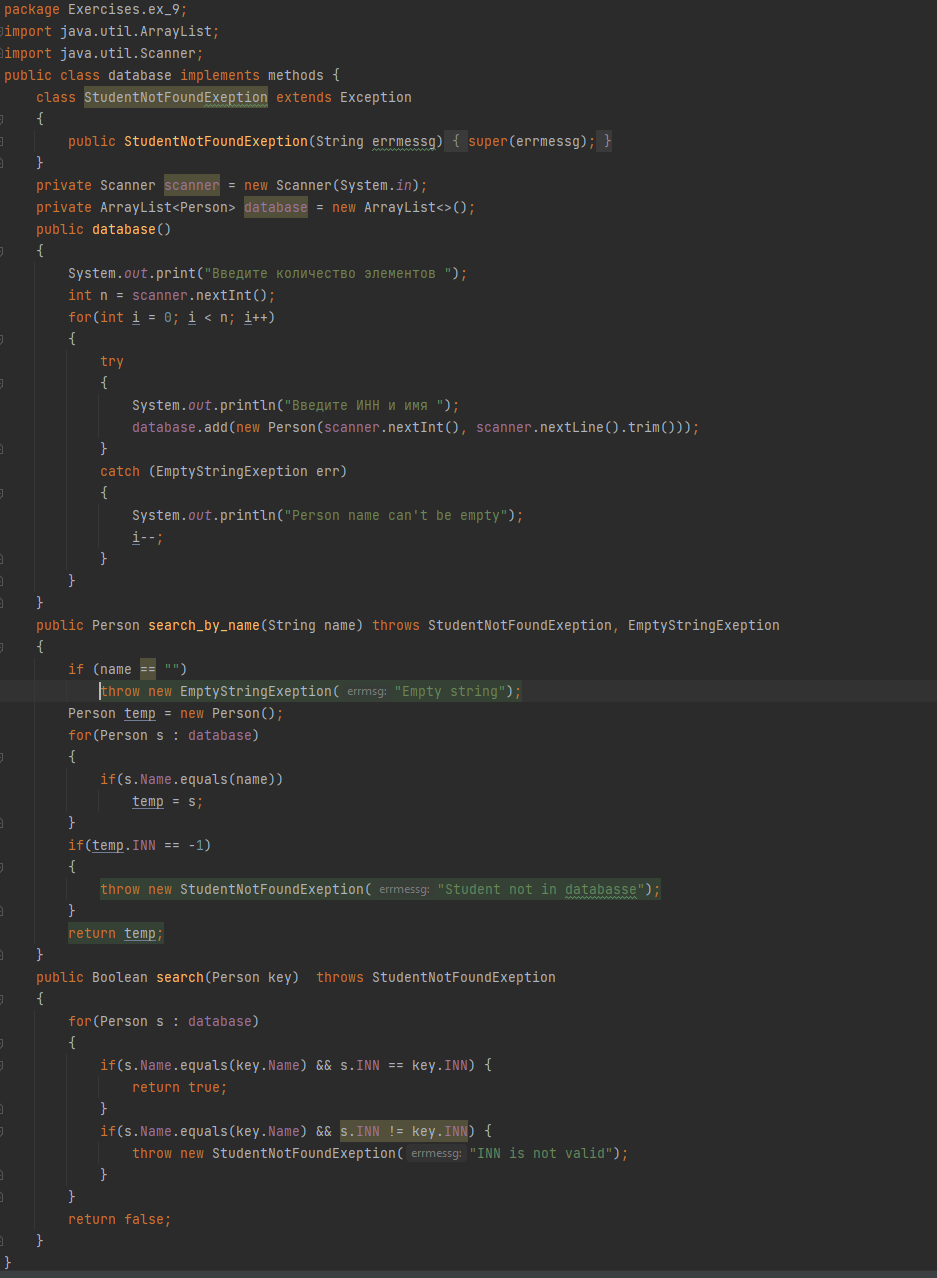


Рис. Диаграмма логики исключений

## Программный код



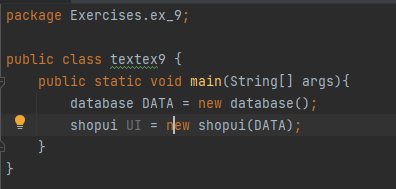
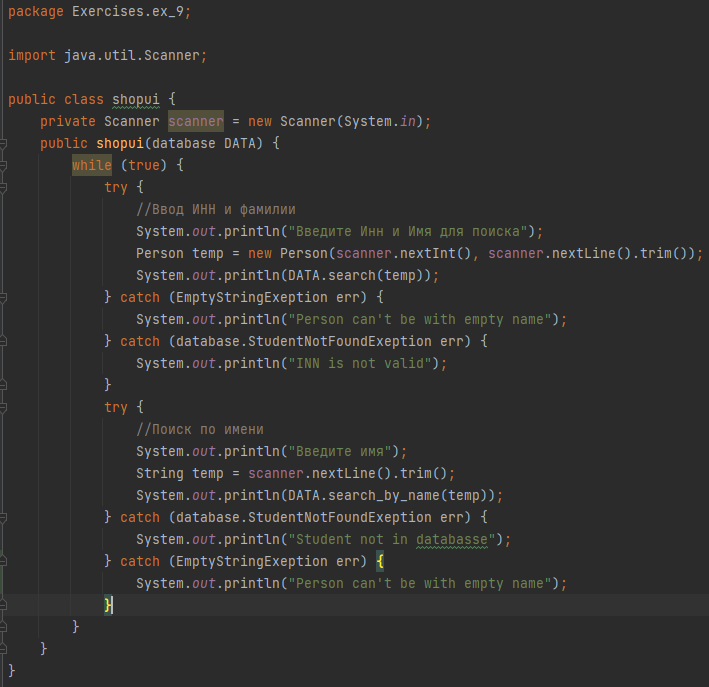
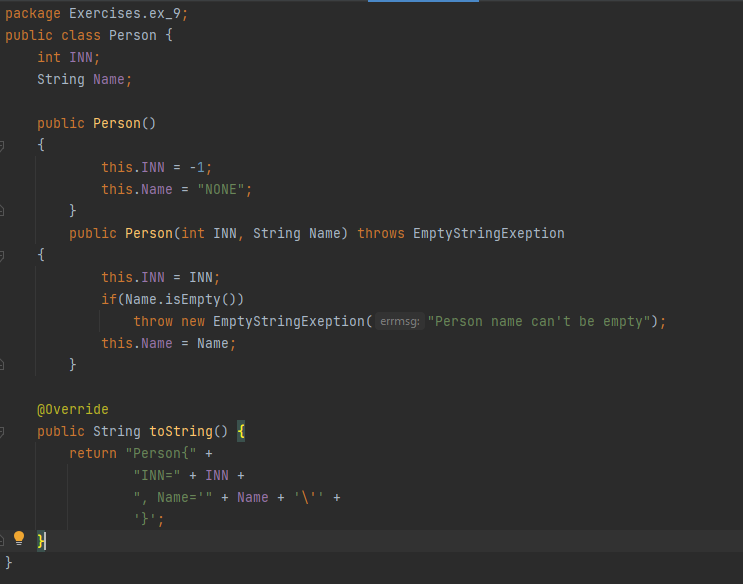
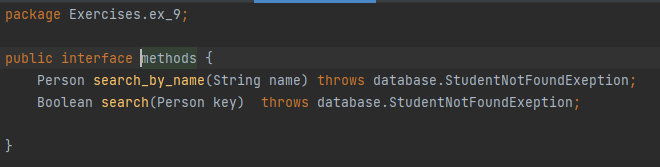
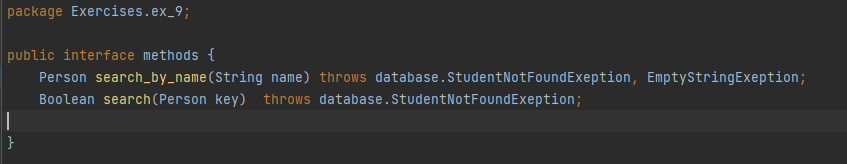


Рис. Код программы

## Вывод программы

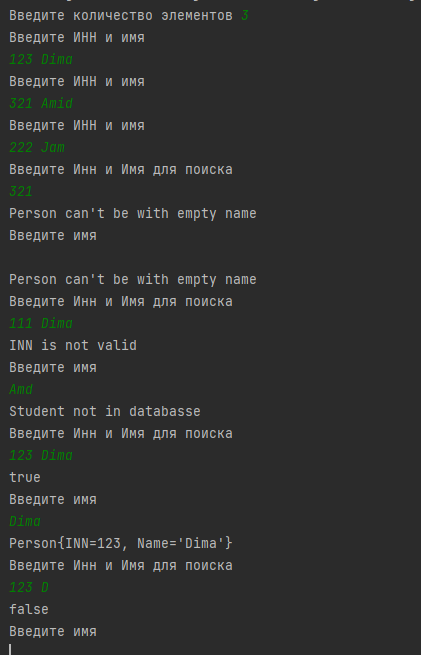


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы изучили исключения в Java. Научились их использовать создавать собственные, обрабатывать и т.д. Также реализовали программу, где мы их прописываем и обрабатываем.

# ПРАКТИЧЕСКАЯ РАБОТА № 10 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ. ПОРОЖДАЮЩИЕ ПАТТЕРНЫ: АБСТРАКТНАЯ ФАБРИКА, ФАБРИЧНЫЙ МЕТОД

## Цель работы

Цель данной практической работы – научиться применять порождающие паттерны при разработке программ на Java. В данной практической работе рекомендуется использовать следующие паттерны: Абстрактная фабрика и фабричный метод.

## Теоретическое введение

Понятие паттерна.

Паттерны (или шаблоны) проектирования описывают типичные способы решения часто встречающихся проблем при проектировании программ.

## Постановка задачи

Упражнение 1

Реализовать класс Абстрактная фабрика для комплексных чисел

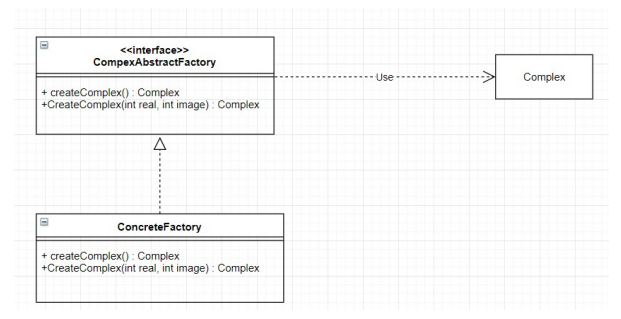


Рис. UML диаграмма задания

Упражнение 2

Реализовать класс Абстрактная фабрика для различных типов стульев: Викторианский стул, Многофункциональный стул, Магический стул, а также интерфейс Стул, от которого наследуются все классы стульев, и класс Клиент, который использует интерфейс стул в своем методе Sit (Chair chair).

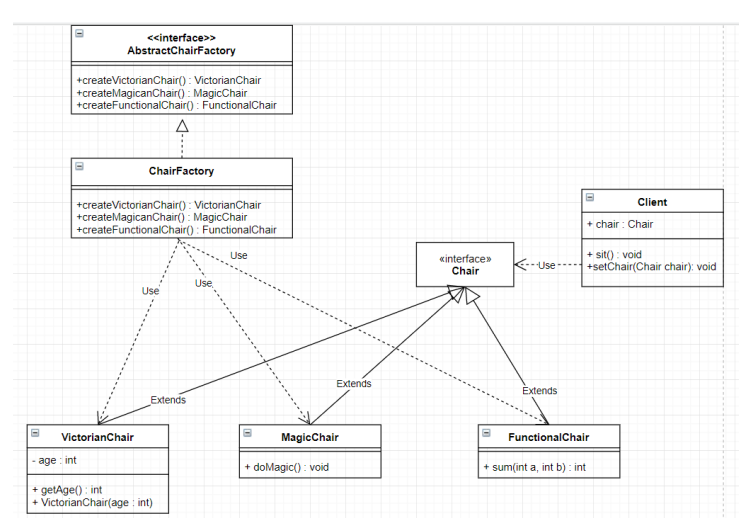
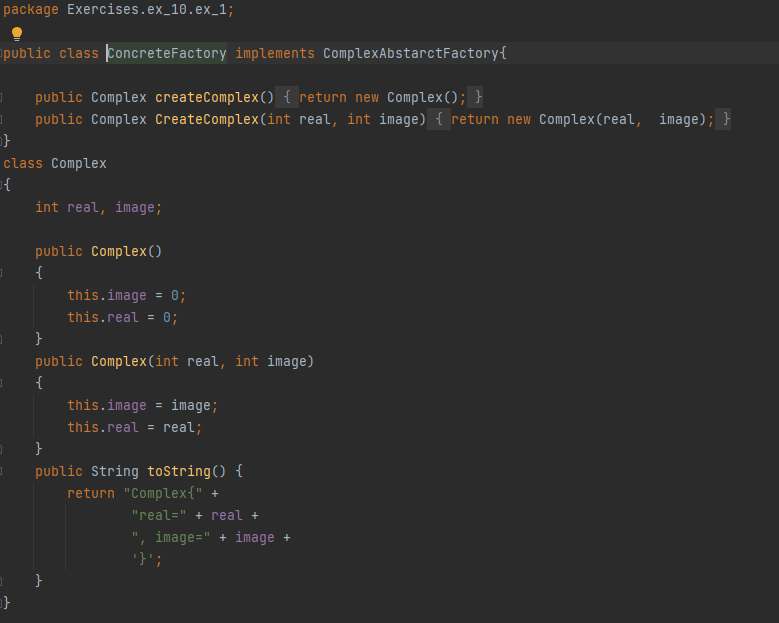
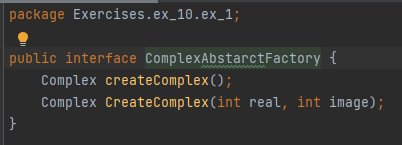
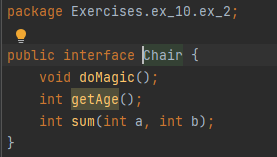
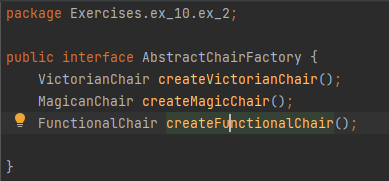
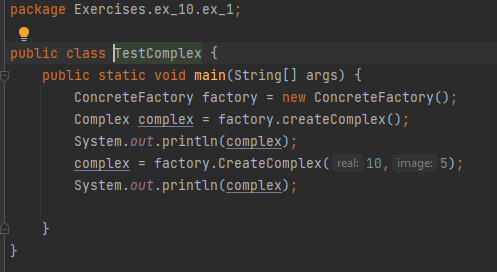
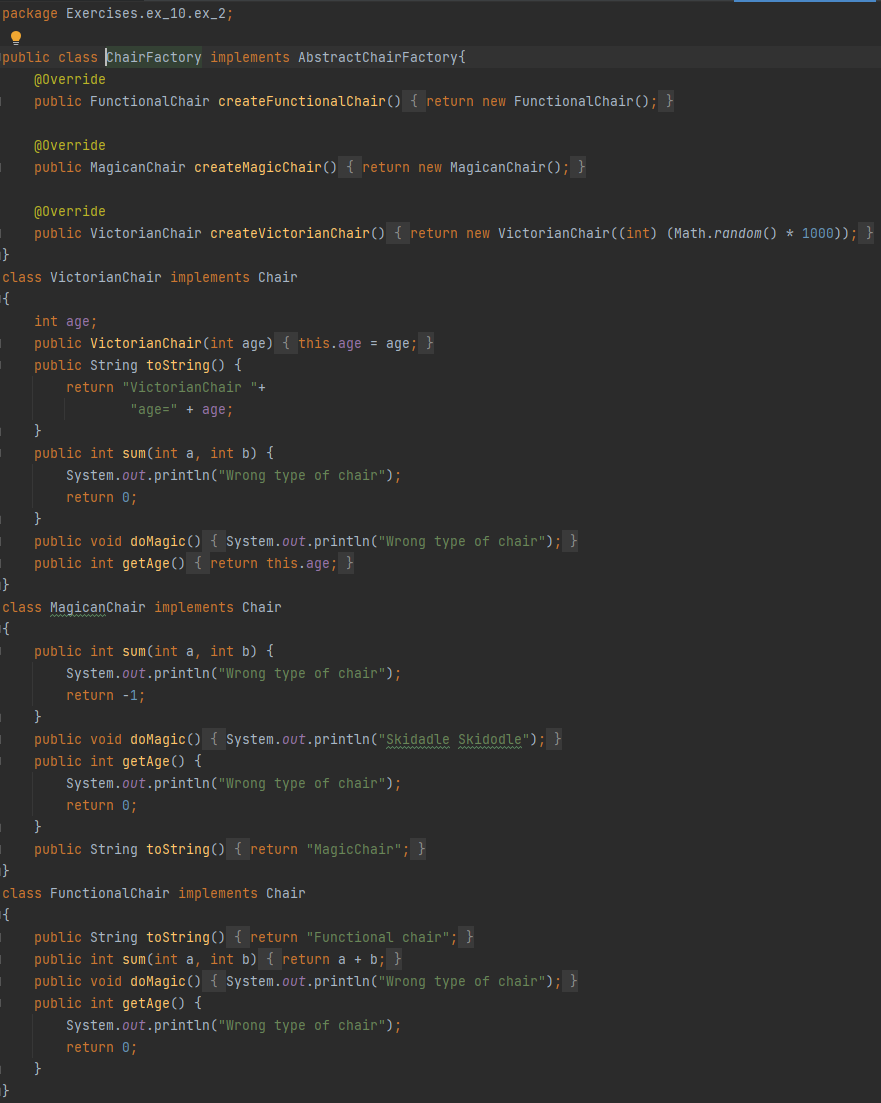


Рис. UML диаграмма задания

## Программный код







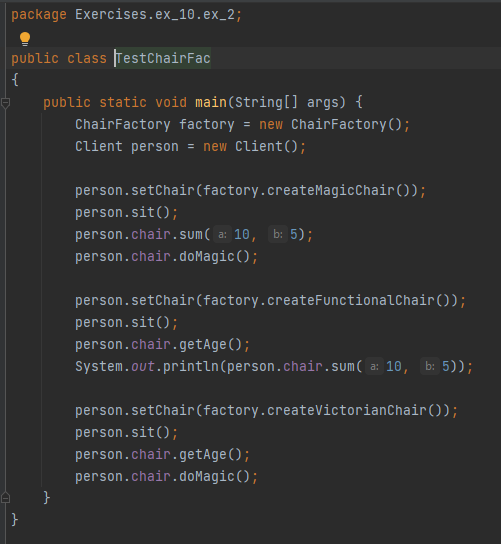
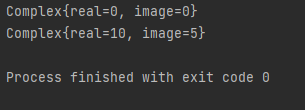


Рис. Код программы

## Вывод программы



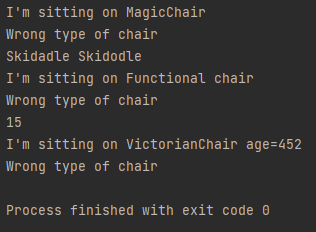
\

Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы изучили метод паттерного программирования абстрактная фабрика. Также научились реализовывать и работать с ним.

