5947

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ В.Ф. УТКИНА

ОСНОВЫ ПРОМЫШЛЕННОЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Часть 2

Методические указания
к лабораторным и практическим занятиям

Рязань 2020

УДК 004.43

Основы промышленной разработки программного обеспечения. Часть 2: методические указания к лабораторным и практическим занятиям / Рязан. гос. радиотехн. ун-т; сост.: Б.В. Костров, А.С. Бастрычкин, Е.А. Трушина. Рязань, 2020, 60 с.

Содержат методические материалы для подготовки к выполнению лабораторных и практических работ.

Предназначены для бакалавров, обучающихся по направлениям: 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.01 «Информатика и вычислительная техника» и 38.03.05 «Бизнес-информатика», а также для специалистов, обучающихся по направлению 27.05.01 «Специальные организационно-технические системы»

Ил. 2. Табл. 4. Библиогр.: 3 назв.

Язык программирования Java, объектно-ориентированное программирование, программное обеспечение, промышленная разработка

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра электронных вычислительных машин Рязанского государственного радиотехнического университета (зав. каф. Б.В. Костров)

Основы промышленной разработки
программного обеспечения

Составители: Костров Борис Васильевич
Бастрычкин Александр Сергеевич
Трушина Евгения Александровна

Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.

Содержание

Занятие № 6
Внутренние классы

Цель работы: изучение внутренних классов, приобретение навыков использования *inner* классов в Java-программах.

1. Теоретическая часть

1.1. Внутренние классы – нестатические вложенные классы

В Java, переменные класса тоже могут иметь в качестве своего члена другой класс. Допускается написание класса внутри другого. Класс, написанный внутри, называется **вложенным классом**, а класс, который содержит внутренний класс, называется **внешним классом.**

Ниже приведен синтаксис для записи **вложенного класса**. Здесь класс **Outer_Demo** –внешний класс, а класс **Inner_Demo** – вложенный.

```
class Outer_Demo {
   class Nested_Demo {
   }
}
```

Вложенные классы в Java