# ПРАКТИЧЕСКАЯ РАБОТА №11, 12 ОБРАБОТКА СТРОК.

## Цель работы

Цель данной лабораторной работы – закрепить знания в области обработки строк, научиться применять методы класса String и других классов для обработки строк.

## Теоретическое введение

Особенности использования строк в Java В Java строки представляют собой неизменяемую последовательность символов Unicode. В отличие от представления в C / C ++, где строка является просто массивом типа char, любая Java, строка является объектом класса java.lang.

Однако Java строка, представляет собой в отличие от других используемых классов особый класс, который обладает довольно специфичными характеристиками. Отличия класса строк от обычных классов:

• строка в Java представляет из себя строку литералов (текст), помещенных в двойные кавычки, например: "Hello , World! ". Вы можете присвоить последовательность строковых литералов непосредственно переменной типа String, вместо того чтобы вызывать конструктор для создания экземпляра класса String.

• Оператор '+' является перегруженным, для объектов типа String, и всегда используется, чтобы объединить две строки операндов. В данном контексте мы говорим об операции конкатенации или сложения строк. Хотя '+' не работает как оператор сложения для любых других объектов, кроме строк, например, таких как Point и Circle.

• Строка является неизменяемой, то есть, символьной константой. Это значит, что ее содержание не может быть изменено после ее (строки как объекта) создания. Например, метод toUpperCase () – преобразования к верхнему регистру создает и возвращает новую строку вместо изменения содержания существующей строки.

Обратитесь к API JDK для того чтобы ознакомиться с полным списком возможностей класса String в java.lang.String.

Статический метод String.format()

Статический метод String.format() (введен в JDK 1.5) может быть использован для получения форматированного вывода, таким же образом как это делается в языке Си с использованием функции printf() и спецификаторов вывода для различных типов данных. Метод format() делает то же самое, что функция printf().

Удобно использовать String.format(), если вам нужно получить простую отформатированную строку для некоторых целей (например, для использования в методе ToString()). Для сложных строк, нужно использовать StringBuffer / StringBuilder с Formatter . Если вам просто нужно отправить простую отформатированную строку на консоль, то просто восльзуйтесь методом System.out.printf().

Особенности класса String Строки получили специальное значение в Java, потому что они часто используются в любой программе. Следовательно, эффективность работы с ними (с точки зрения вычислений и хранения) имеет решающее значение. Что нужно помнить про класс String

• это immutable (неизменный) класс

• это final класс

Разработчики Java все-таки решили сохранить примитивные типы в объектно-ориентированном языке вместо того, чтобы сделать вообще все в виде объектов. Нужно сказать, что сделано это в первую очередь, для того чтобы повысить производительность языка. Ведь примитивы хранятся в стеке вызовов, и следовательно, требуют меньше пространства для хранения, и ими легче управлять. С другой стороны, объекты хранятся в области памяти, которую используют программы, и которая называется “куча” (heap), а этот механизм требуют сложного управления памятью и потребляет гораздо больше места для хранения.

По соображениям производительности, класс String в Java разработан, так, чтобы быть чем-то промежуточным между примитивными типами данных и типами данных типа класс. Как уже было отмечено выше специальные характеристики типа String включают в себя:

• '+' оператор, который выполняет сложение примитивных типов данных (таких, как int и double), и перегружен, чтобы работать на объектах String. Операция '+’ выполняет конкатенацию двух операндов типа String.

• Java не поддерживает механизма перегрузки операций по разработке программного обеспечения. В языке, который поддерживает перегрузку операций, например C ++, вы можете превратить оператор '+' (с помощью перегрузки) в оператор для выполнения сложения или вообще вычитания, например двух матриц, кстати это будет примером плохого кода. В Java оператор '+' является единственным оператором, который внутренне перегружен, чтобы поддержать конкатенацию (сложение) строк в Java. Нужно принять к сведению, что '+' не работает на любых других произвольных объектах, помимо строк, например, таких как рассмотренные нами ранее классы Point или Circle.

Теперь, собственно, о строках. Существуют несколько способов создания строк. Строка String может быть получена одним из способов:

• непосредственно из присвоения строкового литерала ссылке типа String – таким же способом как примитивные типы данных;

• или с помощью оператора new и конструктора класса String, аналогично вызову конструктора любого другого класса. Тем не менее, этот способ не часто используется и использовать его не рекомендуется.4

Строковые литералы и объекты типа String

Как уже упоминалось, есть два способа создания строк: неявное создание путем присвоения строкового литерала переменной или явного создания объекта String, через вызов оператора new и вызов конструктора. У класса String есть еще одна особенность. Все строковые литералы, определенные в Java коде, вроде "asdf", на этапе компиляции кэшируются и добавляются в так называемый пул строк.

Java предоставляет специальный механизм для хранения последовательностей символьных литералов (строк), так называемый общий пул строк. Если две последовательности литералов (строки) имеют одинаковое содержание, то они разделяют общее пространство для хранения внутри общего пула. Такой подход принят для того, чтобы сохранить место для хранения часто используемых строк. С другой стороны, объекты типа String (строки), созданные с помощью оператора new и конструктора хранятся в куче.

Каждый объект String в куче имеет свое собственное место для хранения, как и любой другой объект. Там нет обмена хранения в куче, даже если два объекта Строковые имеют то же содержание.

А в куче нет разделяемого пространства для хранения двух объектов, даже если эти два объекта являются объектами типа String и имеют одинаковое содержание.

Вы можете использовать метод equals() класса String для сравнения содержимого двух строк. Вы можете использовать оператор сравнения на равенство '==', чтобы сравнить ссылки ( или указатели ) двух объектов.

String является неизменяемыми

С тех самых пор, когда в языке Java появились возможности по использованию разделяемого пространства для хранения строк с одинаковым содержанием в виде строкового пула, String в Java стали неизменяемыми. То есть, как только строка создается (как объект в памяти программы), ее содержание не может быть изменено никаким образом (по аналогии с Си – строковые литералы — это символьные константы). В противном случае, если этого не сделать, другие ссылки String разделяющие ту же самую ячейку памяти будут зависеть от изменений, которые могут быть непредсказуемыми и, следовательно, является нежелательными. Такой метод, как например, toUpperCase () казалось-бы может изменить содержимое объекта String. Хотя на самом деле, создается совершенно новый объект String и возвращается как раз он в точку вызова. Исходный объект-строка будет впоследствии удален сборщиком мусора (Garbage-collected), как только не окажется больше ссылок, которые ссылаются на него.

Вот поэтому-то объект типа String и считается неизменяемым объектом, вследствие этого, считается не эффективным использовать тип String, например, в том случае, если вам нужно часто модифицировать строку (так вы в таком случае будете создавать много новых объектов типа String, которые каждый раз будут занимать новые места для хранения).

Замечание.

Если содержимое строки должно часто меняться в вашей программе, используйте классы StringBuffer или StringBuilder вместо класса String.

Классы StringBuffer и StringBuilder

Классы StringBuffer и StringBuilder в Java используются, когда возникает необходимость сделать много изменений в строке символов.В отличие от String, объекты типа StringBuffer и StringBuilder могут быть изменены снова и снова. В Java StringBuilder был введен начиная с Java 5.

Как объяснялось выше, строки String являются неизменяемыми, поэтому строковые литералы с таким контентом хранятся в пуле строк. Изменение содержимого одной строки непосредственно может вызвать нежелательные побочные эффекты и может повлиять на другие строки, использующие ту же память.

JDK предоставляет два класса для поддержки возможностей по изменению строк: это классы StringBuffer и StringBuilder (входят в основной пакет java.lang ).

Объекты StringBuffer или StringBuilder так же, как и любые другие обычные объекты, которые хранятся в куче, а не совместно в общем пуле, и, следовательно, могут быть изменены, не вызывая нехороших побочных эффектов на другие объекты.

Класс StringBuilder как класс был введён в JDK 1.5. Это то же самое, как использование класса StringBuffer, за исключением того, что StringBuilder не синхронизирован по многопоточным операциям. Тем не менее, для программы в виде одного потока или нити управления, использование класса StringBuilder, без накладных расходов на синхронизацию, является более эффективным. Использование StringBuffer

Класс находится в пакете java.lang. Прочитайте спецификацию API JDK для использования java.lang.StringBuffer.

Обратите внимание, что объект класса StringBuffer является обычным объектом в прямом понимании этого слова. Вам нужно будет использовать конструктор для создания объектов типа класс StringBuffer (вместо назначения в строку буквальном). Кроме того, оператор '+' не применяется к объектам, в том числе и к объектам StringBuffer. Вы должны будете использовать такой метод, как append() или insert() чтобы манипулировать StringBuffer.

Чтобы создать строку из частей, более эффективно использовать класс StringBuffer (для многопоточных программ) или StringBuilder (для однопоточных), вместо конкатенации строк.

Использование класса StringBuilder

Класс находится в пакете java.lang. Прочитайте спецификацию API JDK для использования. java.lang.StringBuilder

Программа ниже демонстрирует разные способы инвертирования длинных строк. Сравниваются три способа работы со строками: как с объектами класса String, так и с помощью StringBuffer и StringBuilder с использованием метода reverse(). Для измерения времени выполнения различных участков кода в примере используется метод public static native long nanoTime();

Этот метод возвращает текущее значение наиболее точное время системных часов (таймера), в наносекундах. Его можно использовать только для измерения затраченного времени на выполнение операций, и он никак не связан с системным временем и текущим мировым временем.

Возвращаемое методом значение представлено в виде наносекунд, с момента фиксации, на любой произвольный момент времени.

## Постановка задачи

Задание 1

Разработать класс Person, в котором имеется функция, возвращающая Фамилию И.О. Функция должна учитывать возможность отсутствия значений в полях Имя и Отчество. Программу оптимизировать с точки зрения быстродействия.

Задание 2

Доработать класс адреса, который из полученной строки формата "Страна[d] Регион[d] Город[d] Улица[d] Дом[d] Корпус[d] Квартира" ([d] – разделитель, например, «запятая») выбирает соответствующие части и записывает их в соответствующие поля класса Address. Учесть, что в начале и конце разобранной части адреса не должно быть пробелов. Все поля адреса строковые. Разработать проверочный класс не менее чем на четыре адресных строки. В программе предусмотреть две реализации этого метода: а) разделитель – только запятая (использовать метод split()); Внимание, при разработке нужно учесть, что б) разделитель – любой из символов ,.;- (класс StringTokenizer).

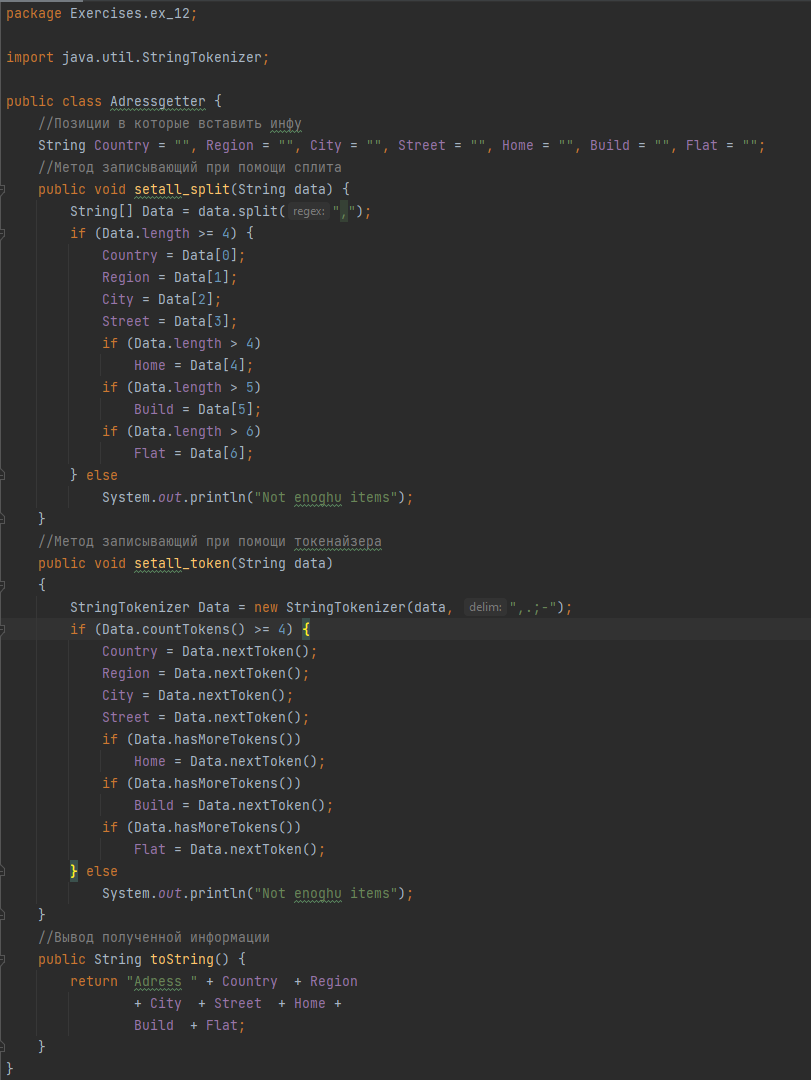
Задание 3

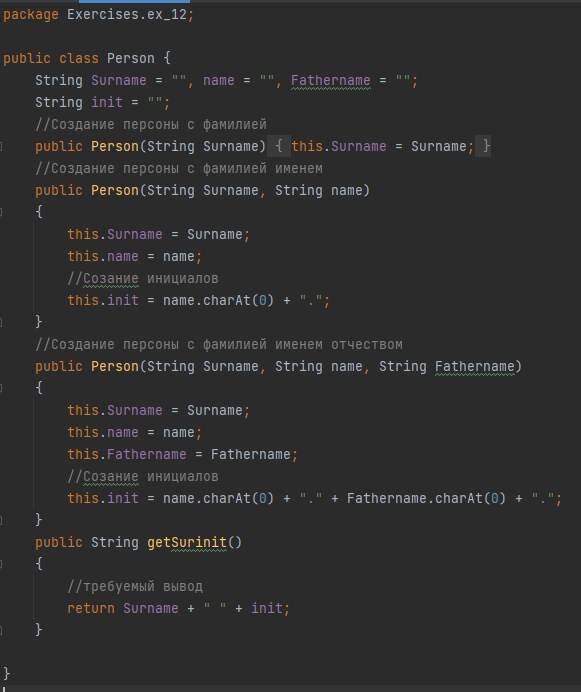
Реализуйте класс Shirt: Метод toString() выводит объяснение и значение полей построчно. Дан также строковый массив: shirts[0] = "S001,Black Polo Shirt,Black,XL"; shirts[1] = "S002,Black Polo Shirt,Black,L"; shirts[2] = "S003,Blue Polo Shirt,Blue,XL"; shirts[3] = "S004,Blue Polo Shirt,Blue,M"; shirts[4] = "S005,Tan Polo Shirt,Tan,XL"; shirts[5] = "S006,Black T-Shirt,Black,XL"; shirts[6] = "S007,White TShirt,White,XL"; shirts[7] = "S008,White T-Shirt,White,L"; shirts[8] = "S009,Green T-Shirt,Green,S"; shirts[9] = "S010,Orange T-Shirt,Orange,S"; shirts[10] = "S011,Maroon Polo Shirt,Maroon,S"; Преобразуйте строковый массив в массив класса Shirt и выведите его на консоль.

Задание 4

Разработайте класс, который получает строковое представление телефонного номера в одном из двух возможных строковых форматов: +, например “+79175655655” или “+104289652211” или 8 для России, например “89175655655” и преобразует полученную строку в формат: +–– Задание 4. (30%) В методе main считай с консоли имя файла, который содержит слова, разделенные пробелом. В методе getLine() используя StringBuilder расставьте все слова в таком порядке, чтобы последняя буква данного слова совпадала с первой буквой следующего не учитывая регистр. Каждое слово должно участвовать 1 раз

## Программный код





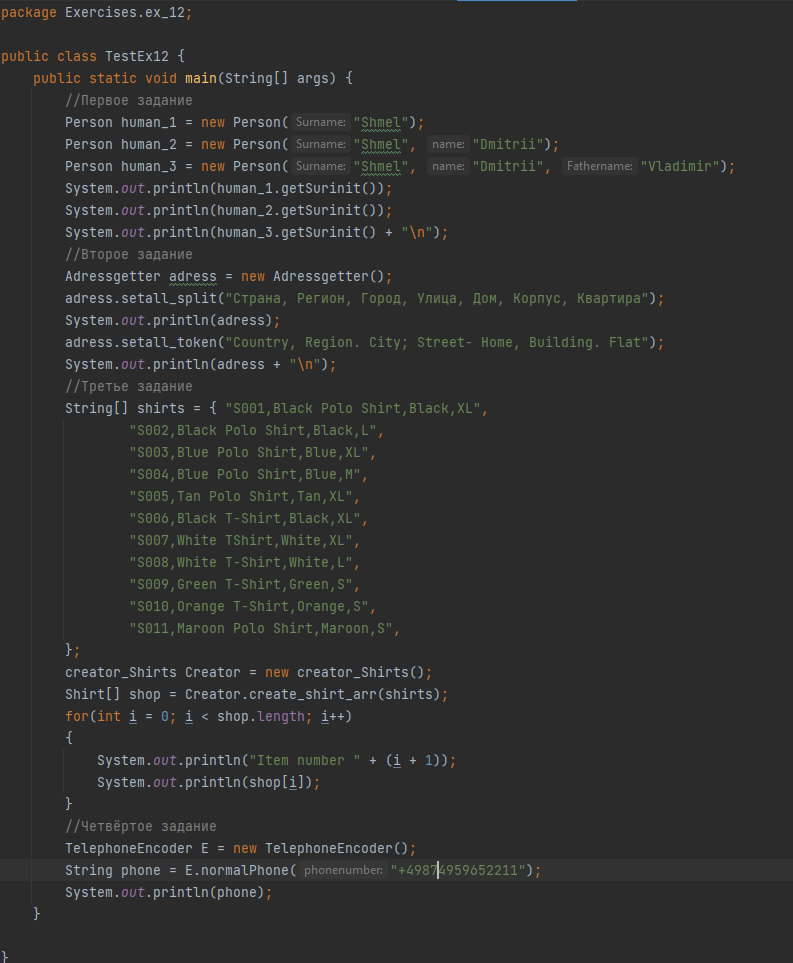
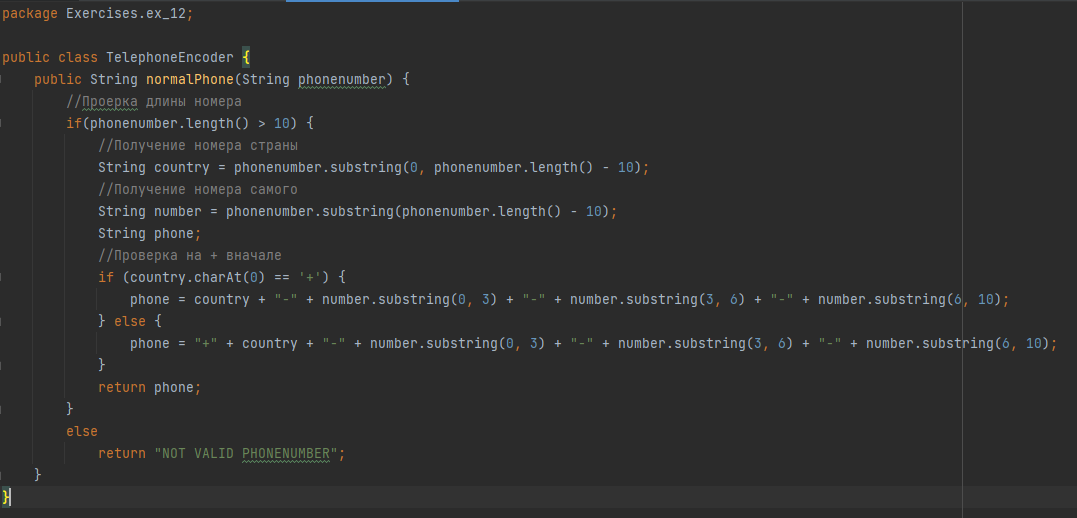
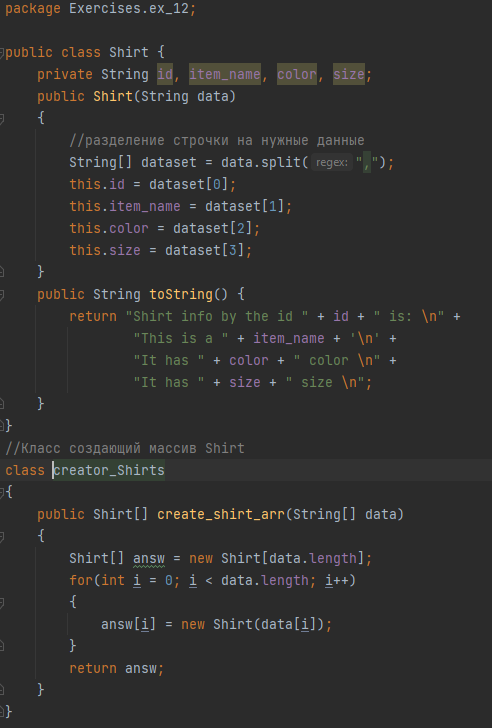


Рис. Код программы

## Вывод программы

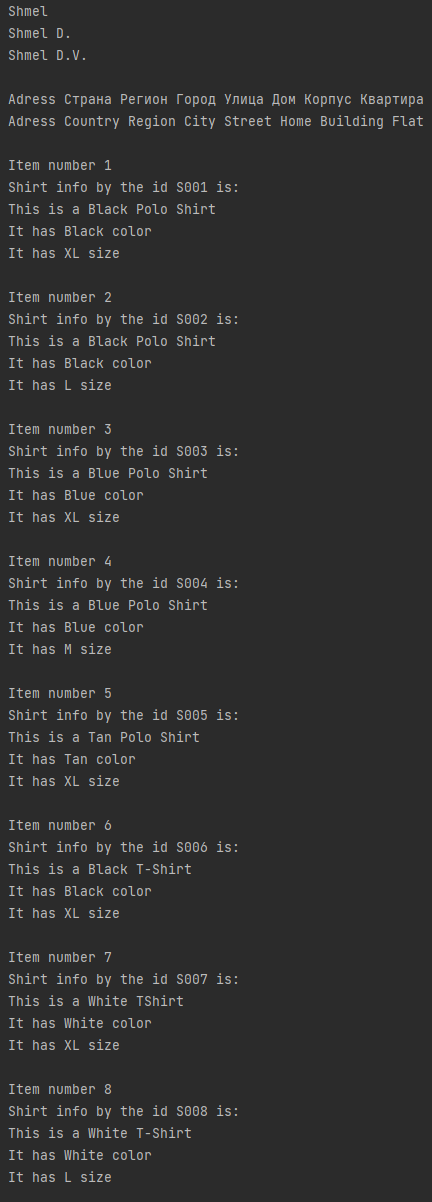




Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы научились работать со строками в Java. Научились их считывать, обрабатывать, изменять и строить при помощи StringBuilder, StringBuffer. Также научились использовать StringTokenizer, для разбития строк.

# ПРАКТИЧЕСКАЯ РАБОТА № 13, 14 РАБОТА С ДАТОЙ И ВРЕМЕНЕМ

## Цель работы

Цель данной практической работы –научиться работать с датами и временем, применять методы класса Date и Calenadar, других классов для обработки строк.

## Теоретическое введение

В Java есть много классов, доступных для работы с датой/временем.

Класс Date

Класс Date изначально предоставлял набор функций для работы с датой - для получения текущего года, месяца и т.д. однако сейчас все эти методы не рекомендованы к использованию и практически всю функциональность для этого предоставляет класс Calendar. Класс Date так же определен в пакете java.sql поэтому желательно указывать полностью квалифицированное имя класса Date.

Существует несколько конструкторов класса Date однако рекомендовано к использованию два Date() и Date(long date) второй конструктор использует в качестве параметра значение типа long который указывает на количество миллисекунд прошедшее с 1 Января 1970, 00:00:00 по Гринвичу. Первый конструктор создает дату использует текущее время и дату (т.е. время выполнения конструктора). Фактически это эквивалентно второму варианту new Date(System.currentTimeMillis); Можно уже после создания экземпляра класса Date использовать метод setTime(long time), для того, что бы задать текущее время.

Для сравнения дат служат методы after(Date date), before(Date date) которые возвращают булевское значение в зависимости от того выполнено условие или нет. Метод compareTo(Date anotherDate) возвращает значение типа int которое равно -1 если дата меньше сравниваемой, 1 если больше и 0 если даты равны. Метод toString() представляет строковое представление даты, однако для форматирования даты в виде строк рекомендуется пользоваться классом SimpleDateFormat определенном в пакте java.text

Классы Calendar и GregorianCalendar

Более развитые средства для работы с датами представляет класс Calendar. Calendar является абстрактным классом. Для различных платформ реализуются конкретные подклассы календаря. На данный момент существует реализация Грегорианского календаря - GregorianCalendar. Экземпляр этого класса получается вызовом статического метода getInstance(), который возвращает экземпляр класса Gregorian. Подклассы класса Calendar должны интерпретировать объект Date по-разному. В будущем предполагается реализовать так же лунный календарь, используемый в некоторых странах. Calendar обеспечивает набор методов позволяющих манипулировать различными "частями" даты, т.е. получать и устанавливать дни, месяцы, недели и т.д. Если при задании параметров календаря упущены некоторые параметры, то для них будут использованы значения по умолчанию для начала отсчета. т.е.YEAR = 1970, MONTH = JANUARY, DATE = 1 и т.д.

Для считывания, установки манипуляции различных "частей" даты используются методы get(int filed), set(int filed, int value), add(int field, int amount), roll(int field, inr amount), переменная типа int с именем filed указывает на номер поля с которым нужно произвести операцию. Для удобства все эти поля определены в Calendar, как статические константы типа int. Рассмотрим подробнее порядок выполнения перечисленных методов.

Метод set(int field,int value)

Как уже отмечалось ранее данный метод производит установку какого - либо поля даты. На самом деле после вызова этого метода, немедленного пересчета даты не производится. Пересчет даты будет осуществлен только после вызова методов get(), getTime() или TimeInMillis(). Т.о. последовательная установка нескольких полей, не вызовет не нужных вычислений. Помимо этого, появляется еще один интересный эффект. Рассмотрим следующий пример. Предположим, что дата установлена на последний день августа. Необходимо перевести ее на последний день сентября. Если внутреннее представление даты изменялось бы после вызова метода set, то при последовательной установке полей мы получили бы вот такой эффект.

Вывод

Класса Date будет достаточно, если вам просто нужна простая отметка времени. Вы можете использовать SimpleDateFormat для управления форматом отображения даты /времени.

Используйте класс java.util.Calendar, если вам нужно извлечь год, месяц, день, час, минуту и секунду или манипулировать этими полями (например, 7 дней спустя, 3 недели назад). Используйте java.text.DateFormat для форматирования даты (от даты до текста) и разбора строки даты (от текста к дате). SimpleDateForamt является подклассом DateFormat.

Date является устаревшим классом, который не поддерживает интернационализацию. Calendar и DateFormat поддерживают локалзацию (вам нужно учитывать локализацию только в том случае, если ваша программа будет работать во нескольких странах одновременно).

Измерение времени

Любые приложения (такие как игры и анимация) требуют хорошего контроля времени. Java предоставляет эти статические методы в классе System. Метод System.currentTimeMillis() возвращает текущее время в миллисекундах с 1 января 1970 г. 00:00:00 по Гринвичу (известное как «эпоха») в длинном формате.

Метод System.nanoTime (): возвращает текущее значение наиболее точного доступного системного таймера, в наносекундах, в течение длительного времени. Введенный с JDK 1.5. метод nanoTime () предназначен для измерения относительного временного интервала вместо предоставления абсолютного времени.

## Постановка задачи

Задание 1

Написать программу, выводящую фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Для получения последней даты и времени использовать класс Date из пакета java.util.\* (Объявление Dated=newDate() или метод System.currentTimeMillis().

Задание 2

Приложение, сравнивающее текущую дату и дату, введенную пользователем c текущим системным временем

Задание 3

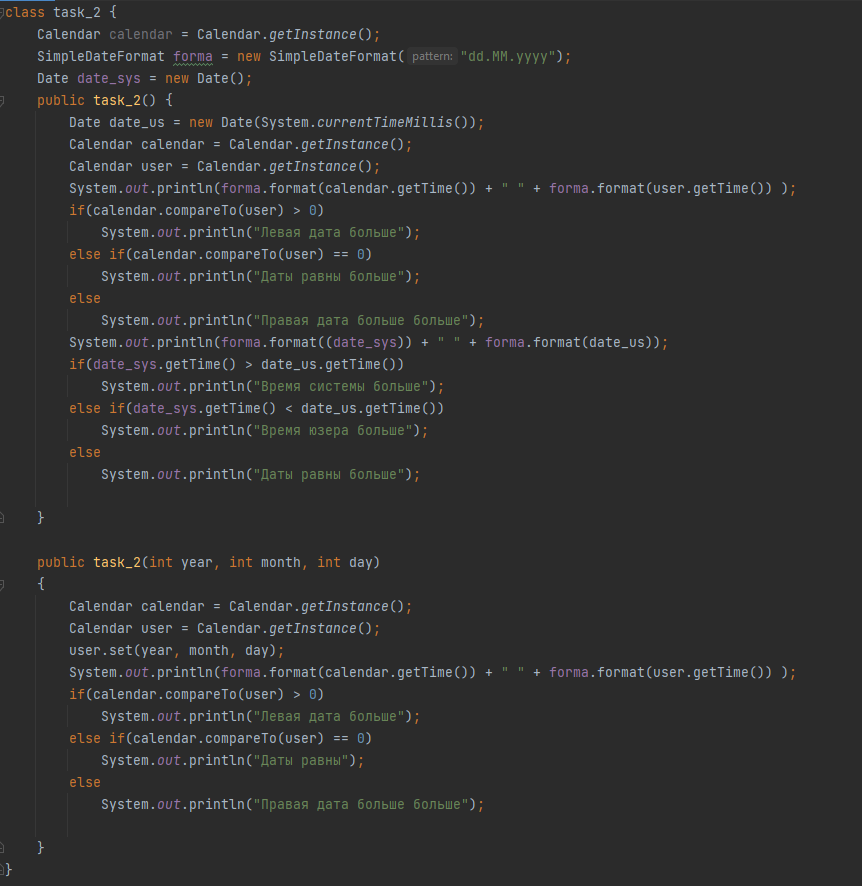
Доработайте класс Student предусмотрите поле для хранения даты рождения, перепишите метод toString() таким образом, чтобы он разработайте метод, возвращал строковое представление даты рождения по вводимому в метод формату даты (например, короткий, средний и полный формат даты).

Задание 4

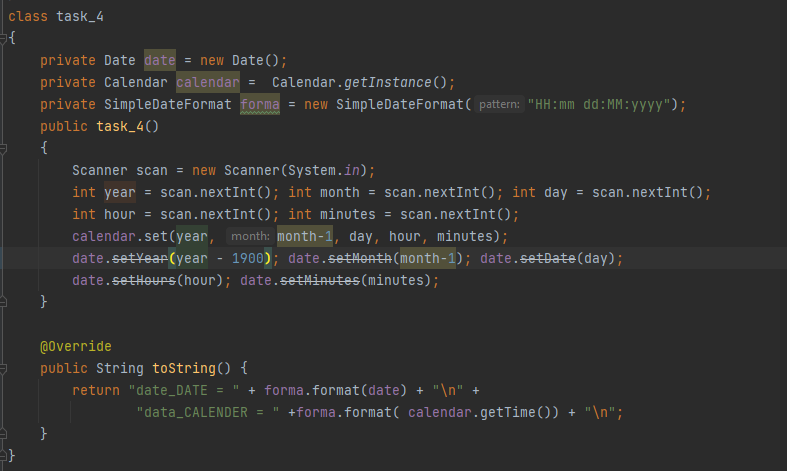
Напишите пользовательский код, который формирует объекты Date и Calendar по следующим данным, вводимым пользователем: Задание 5 (30%) Сравнить время выполнения кода в реализации кода в виде различных структур данных из предыдущих заданий (сравнить ArrayList и LinkedList по производительности – операции вставки, удаления, добавления и поиска по образцу)

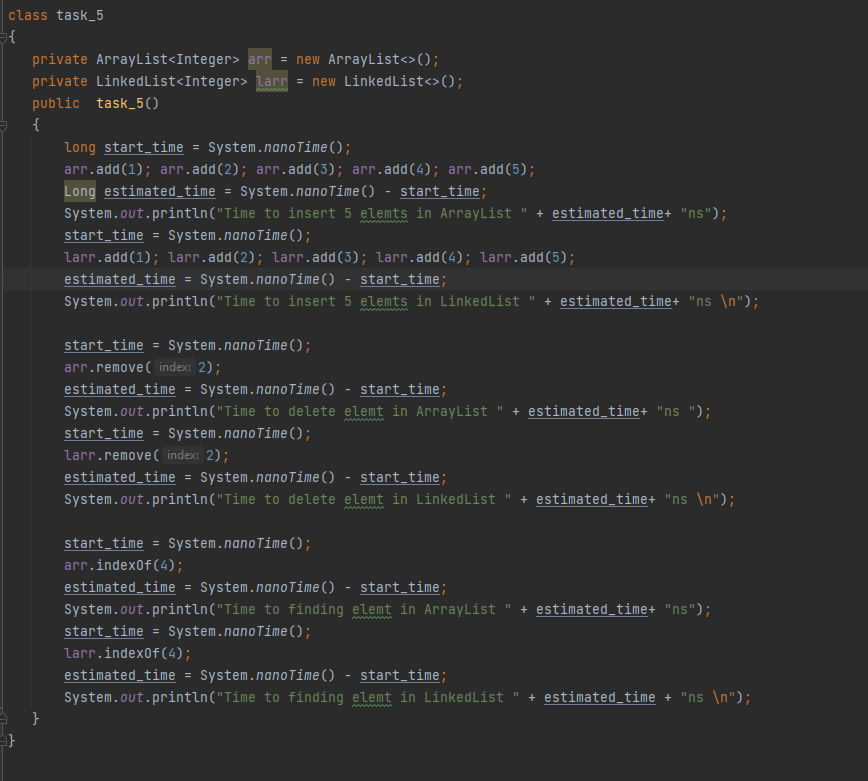
## Программный код











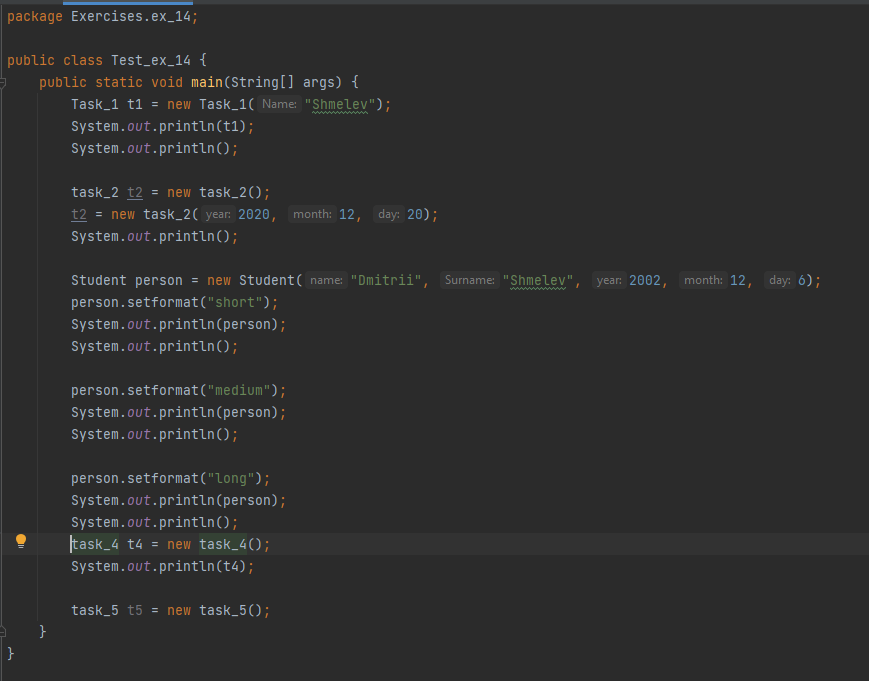


Рис. Код программы

## Вывод программы

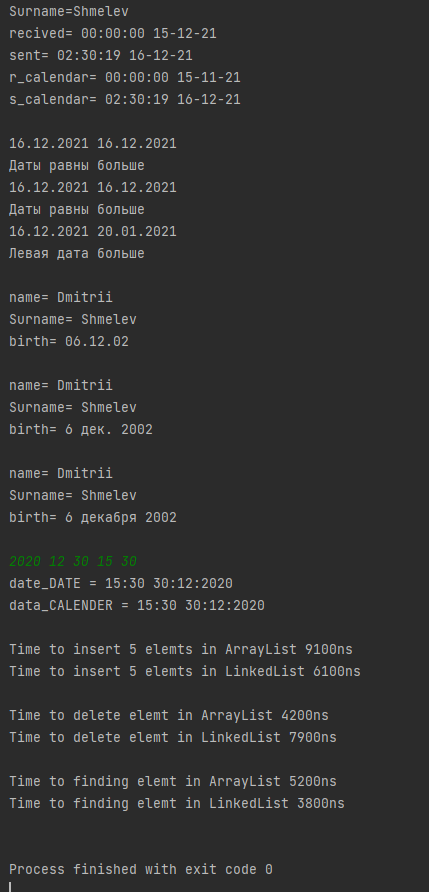


Рис. Вывод программы

## Вывод

В результате выполнения данной практической работы мы научились работать с классами Date и Calendar в Java. Получать нынешнею дату, создавать даты для других классов, сравнивать дату и сравнивать быстроту работы кода. Также мы научились форматировать дату для определенного её вывода.

# ПРАКТИЧЕСКАЯ РАБОТА № 15, 16 ИТОГОВАЯ РАБОТА

## Цель работы

Цель работы:

## Теоретическое введение

## Постановка задачи

## Программный код

## Вывод программы

## Вывод

В результате выполнения данной практической работы

# ЛАБОРАТОРНАЯ РАБОТА №1 ЦИКЛЫ, УСЛОВИЯ, ПЕРЕМЕННЫЕ И МАССИВЫ В JAVA.

## Цель работы

Целями данной лабораторной работы являются получение практических навыков разработки программ, изучение синтаксиса языка Java, освоение основных конструкций языка Java (циклы, условия, создание переменных и массивов, создание методов, вызов методов), а также научиться осуществлять стандартный ввод/вывод данных.

## Постановка задачи

Сгенерировать массив целых чисел случайным образом, вывести его на экран, отсортировать его, и снова вывести на экран.

## Программный код





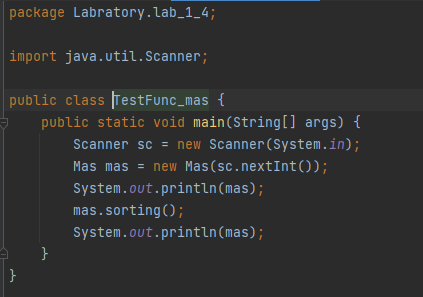


Рис. Код программы

## Вывод программы

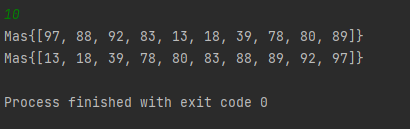


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы узнали о понятиях циклов и переменных в Java. Также научились их реализовывать и использовать в программе.

# ЛАБОРАТОРНАЯ РАБОТА № 2 ООП В JAVA. ПОНЯТИЕ КЛАССА.

## Цель работы

Цель данной лабораторной работы - изучить основные концепции объектно-ориентированного программирования, изучить понятие класса и научиться создавать классы.

## Постановка задачи

Создать класс, описывающий тело человека(Human). Для описания каждой части тела создать отдельные классы(Head, Leg, Hand). Описать необходимые свойства и методы для каждого класса. Протестировать работу класса Human.

## Программный код



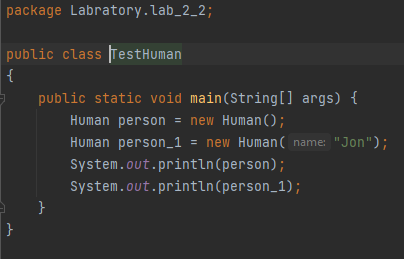


Рис. Код программы

## Вывод программы

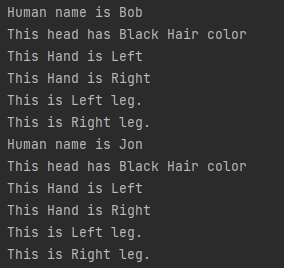


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы познакомились с понятием классов, их конструкторами, методами и как работать с ними в Java. Также научились реализовывать их и выводить информацию о них.

# ЛАБОРАТОРНАЯ РАБОТА № 3 НАСЛЕДОВАНИЕ В JAVA

## Цель работы

Цель данной лабораторной работы - изучить понятие наследования, и научиться реализовывать наследование в Java.

## Постановка задачи

Создать абстрактный класс, описывающий собак(Dog). С помощью наследования реализовать различные породы собак. Протестировать работу классов.

## Программный код



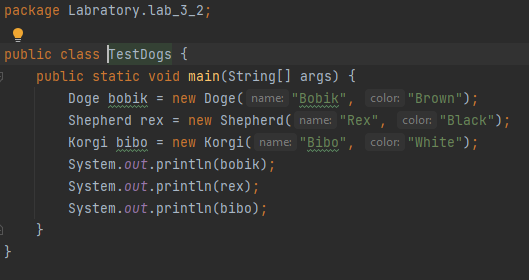


Рис. Код программы

## Вывод программы

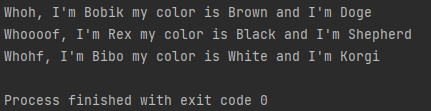


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы познакомились с концептом наследования и абстрактными классами в Java. Научились пользоваться наследованием и создавать и реализовывать абстрактные классы.

# ЛАБОРАТОРНАЯ РАБОТА № 4 ИНТЕРФЕЙСЫ В JAVA.

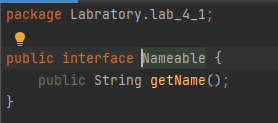
## Цель работы

Цель данной лабораторной работы - изучить понятие интерфейса, научиться создавать интерфейсы в Java и применять их в программах.

## Постановка задачи

Создать интерфейс Nameable, с методом getName(), возвращающим имя объекта, реализующего интерфейс. Проверить работу для различных объектов (например, можно создать классы, описывающие разные сущности, которые могут иметь имя: планеты, машины, животные и т. д.).

## Программный код





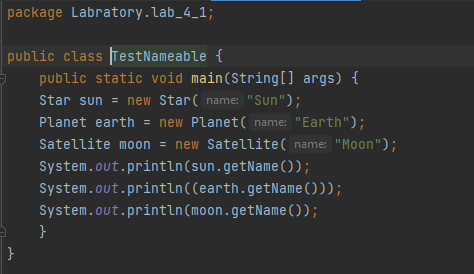


Рис. Код программы

## Вывод программы

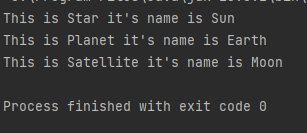


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы познакомились с интерфейсами в Java. Узнали как они работают, объявляются и как их использовать. Также научились реализовывать их.

# ЛАБОРАТОРНАЯ РАБОТА № 5 СОЗДАНИЕ ПРОГРАМ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ НА JAVA

## Цель работы

Цель данной лабораторной работы - научиться создавать графический интерфейс пользователя, освоить на практике работу с различными объектами для создания ГИП, менеджерами размещения компонентов.

## Постановка задачи

Создать окно, отобразить в нем картинку, путь к которой указан в аргументах командной строки.

## Программный код



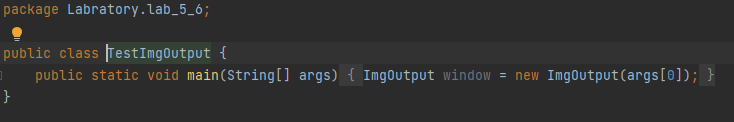


Рис. Код программы

## Вывод программы

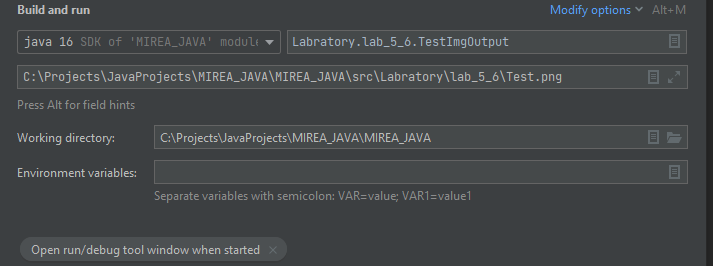


Рис. Вводимые параметры

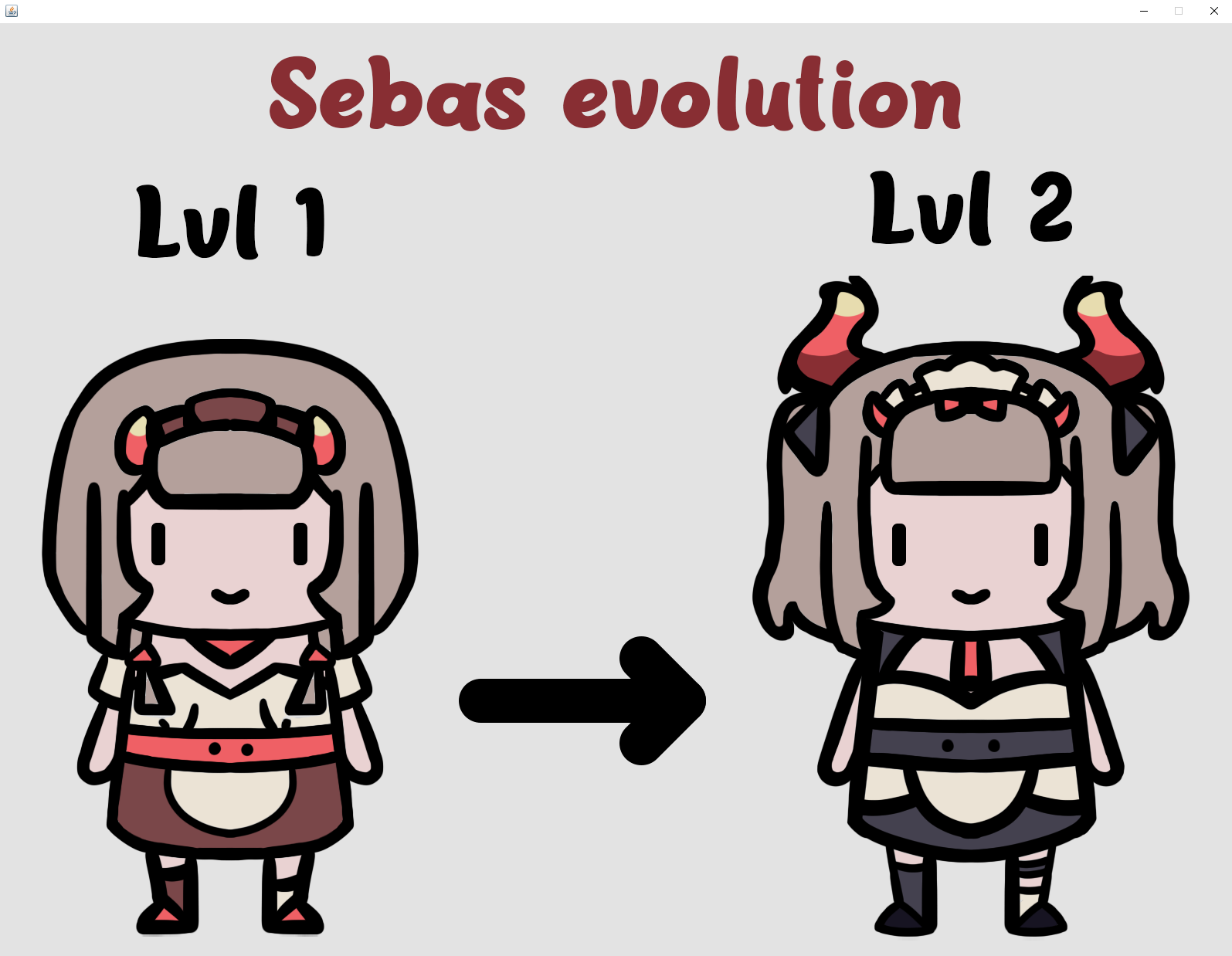


Рис. Окно программы на Java

## Вывод

В результате выполнения данной лабораторной работы мы познакомились с классами и методами для создания меню и окон приложений на Java. Научились использовать их и создавать первые простые окна приложений, которые могут выводить картинки, массив картинок и т.д.

# ЛАБОРАТОРНАЯ РАБОТА № 6 ОБРАБОТКА СОБЫТИЙ В JAVA ПРОГРАММАХ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ

## Цель работы

Цель данной лабораторной работы - научиться обрабатывать различные события для разных компонентов(кнопок, меню и т. д.).

## Теоретическое введение

Механизм обработки событий библиотеки Swing.

В контексте графического интерфейса пользователя наблюдаемыми объектами являются элементы управления: кнопки, флажки, меню и т.д. Они могут сообщить своим наблюдателям об определенных событиях, как элементарных (наведение мышкой, нажатие клавиши на клавиатуре), так и о высокоуровневых (изменение текста в текстовом поле, выбор нового элемента в выпадающем списке и т.д.).

Наблюдателями должны являться объекты классов, поддерживающих специальные интерфейсы (в классе наблюдателя должны быть определенные методы, о которых «знает» наблюдаемый и вызывает их при наступлении события). Такие классы в терминологии Swing называются слушателями. Интерфейс MouseListener и обработка событий от мыши.

События от мыши — один из самых популярных типов событий. Практически любой элемент управления способен сообщить о том, что на него навели мышь, щелкнули по нему и т.д. Об этом будут оповещены все зарегистрированные слушатели событий от мыши.

Слушатель событий от мыши должен реализовать интерфейс MouseListener. В этом интерфейсе перечислены следующие методы:

• public void mouseClicked(MouseEvent event) — выполнен щелчок мышкой на наблюдаемом объекте

• public void mouseEntered(MouseEvent event) — курсор мыши вошел в область наблюдаемого объекта

• public void mouseExited(MouseEvent event) — курсор мыши вышел из области наблюдаемого объекта

• public void mousePressed(MouseEvent event) — кнопка мыши нажата в момент, когда курсор находится над наблюдаемым объектом

• public void mouseReleased(MouseEvent event) — кнопка мыши отпущена в момент, когда курсор находится над наблюдаемым объектом

Чтобы обработать нажатие на кнопку, требуется описать класс, реализующий интерфейс MouseListener. Далее необходимо создать объект этого класса и зарегистрировать его как слушателя интересующей нас кнопки. Для регистрации слушателя используется метод addMouseListener().

Класс MouseAdapter

Программа стала выглядеть загроможденной главным образом из-за того, что помимо полезного для нас метода mouseClicked() пришлось определять пустые реализации всех остальных, не нужных методов. Но этого можно избежать.

Класс MouseAdapter реализует интерфейс MouseListener, определяя пустые реализации для каждого из его методов. Можно унаследовать своего слушателя от этого класса и переопределить те методы, которые нам нужны.

Общая структура слушателей Кроме слушателей MouseListener визуальные компоненты Swing поддерживают целый ряд других слушателей.

Каждый слушатель должен реализовывать интерфейс \*\*\*Listener, где \*\*\* — тип слушателя. Практически каждому из этих интерфейсов (за исключением тех, в которых всего один метод) соответствует пустой класс-заглушка \*\*\*Adapter. Каждый метод интерфейса слушателя принимает один параметр типа \*\*\*Event, в котором собрана вся информация, относящаяся к событию.

Чтобы привязать слушателя к объекту (который поддерживает соответствующий тип слушателей) используется метод add\*\*\*Listener(\*\*\*Listener listener).

Например, слушатель MouseListener должен реализовать интерфейс с таким же именем, которому соответствует классзаглушка MouseAdapter. Методы этого интерфейса обрабатывают параметр типа MouseEvent, а регистрируется слушатель методом addMouseListener(MouseListener listener).

Слушатель фокуса FocusListener

Слушатель FocusListener отслеживает моменты, когда объект получает фокус (то есть становится активным) или теряет его. Концепция фокуса очень важна для оконных приложений. В каждый момент времени в окне может быть только один активный (находящийся в фокусе) объект, который получает информацию о нажатых на клавиатуре клавишах (т.е. реагирует на события клавиатуры), о прокрутке колесика мышки и т.д. Пользователь активирует один из элементов управления нажатием мышки или с помощью клавиши Tab (переключаясь между ними).

Интерфейс FocusListener имеет два метода:

public void focusGained(FocusEvent event) — вызывается, когда наблюдаемый объект получает фокус

public void focusLost(FocusEvent event) — вызывается, когда наблюдаемый объект теряет фокус.

Слушатель колесика мышки MouseWheelListener

Слушатель MouseWheelListener оповещается при вращении колесика мыши в тот момент, когда данный компонент находится в фокусе. Этот интерфейс содержит всего один метод:

public void mouseWheelMoved(MouseWheelEvent event).

Слушатель клавиатуры KeyListener

Слушатель KeyListener оповещается, когда пользователь работает с клавиатурой в тот момент, когда данный компонент находится в фокусе. В интерфейсе определены методы:

public void mouseKeyTyped(KeyEvent event) — вызывается, когда с клавиатуры вводится символ

public void mouseKeyPressed(KeyEvent event) — вызывается, когда нажата клавиша клавиатуры

public void mouseKeyReleased(KeyEvent event) — вызывается, когда отпущена клавиша клавиатуры.

Аргумент event этих методов способен дать весьма ценные сведения. В частности, команда event.getKeyChar() возвращает символ типа char, связанный с нажатой клавишей. Если с нажатой клавишей не связан никакой символ, возвращается константа CHAR\_UNDEFINED. Команда event.getKeyCode() возвратит код нажатой клавиши в виде целого числа типа int. Его можно сравнить с одной из многочисленных констант, определенных в классе KeyEvent: VK\_F1, VK\_SHIFT, VK\_D, VK\_MINUS и т.д. Методы isAltDown(), isControlDown(), isShiftDown() позволяют узнать, не была ли одновременно нажата одна из клавиш-модификаторов Alt, Ctrl или Shift.

Слушатель изменения состояния ChangeListener

Слушатель ChangeListener реагирует на изменение состояния объекта. Каждый элемент управления по своему определяет понятие «изменение состояния». Например, для панели со вкладками JTabbedPane это переход на другую вкладку, для ползунка JSlider — изменение его положения, кнопка JButton рассматривает как смену состояния щелчок на ней. Таким образом, хотя событие это достаточно общее, необходимо уточнять его специфику для каждого конкретного компонента. В интерфейсе определен всего один метод:

public void stateChanged(ChangeEvent event).

Слушатель событий окна WindowListener

Слушатель WindowListener может быть привязан только к окну и оповещается о различных событиях, произошедших с окном:

public void windowOpened(WindowEvent event) — окно открылось.

public void windowClosing(WindowEvent event) — попытка закрытия окна (например, пользователя нажал на крестик). Слово «попытка» означает, что данный метод вызовется до того, как окно будет закрыто и может воспрепятствовать этому (например, вывести диалог типа «Вы уверены?» и отменить закрытие окна, если пользователь выберет «Нет»).

public void windowClosed(WindowEvent event) — окно закрылось.

public void windowIconified(WindowEvent event) — окно свернуто. public void windowDeiconified(WindowEvent event) — окно развернуто.

public void windowActivated(WindowEvent event) — окно стало активным.

public void windowDeactivated(WindowEvent event) — окно стало неактивным.

Слушатель событий компонента СomponentListener

Слушатель ComponentListener оповещается, когда наблюдаемый визуальный компонент изменяет свое положение, размеры или видимость. В интерфейсе четыре метода:

public void componentMoved(ComponentEvent event) — вызывается, когда наблюдаемый компонент перемещается (в результате вызова команды setLocation(), работы менеджера размещения или еще по какой-то причине).

public void componentResized(ComponentEvent event) — вызывается, когда изменяются размеры наблюдаемого компонента.

public void componentHidden(ComponentEvent event) — вызывается, когда компонент становится невидимым.

public void componentShown(ComponentEvent event) — вызывается, когда компонент становится видимым.

Слушатель выбора элемента ItemListener

Слушатель ItemListener реагирует на изменение состояния одного из элементов, входящих в состав наблюдаемого компонента. Например, выпадающий список JComboBox состоит из множества элементов и слушатель реагирует, когда изменяется выбранный элемент. Также данный слушатель оповещается при выборе либо отмене выбора флажка JCheckBox или переключателя JRadioButton, изменении состояния кнопки JToggleButton и т.д. Слушатель обладает одним методом:

public void itemStateChanged(ItemEvent event).

Универсальный слушатель ActionListener Среди многочисленных событий, на которые реагирует каждый элемент управления (и о которых он оповещает соответствующих слушателей, если они к нему присоединены), есть одно основное, вытекающее из самой сути компонента и обрабатываемое значительно чаще, чем другие. Например, для кнопки это щелчок на ней, а для выпадающего списка — выбор нового элемента.

Для отслеживания и обработки такого события может быть использован особый слушатель ActionListener, имеющий один метод:

public void actionPerformed(ActionEvent event).

У использования ActionListener есть небольшое преимущество в эффективности (так, при обработке нажатия на кнопку не надо реагировать на четыре лишних события — ведь даже если методы-обработчики пустые, на вызов этих методов все равно тратятся ресурсы). А кроме того очень удобно запомнить и постоянно использовать один класс с одним методам и обращаться к остальным лишь в тех относительно редких случаях, когда возникнет такая необходимость

## Постановка задачи

Реализация приложения Java, который имеет макет границы и надписи для каждой области в макете. Теперь определим события мыши, чтобы описать действия:

a. Когда мышь входит CENTER программа показывает диалоговое окно (Добро пожаловать в )

b. Когда мышь входит WEST программа показывает диалоговое окно (Добро пожаловать в Джидда)

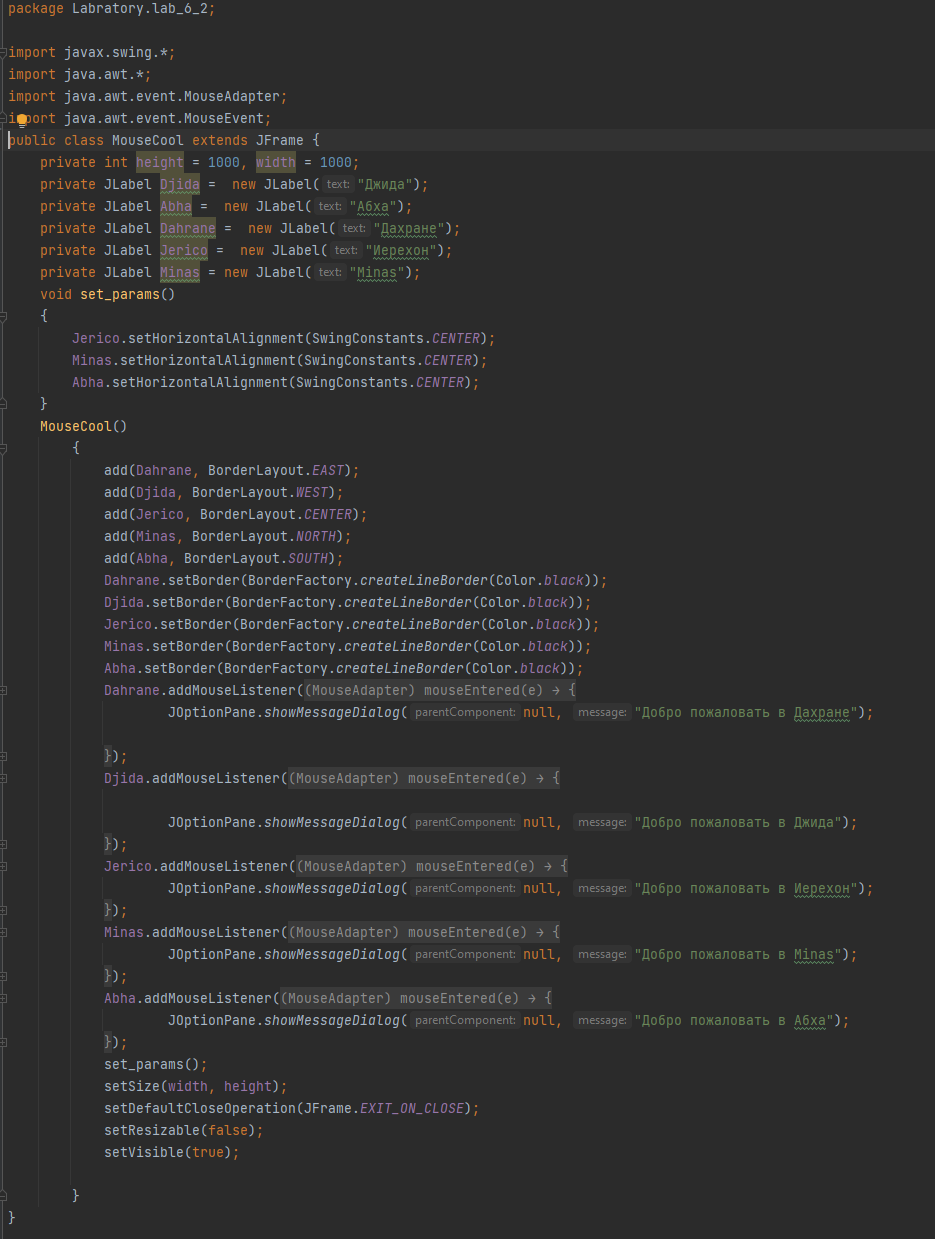
c. Когда мышь входит SOUTH программа показывает диалоговое окно (Добро пожаловать Абха)

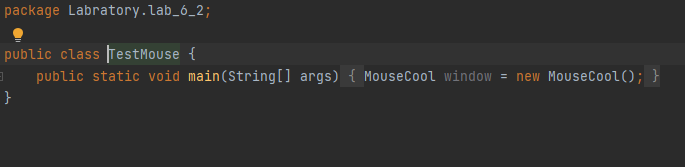
d. Когда мышь входит в NORTH программа показывает диалоговое окно (Добро пожаловать в)

e. Когда мышь входит EAST программа показывает диалоговое окно

(Добро пожаловать в Дахране)

## Программный код



Рис. Код программы

## Вывод программы

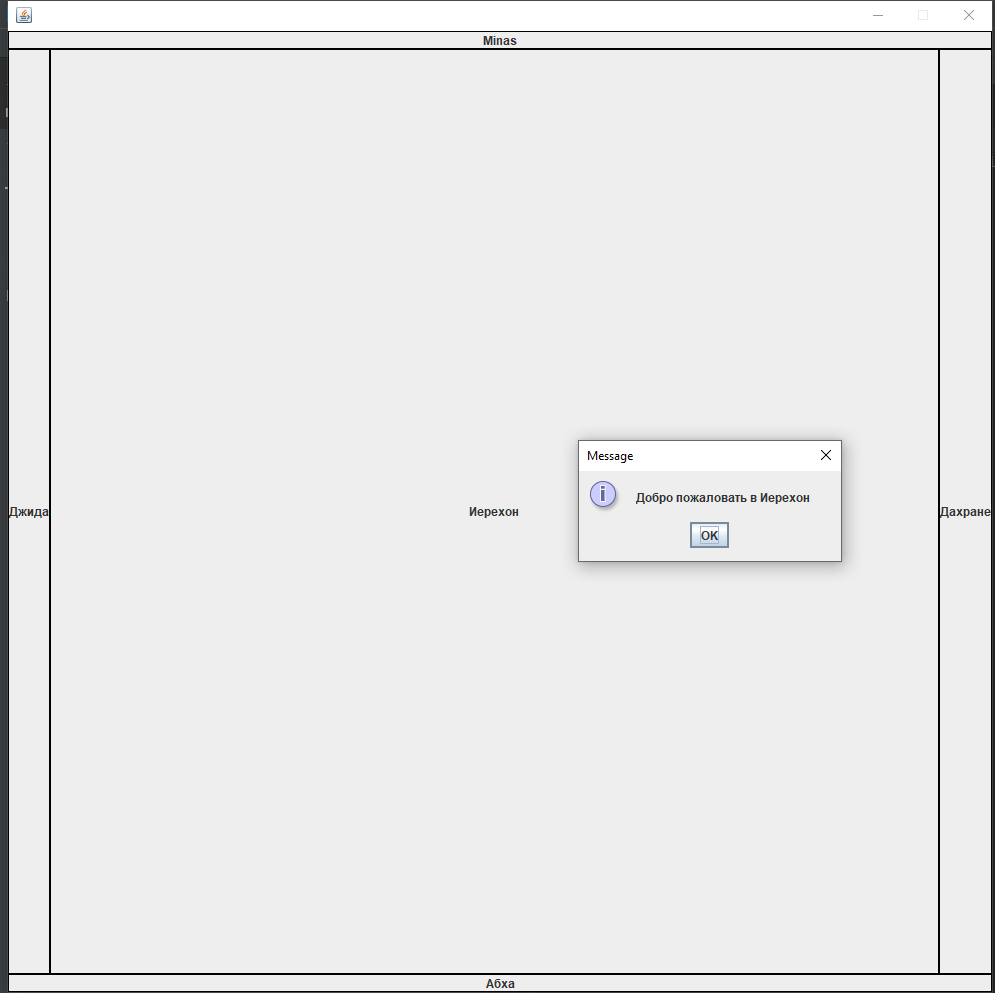
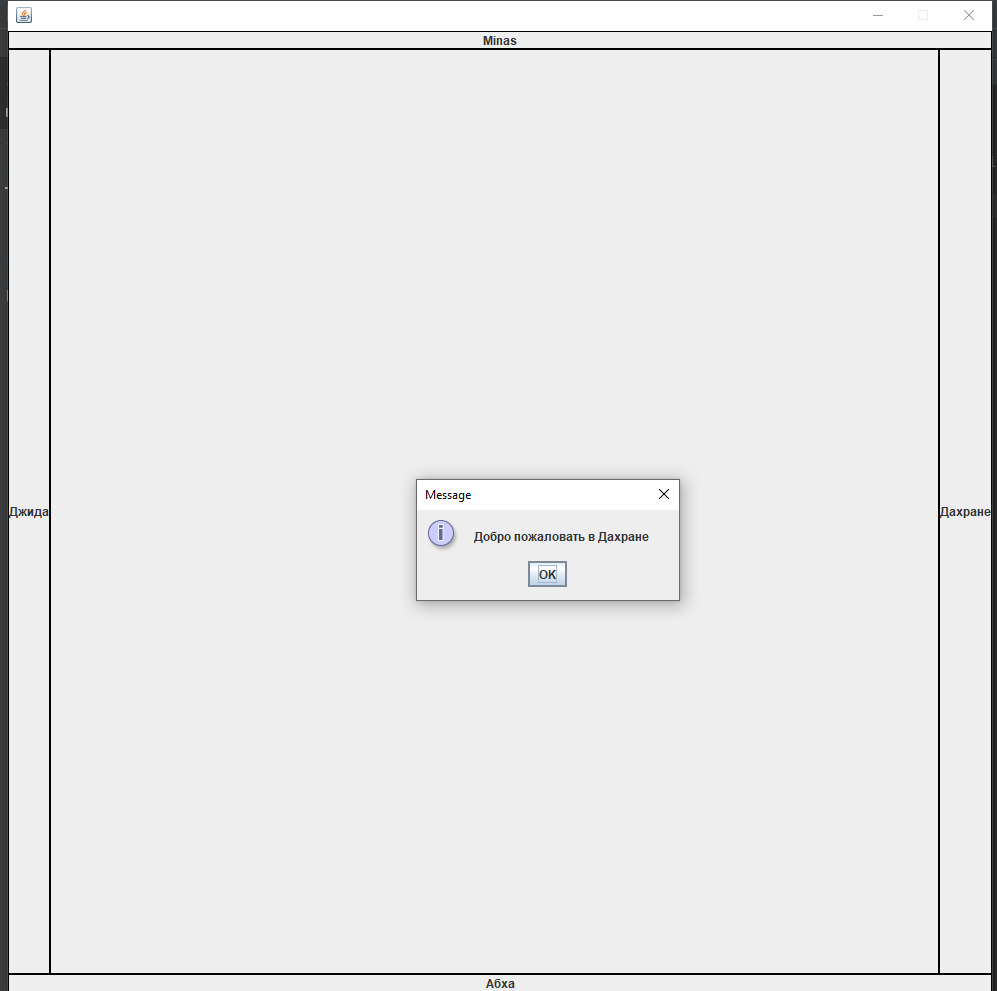
 

Рис. Вывод окна программы

## Вывод

В результате выполнения данной лабораторной работы мы ещё лучше познакомились с методами реализации окна программы на Java, познакомились с обработчиками событий, узнали разницу между ними, научились использовать и реализовывать их.

# ЛАБОРАТОРНАЯ РАБОТА № 7 КОЛЛЕКЦИИ, ОЧЕРЕДИ, СПИСКИ В JAVA

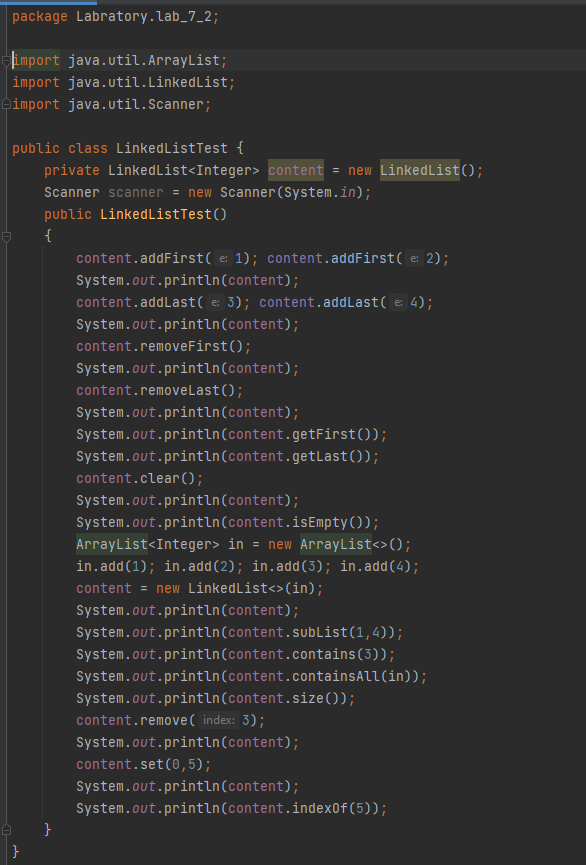
## Цель работы

Целью данной лабораторной работы является изучение работы с различными коллекциями в Java.

## Постановка задачи

Протестировать работу коллекции LinkedList.

## Программный код



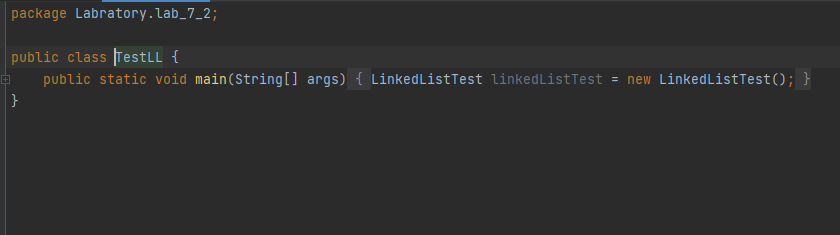


Рис. Код программы

## Вывод программы

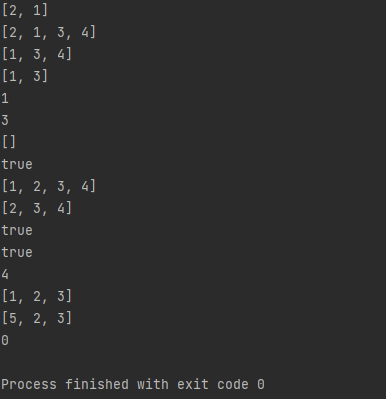


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы узнали о разных коллекциях хранения данных, способов их реализации, работе и особенностей. Также мы научились реализовывать их.

# ЛАБОРАТОРНАЯ РАБОТА №8 РАБОТА С ФАЙЛАМИ

## Цель работы

Освоить на практике работу с файлами на языке Java. Получить практические навыки по чтению и записи данных в файл.

## Теоретическое введение

Запись файлов. Класс FileWriter

Класс FileWriter является производным от класса Writer. Он используется для записи текстовых файлов

Так, в конструктор передается либо путь к файлу в виде строки, либо объект File, который ссылается на конкретный текстовый файл. Параметр append указывает, должны ли данные дозаписываться в конец файла (если параметр равен true), либо файл должен перезаписываться.

Чтение файлов. Класс FileReader

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность для чтения текстовых файлов.

## Постановка задачи

Реализовать запись в файл введённой с клавиатуры информации

## Программный код



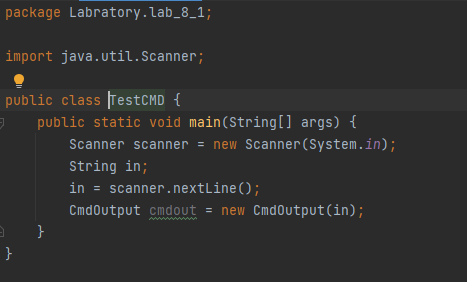
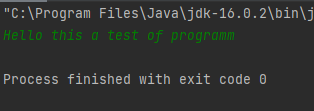


Рис. Код программы

## Вывод программы



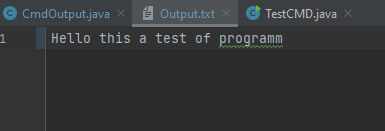


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы узнали о классах и методах для работы с файлами. Научились работать с ними и использовать их для вывода и ввода информации.

# ЛАБОРАТОРНАЯ РАБОТА № 9 РАБОТА С ИСКЛЮЧЕНИЯМИ

## Цель работы

Целью данной лабораторной работы являются получение практических навыков разработки программ, изучение синтаксиса языка Java, освоение основных конструкций языка Java (циклы, условия, создание переменных и массивов, создание методов, вызов методов), а также научиться осуществлять стандартный ввод/вывод данных.

## Теоретическое введение

Механизм исключительных ситуаций в Java поддерживается пятью ключевыми словами:

• try

• catch

• finally

• throw

• throws

В Java всего около 50 ключевых слов, и пять из них связано с исключениями: try, catch, finally, throw, throws. Из них catch, throw и throws применяются к экземплярам класса, причём работают они только с Throwable и его наследниками.

Наиболее популярные исключений в Java представлены в таблице 1.

Таблица 1. Классы иcключений в Java

|  |  |  |
| --- | --- | --- |
| № пп | Класс исключения | Класс предок/тип |
| 1 | ArithmeticException | RuntimeException |
| 2 | NegativeArraySizeException | RuntimeException |
| 3 | ArrayIndexOutOfBoundsException | RuntimeException |
| 4 | NoSuchElementException | RuntimeException |
| 5 | ArrayStoreException | RuntimeException |
| 6 | NotSerializableException | Exception |
| 7 | AssertionError | Error |
| 8 | NullPointerException | RuntimeException |
| 9 | ClassCastException | RuntimeException |
| 10 | NumberFormatException | RuntimeException |
| 11 | ClassNotFoundException | Exception |
| 12 | OutOfMemoryError | Error |
| 13 | CloneNotSupportedException | Exception |
| 14 | SecurityException | RuntimeException |
| 15 | ConcurrentModificationException | RuntimeException |
| 16 | StackOverflowError | Error |
| 17 | EOFException | Exception |
| 18 | StringIndexOutOfBoundsException | RuntimeException |
| 19 | FileNotFoundException | Exception |
| 20 | ThreadDeath | Error |
| 21 | IllegalArgumentException | RuntimeException |
| 22 | UnsupportedEncodingException | Exception |
| 23 | InterruptedException | Exception |
| 24 | UnsupportedOperationException | RuntimeException |

То, что исключения являются объектами важно по двум причинам:

1) они образуют иерархию с корнем java.lang.Throwable (java.lang.Object — предок java.lang.Throwable, но Object —это не исключение!)

2) они могут иметь поля и методы

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу класса родителя перехватывает исключения для экземпляров объектов как родительского класса, так или его наследников(т.е. экземпляры непосредственно самого родительского класса или любого его потомка).

## Постановка задачи

Упражнения (Основы Try-Catch- Finally)

Задание № 1

Выполните следующую программу и посмотрите, что происходит:

public class Exception1 {

public void exceptionDemo() {

System.out.println( 2 / 0 );

}

}

Листинг 1

Необходимо инстанцировать класс и выполнить exceptionDemo (). Программа даст сбой, и вы получите следующее сообщение j

ava.lang.ArithmeticException: / by zero at Exception1.exceptionDemo(Exception1.java:12)

Это говорит нам о том, что программа пытается выполнить деление на ноль, который он не в состоянии выполнить.

Интересный эксперимент

Замените 2/0 на 2,0 / 0,0 и повторно вызовите метод. Что произойдет?

Теперь измените код в Exception1 чтобы включить try-catch block следующим образом:

public class Exception1 {

public void exceptionDemo() {

try{ System.out.println( 2 / 0 ); }

catch ( ArithmeticException e ) {

System.out.println("Attempted division by zero");

}

}

}

Листинг 2

Запустите программу и обратите внимание на новое поведение.

Задание № 2

Измените код в листинге 1 на следующий:

public class Exception2 {

public void exceptionDemo() {

Scanner myScanner = new Scanner( System.in );

System.out.print( "Enter an integer ");

String intString = myScanner.next();

int i = Integer.parseInt(intString); System.out.println( 2 / i );

}

}

Листинг 3

Запустите эту программу со следующими выводами: Qwerty 0 1.2 1

Посмотрите на вывод. Какие исключения выбрасываются?

Измените код, добавив блоки try – catch, чтобы иметь дело с определяемыми исключениями.

Задание № 3

С помощью перехватывания исключений можно оказывать влияние на поведение программы. В вашем решении в предыдущем упражнении вы можете добавить новый пункт - catch в начале списка пунктов catch. Выполите это, чтобы поймать общее исключение класса Exception. Перезапустите программу с приведенными выше данными и обратите внимание на ее поведение. Объясните новое поведение программы

Задание № 4

И наконец добавьте блок finally к решению Задания №2. Повторно запустите программу, чтобы проверить ее поведение. Объясните новое поведение программы

Генерация собственных исключений

На предыдущем шаге при выполнении заданий мы рассмотрели, как Java работает с предопределенными исключениями, теперь перейдем к тому, как генерируется исключение.

Все исключения в рассмотренных ранее примерах и заданиях были определены заранее. В этом разделе лабораторной работы вы будете создавать и пробрасывать свои собственные исключения (exceptions).

Задание № 5

Самый простой способ генерации исключения показан в следующем примере кода:

public class ThrowsDemo {

public void getDetails(String key) {

if(key == null) {

throw new NullPointerException( "null key in getDetails" );

}

// do something with the key

}

}

Листинг 4

Когда определяется условие ошибки, то мы выбрасываем исключение с определенным именем. Сообщение может быть связано с исключением. Откомпилируйте этот класс, создайте его экземпляр и выполните метод getDetails() с нулем в качестве значения параметра.

Вывод может быть следующим:

java.lang.NullPointerException: null key in getDetails at ThrowsDemo.getDetails(ThrowsDemo.java:13)

Добавьте блок try-catch таким образом, чтобы перехватить исключение и рассмотреть его внутри метода.

Подумайте, является ли этот способ подходящим, чтобы иметь дело с этим исключением? Объясните поведение программы.

Причиной ошибки, может является, например неправильное значение для параметра. Может было бы лучше, если бы метод вызывал getDetails() и там решалась бы эта проблема.

Обратите внимание на следующее:

public class ThrowsDemo {

public void printMessage(String key) {

String message = getDetails(key);

System.out.println( message );

}

public String getDetails(String key) {

if(key == null) {

throw new NullPointerException( “null key in getDetails” );

}

return “data for” + key;

}

}

Листинг 5

Откомпилируйте и запустите эту программу с правильным значением для ключа и с нулем в качестве значения. При выполнении с нулевым значением вы увидите некоторый вывод. Обобщите все вышесказанное и выполните эту программу с правильным значением для ключа и с нулем в ключе.

java.lang.NullPointerException: null key in getDetails

at ThrowsDemo.getDetails(ThrowsDemo.java:21)

at ThrowsDemo.printMessage(ThrowsDemo.java:13)

Теперь добавьте блоки try-catch, чтобы использовать для вывода сообщений метод printMessage(), таким образом, чтобы исключения обрабатывались и программа не “ломалась”.

Задание № 6

Теперь мы расширим наш пример для демонстрации прохождения исключения через цепочку вызовов. Создайте следующий класс и попытатайтесь его скомпилировать:

public class ThrowsDemo {

public void getKey() {

Scanner myScanner = new Scanner( System.in );

String key = myScanner.next();

printDetails( key );

}

public void printDetails(String key) {

try { String message = getDetails(key); System.out.println( message ); }

catch ( Exception e){ throw e; }

}

private String getDetails(String key) {

if(key == "") {

throw new Exception( "Key set to empty string" );

}

return "data for " + key;

}

}

Листинг 6

При попытке компиляции вы получите следующий синтаксис ошибки:

Unreported java.lang.Exception исключение;

В результате успешного пробрасывания исключение должен быть поймано или объявлено. Причиной полученной ошибки является выражение throw е.

В данном случае метод printDetails () решил, что он не может иметь дело с исключением и проходит все дерево его вызовов. Поскольку метод GetKey() не имеет блока try-catch для обработки исключений, то Java становится перед выбором, что в таком случае делать.

Проблему можно решить несколькими способами:

1) Добавьте соответствующие try-catch блоки таким образом, чтобы в конечном итоге один из них обрабатывал исключение;

2) Удалите блоки try-catch для всех методов, кроме одного, который обрабатывает исключение. Добавьте throws, котрый бросает исключение методу, который проходит исключение без обработки.

Задание № 7

Измените код программы, чтобы она работала корректно.

Следующий фрагмент кода программы демонстрирует случай, когда никакие методы не будут брать на себя ответственность за обработку ошибок, и программа в конечном итоге выходит “сломается “.

public class ThrowsDemo {

public void getKey() throws Exception {

Scanner myScanner = new Scanner( System.in );

System.out.print("Enter Key ");

String key = myScanner.nextLine();

printDetails( key );

}

public void printDetails(String key) throws Exception {

String message = getDetails(key);

System.out.println( message );

}

private String getDetails(String key) throws Exception {

if(key.equals("")) {

throw new Exception( "Key set to empty string" );

}

return "data for " + key;

}

}

Листинг 7

Задание №8

Измените этот код следующим образом:

1. Удалите throws Exception из метода getKey().

2. Измените метод getKey(), добавив try-catch блок для обработки исключений.

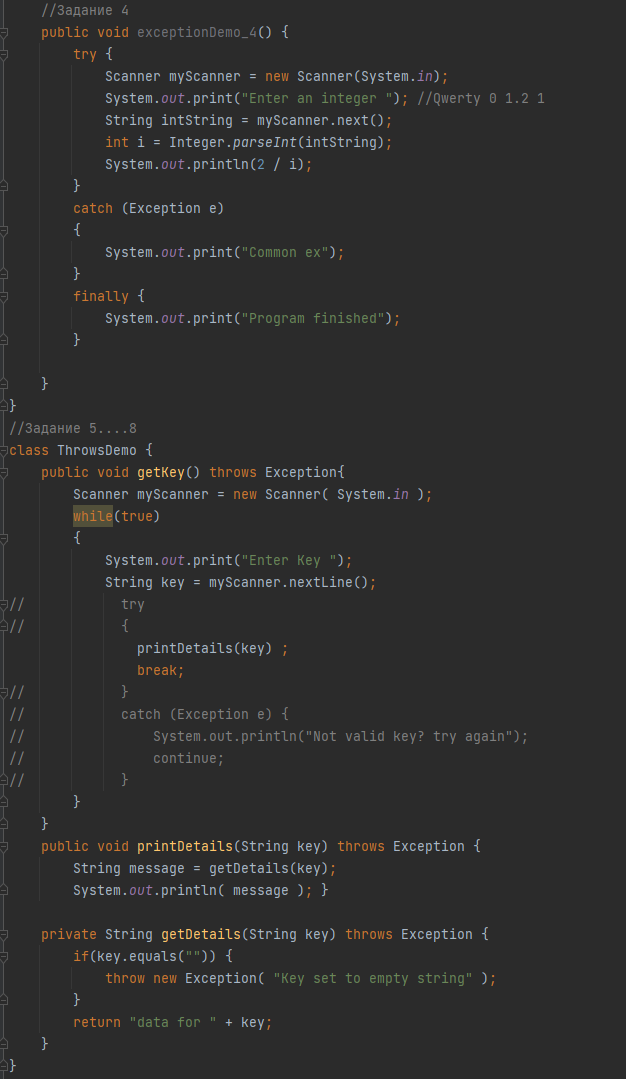
3. Добавьте цикл к getKey() таким образом, чтобы пользователь получил еще один шанс на ввод значение ключа

Замечания:

Инструкция throw очень аналогична инструкции return – она прекращает выполнение метода, дальше мы не идем. Если мы нигде не ставим catch, то у нас бросок Exception очень похож на System.exit() – система вырубается. Но мы в любом месте можем поставить catch и, таким образом, предотвратить поломку кода.

## Программный код





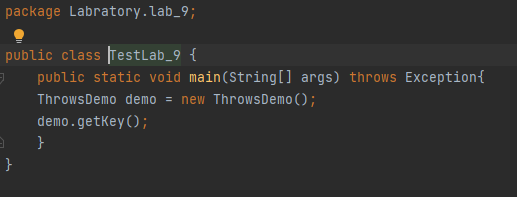


Рис. Код программы

## Вывод программы

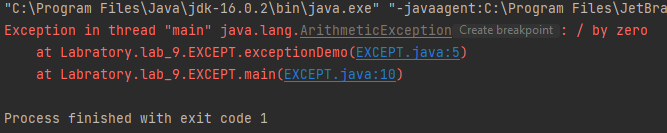


Рис. Деление 2/0 невозможно

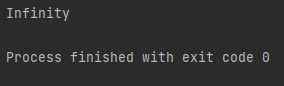


Рис. Деление на 2.0 на 0.0 Выведется бесконечность т.к. 0.0. считается как бесконечно малую. А деление на бесконечно малую даёт бесконечность.



Рис. Мы поймали арифметическое исключение и вывели сообщение об этом, благодаря этому программа продолжала работать.

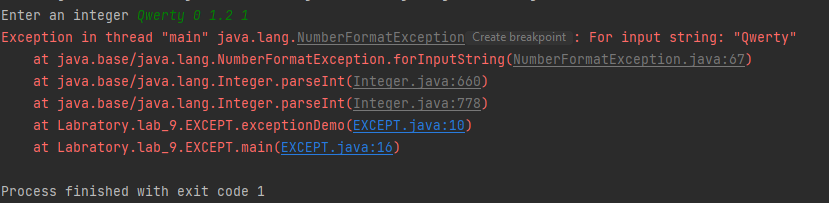


Рис. Выбросилось исключение типа переменной. Невозможность перевода Qwerty в число

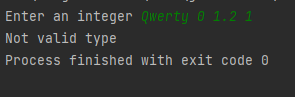


Рис. Теперь мы обработали данное исключение и выводим сообщение, когда оно возникает

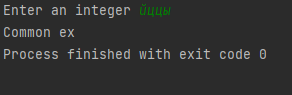
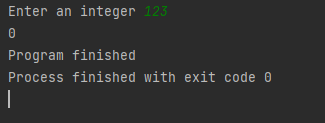


Рис. Когда мы поставили обычное exception, в начало, то мы ловим абсолютно все исключения, из-за чего после него нету смысла просто что-то ловить, т.к. всё поймается и на нём. Поэтому должна быть очередность исключений для проверки, и этот exception должен быть в самом низу, если мы хотим поймать определенные исключения.



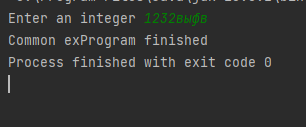
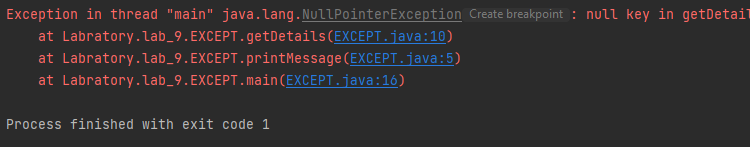
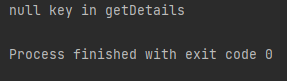


Рис. Блок finally обрабатывается в самом конце блока try catch вне зависимости от того, как он сработает





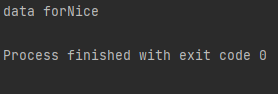
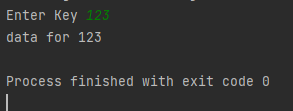


Рис. До и после добавления блока try catch



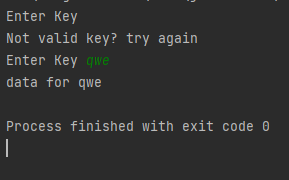


Рис. Вывод после изменения данного кода

## Вывод

Фактически при работе с исключениями весь материал делится на две части: синтаксис (ответ на вопрос, что компилятор пропустит, а что нет) и семантика (вопрос, как лучше делать) исключений. В отличие от вариантов с for, while, switch, использование исключений – более сложный механизм. Но он сложен не синтаксически, а семантически, по своему подходу. То есть при генерации исключений нужно думать о том - не как правильно его использовать, а с каким смыслом его использовать. То есть вопрос, в каких ситуациях стоит ли ломать систему и где, а в каких ситуациях ее восстанавливать.

В хорошей инженерной системе каждый любой модуль всегда проверяет все аргументы на входе.

Можно еще сказать что существует иерархия различных способов прекращения выполнения некоего действия (или ряда действий) в виде участка кода и эта иерархия классифицирует возможные действия по мощности используемого оператора: continue, break, return, throw.

• continue прекращает выполнение данной итерации в цикле;

• break прекращает выполнение данного блока (например цикла);

• return – это инструкция выхода из данного метода (например прекращение выполнения функции);

• throw – еще более сильная инструкция, она прекращает выполнение данного метода и метода, который его вызвал. Так-как исключения вообще-то позволяют сломать весь стек работы программы.

# ЛАБОРАТОРНАЯ РАБОТА № 10 РАБОТА С ДЖЕНЕРИКАМИ. СТИРАНИЕ ТИПОВ

## Цель работы

Цель данной лабораторной работы – научиться работать с обобщенными типами в Java и применять их в программах.

## Теоретические сведения

Понятие дженериков.

Введение в Дженерики. В JDK представлены дженерики (перевод с англ. generics), которые поддерживают абстрагирование по типам (или параметризованным типам). В объявлении классов дженерики представлены обобщенными типами, в то время как пользователи классов могут быть конкретными типами, например во время создания объекта или вызова метода.

Вы, конечно, знакомы с передачей аргументов при вызове методов в языке С++. Когда вы передаете аргументы внутри круглых скобок () в момент вызова метода, то аргументы подставляются вместо формальных параметров с которыми объявлен и описан метод. Схожим образом в generics вместо передаваемых аргументов мы передаем информацию только о типах аргументов внутри угловых скобок <> (так называемая diamond notation или алмазная запись).

Основное назначение использования дженериков — это абстракция работы над типами при работе с коллекциями («Java Collection Framework»).

Чтобы создать экземпляр ArrayList, пользователям необходимо предоставить фактический тип для для данного конкретного экземпляра. Фактический тип будет заменять все ссылки на E внутри класса.

В приведенном выше примере показано, что при проектировании или определении классов, они могут быть типизированными по общему типу; в то время как пользователи классов предоставляют конкретную фактическую информацию о типе во время создания экземпляра объекта типа класс. Информация о типе передается внутри угловых скобок <>, так же как аргументы метода передаются внутри круглой скобки ().

В этом плане общие коллекции не являются безопасными типами!

Если вы знакомы с классами- коллекциями, например, такими как ArrayList, то вы знаете, что они предназначены для хранения объектов типа java.lang.Object. Используя основной принцип ООП-полиморфизм, любой подкласс класса Object может быть заменен на Object. Поскольку Object является общим корневым классом всех классов Java, то коллекция, предназначенная для хранения Object, может содержать любые классы Java. Однако есть одна проблема. Предположим, например, что вы хотите определить ArrayList из объектов класса String. То при выполнении операций, например операции add(Object) объект класса String будет неявным образом преобразовываться в Object компилятором. Тем не менее, во время поиска ответственность программиста заключается в том, чтобы явно отказаться от Object обратно в строку. Если вы непреднамеренно добавили объект не-String в коллекцию, то компилятор не сможет обнаружить ошибку, а понижающее приведение типов (от родителя к дочернему) завершится неудачно во время выполнения (будет сгенерировано ClassCastException throw).

Классы дженерики или параметризованные классы

В JDK введены так называемые обобщенные или параметризованные типы – generics или подругому обобщенные типы для решения вышеописанной проблемы. Параметризованных (generic) классы и методы, позволяют использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Использование параметризации позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Дженерики или обобщенные типы позволяют вам абстрагироваться от использования конкретных типов. Вы можете создать класс с таким общим типом и предоставить информацию об определенном типе во время создания экземпляра объекта типа класс. А компилятор сможет выполнить необходимую проверку типов во время компиляции. Таким образом, вы сможете убедиться, что во время выполнения программы не возникнет ошибка выбора типа еще на этапе компиляции, что как раз является безопасностью для используемого типа.

Такая запись - называется формальным параметром типа, который может использоваться для передачи параметров «типа» во время создания фактического экземпляра типа. Механизм похож на вызов метода Напомним, что в определении метода мы всегда объявляем формальные параметры для передачи данных в метод (при описании метода используются формальные параметры, а при вызове на их место подставляются аргументы).

Параметры формального типа, используемые в объявлении класса, имеют ту же цель, что и формальные параметры, используемые в объявлении метода. Класс может использовать формальные параметры типа для получения информации о типе, когда экземпляр создается для этого класса. Фактические типы, используемые во время создания, называются фактическими типами параметров.

Вернемся к java. util.List , итак в действительности, когда тип определен, например List , все вхождения параметра формального типа заменяются актуальным или фактическим параметром типа . Используя эту дополнительную информацию о типе, компилятор может выполнить проверку типа во время компиляции и убедиться, что во время выполнения не будет ошибки при использовании типов.

Конвенция кода Java об именах для формальных типов

Мы должны помнить, что написания “чистого кода” необходимо руководствоваться конвенцией кода на Java. Поэтому, для создания имен формальных типов используйте один и тот же символ в верхнем регистре. Например,

• <E> для элемента коллекции;

• <T> для обобщенного типа;

• <K,V> ключ и значение.

• <N> для чисел

• S,U,V, и т.д. для второго, третьего, четвертого типа параметра

Стирание типов

Из предыдущего примера может создаться видимость того, что компилятор заменяет параметризованный тип актуальным или фактическим типом (таким как String, Integer) во время создания экземпляра объекта типа класс. Если это так, то компилятору необходимо будет создавать новый класс для каждого актуального или фактического типа (аналогично шаблону C ++).

На самом же деле происходит следующее - компилятор заменяет всю ссылку на параметризованный тип E на ссылку на Object, выполняет проверку типа и вставляет требуемые операторы, обеспечивающие понижающее приведения типов.

Вывод. Таким образом, для всех типов используется одно и то же определение класса. Самое главное, что байт-код всегда совместим с теми классами, у которых нет дженериков. Этот процесс называется стиранием типа.

В объявлении MyGenericArrayList объявляется класс-дженерик с формальным параметром типа . Во время фактического создания объектов типа класс, например, MyGenericArrayList , определенного типа или параметра фактического типа, подставляется вместо параметра формального типа .

Помните! Что Дженерики реализуются компилятором Java в качестве интерфейсного преобразования, называемого стиранием, которое переводит или перезаписывает код, который использует дженерики в не обобщенный код (для обеспечения обратной совместимости). Это преобразование стирает всю информацию об общем типа. Например, ArrayList станет ArrayList. Параметр формального типа, такой как , заменяется объектом по умолчанию (или верхней границей типа). Когда результирующий код не корректен, компилятор вставляет оператор преобразования типа.

Выводы: с помощью дженериков компилятор может выполнять проверку типов во время компиляции и обеспечивать безопасность использования типов во время выполнения.

Запомните. В отличие от «шаблона» в C ++, который создает новый тип для каждого определенного параметризованного типа, в Java класс generics компилируется только один раз, и есть только один файл класса, который используется для создания экземпляров для всех конкретных типов.

Обобщенные методы (параметризованные методы)

Методы также могут быть определены с помощью общих типов (аналогично родовому классу).

При объявлении обобщенного метода или метода-дженерика могут объявляться параметры формального типа (например, такие как , ) перед возвращаемым типом (в примере выше это static void). Параметры формального типа могут затем использоваться в качестве заполнителей для типа возвращаемого значения, параметров метода и локальных переменных в общем методе для правильной проверки типов компилятором.

Подобно классам-дженерикам, когда компилятор переводит общий метод, он заменяет параметры формального типа, используя операцию стирания типа. Все типы заменяются типом Object по умолчанию (или верхней границей типа – типом классом, стоящим на самой вершине иерархии наследования).

## Постановка задачи

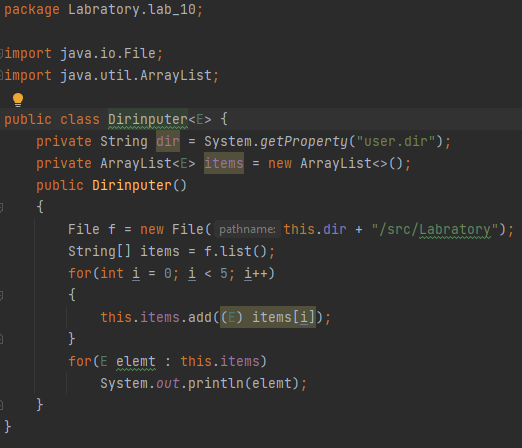
1. Написать метод для конвертации массива строк/чисел в список.

2. Написать класс, который умеет хранить в себе массив любых типов данных (int, long etc.).

3. Реализовать метод, который возвращает любой элемент массива по индексу.

4. Написать функцию, которая сохранит содержимое каталога в список и выведет первые 5 элементов на экран.

## Программный код







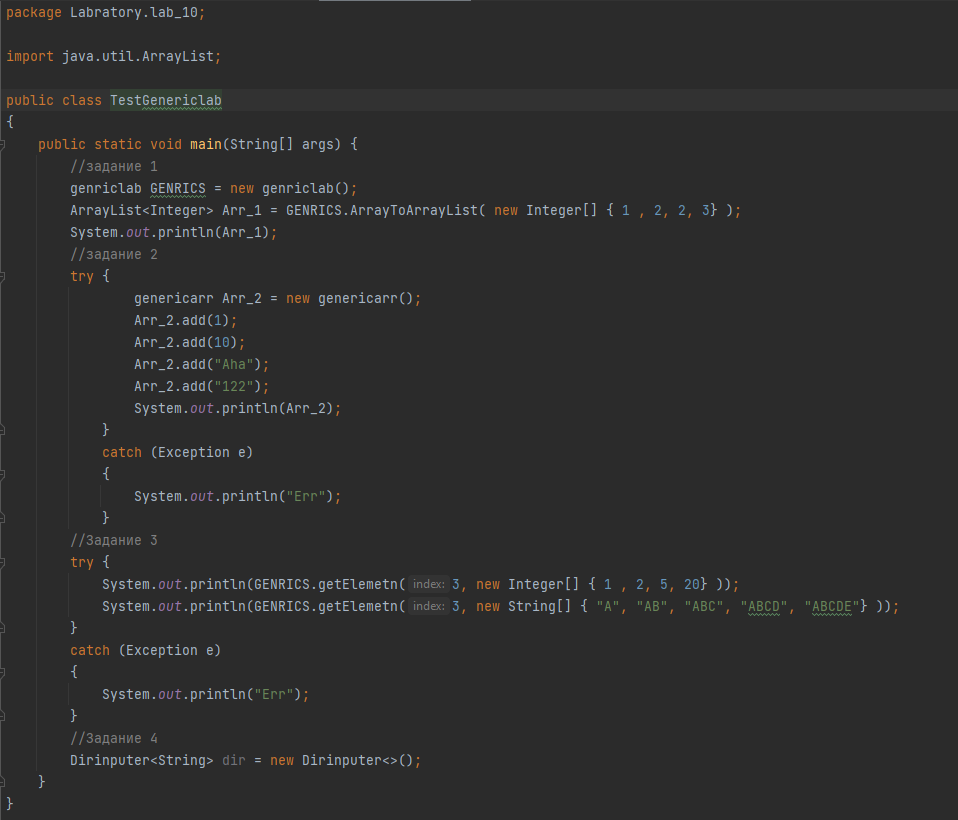


Рис. Код программы

## Вывод программы

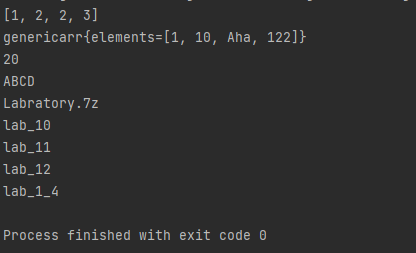


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы познакомились с понятием дженериков. Научились писать классы с их использованием, узнали разницу между разными типами дженериков. Также научились писать метод с использованием дженериков и в целом научились реализовывать их.

# ЛАБОРАТОРНАЯ РАБОТА №11 ДЖЕНЕРИК MAP

## Цель работы

Цель работы разработать generic hashmap.

## Постановка задачи

Основываясь на предыдущей работе: Реализуйте вспомогательные методы в классе Solution, которые должны создавать соответствующую коллекцию и помещать туда переданные объекты. Методы newArrayList, newHashSet параметризируйте общим типом T. Метод newHashMap параметризируйте парой , то есть типами К- ключ и Vзначение.

## Программный код



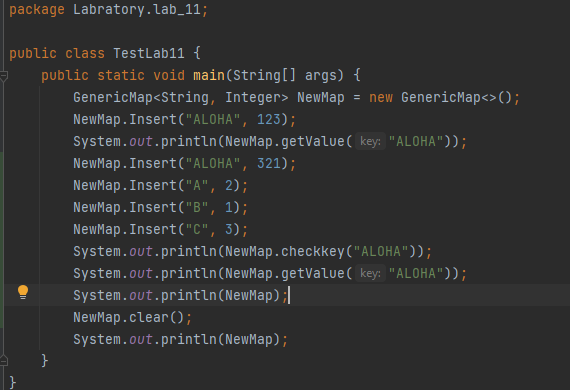


Рис. Код программы

## Вывод программы

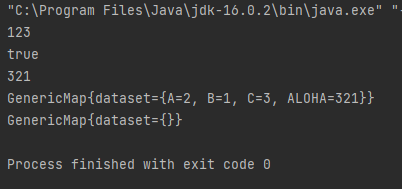


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы научились создавать структуры с использованием дженериков. Таких как например HashMap, реализовывать методы для них и работать в общем.

# ЛАБОРАТОРНАЯ РАБОТА №12 ОБРАБОТКА СТРОК. ИСПОЛЬЗОВАНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ В JAVA-ПРИЛОЖЕНИЯХ

## Цель работы

Цель данной лабораторной работы – понять особенности использования регулярных выражений в Java, научиться работать с строками и применять регулярные выражения для обработки строк в программах

## Теоретическое введение

Регулярные выражения и их использование в Java программах для обработки строк

Регулярные выражения – эта система обработки текста, основанная на специальной системе записи образцов для поиска. Образец (pattern), задающий правило поиска, по-русски также иногда называют «шаблоном», «маской». Сейчас регулярные выражения используются многими текстовыми редакторами и утилитами для поиска и изменения текста на основе выбранных правил. Язык программирования Java также поддерживает регулярные выражения для работы со строками.

Основными классами для работы с регулярные выражения являются класс java.util.regex.Pattern и класс java.util.regex.Matcher. Для определения шаблона применяются специальные синтаксические конструкции.

Класс Pattern

Класс java.util.regex.Pattern применяется для определения регулярных выражений, для которого ищется соответствие в строке, файле или другом объекте представляющем собой некоторую последовательность символов. Этот класс используется для простой обработки строк. Для более сложной обработки строк используется класс Matcher, рассматриваемый ниже.

В классе Pattern объявлены следующие методы:

• compile(String regex) – возвращает Pattern, который соответствует regex;

• matcher(CharSequence input) – возвращает Matcher, с помощью которого можно находить соответствия в строке input;

• matches(String regex, CharSequence input) – проверяет на соответствие строки input шаблону regex;

• pattern() – возвращает строку, соответствующую шаблону;

• split(CharSequence input) – разбивает строку input, учитывая, что разделителем является шаблон;

• split(CharSequence input, int limit) – разбивает строку input на не более чем limit частей.

С помощью метода matches() класса Pattern можно проверять на соответствие шаблону целой строки, но если необходимо найти соответствия внутри строки, например, определять участки, которые соответствуют шаблону, то класс Pattern не может быть использован. Для таких операций необходимо использовать класс Matcher.

Класс Matcher

С помощью класса java.util.regex.Matcher можно получить больше информации каждом соответствии.

Начальное состояние объекта типа Matcher не определено. Попытка вызвать какой-либо метод класса для извлечения информации о найденном соответствии приведет к возникновению ошибки IllegalStateException. Для того чтобы начать работу с объектом Matcher нужно вызвать один из его методов:

• matches() – проверяет, соответствует ли вся строка шаблону;

• lookingAt() – пытается найти последовательность символов, начинающуюся с начала строки и соответствующую шаблону;

• find() или find(int start) – пытается найти последовательность символов, соответствующих шаблону, в любом месте строки. Параметр start указывает на начальную позицию поиска.

Иногда необходимо сбросить состояние объекта класса Matcher в исходное, для этого применяется метод reset() или reset(CharSequence input), который также устанавливает новую последовательность символов для поиска.

Для замены всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку можно применить метод replaceAll(String replacement).

Для того чтобы ограничить поиск границами входной последовательности применяется метод region(int start, int end), а для получения значения этих границ – regionEnd() и regionStart(). С регионами связано несколько методов:

useAnchoringBounds(boolean b) – если установлен в true, то начало и конец региона соответствуют символам ^ и $ соответственно; hasAnchoringBounds() – проверяет закрепленность границ.

В регулярном выражении для более удобной обработки входной последовательности применяются группы, которые помогают выделить части найденной подпоследовательности. В шаблоне они обозначаются скобками «(» и «)». Номера групп начинаются с единицы. Нулевая группа совпадает со всей найденной подпоследовательностью. Далее приведены методы для извлечения информации о группах:

end() – возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;

end(int group) – возвращает индекс последнего символа указанной группы;

group() – возвращает всю подпоследовательность, удовлетворяющую шаблону;

group(int group) – возвращает конкретную группу;

groupCount() – возвращает количество групп;

start() – возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;

start(int group) – возвращает индекс первого символа указанной группы;

hitEnd() – возвращает истину, если был достигнут конец входной последовательности.

В классе Matcher объявлены два полезных метода для замены найденных подпоследовательностей во входной строке.

Matcher appendReplacement(StringBuffer sb, String replacement) – метод читает символы из входной стоки и добавляет их в sb. Чтение останавливается на start() – первой позиции предыдущего совпадения, после чего происходит добавление в sb строки replacement. При следующем вызове этого метода, производится добавление символов, начиная с символа с индексом end() предыдущего совпадения.

Класс StringTokenizer

Этот класс предназначен для разбора строки по лексемам (tokens). Строка которую необходимо разобрать передается в качестве параметра конструктору StringTokenizer(String str). Определено еще два перегруженных конструктора, которым дополнительно можно передать строку-разделитель лексем StringTokenizer(String str,String delim) и признак возврата разделителя лексем StringTokenizer(String str,String delim,Boolean retirnDelims). Разделителем лексем по умолчанию служит пробел.

## Постановка задачи

1. Необходимо реализовать консольное приложение, позволяющее манипулировать строкой, разбив ее на элементы путем использования регулярных выражений.

2. Написать регулярное выражение, определяющее является ли данная строка строкой "abcdefghijklmnopqrstuv18340" или нет.

a) пример правильных выражений: abcdefghijklmnopqrstuv18340

b) пример неправильных выражений: abcdefghijklmnoasdfasdpqrstuv18340.

1. Дан текст со списками цен. Извлечь из него цены в USD, RUВ, EU. – пример правильных выражений: 25.98 USD. – пример неправильных выражений: 44 ERR, 0.004 EU.

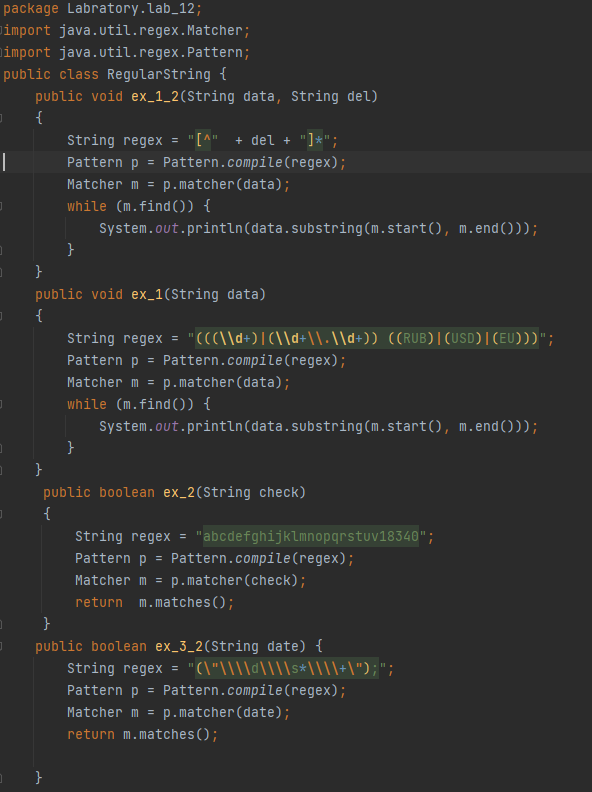
2. Дан текст, необходимо проверить есть ли в тексте цифры, за которыми не стоит знак «+». – пример правильных выражений: (1 + 8) – 9 / 4. – пример неправильных выражений: 6 / 5 – 2 \* 9 .

3. Написать регулярное выражение, определяющее является ли данная строчка датой в формате dd/mm/yyyy. Начиная с 1900 года до 9999 года. – пример правильных выражений: 29/02/2000, 30/04/2003, 01/01/2003. – пример неправильных выражений: 29/02/2001, 30-04-2003, 1/1/1899.

4. Написать регулярное выражение, определяющее является ли данная строчка допустимым (корректным) е-mail адресом согласно RFC под номером 2822. – пример правильных выражений: user@example.com, root@localhost – пример неправильных выражений: myhost@@@com.ru, @my.ru, Julia String.

5. Проверить, надежно ли составлен пароль. Пароль считается надежным, если он состоит из 8 или более символов. Где символом может быть цифр, английская буква, и знак подчеркивания. Пароль должен содержать хотя бы одну заглавную букву, одну маленькую букву и одну цифру. – пример правильных выражений: F032\_Password, TrySpy1. – пример неправильных выражений: smart\_pass, A007.

## Программный код



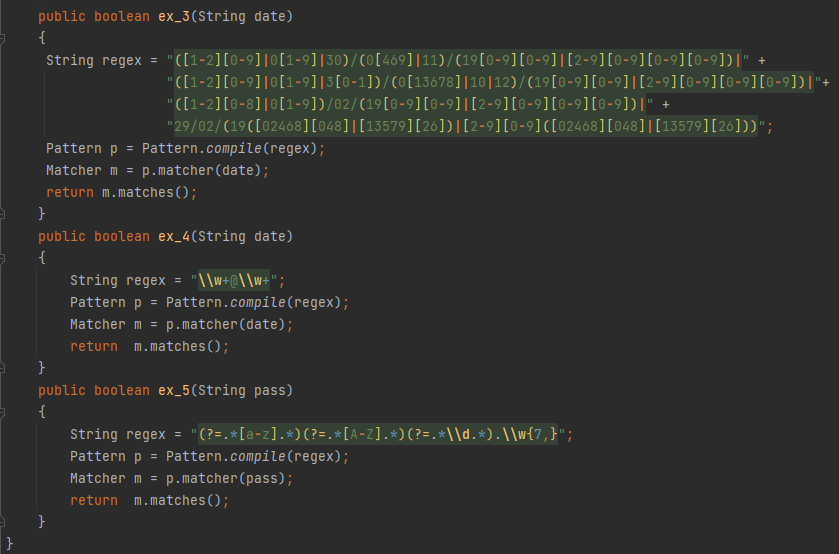




Рис. Код программы

## Вывод программы

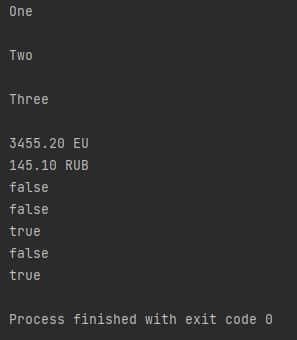


Рис. Вывод программы

## Вывод

В результате выполнения данной лабораторной работы мы узнали, что такое строки и регулярные выражения в Java. Научились работать с ними, узнали методы и классы для работы с ними, и в целом научились реализовывать программы используя их.

# ЛАБОРАТОРНАЯ РАБОТА №

## Цель работы

## Постановка задачи

## Программный код

## Вывод программы

## Вывод

В результате выполнения данной лабораторной работы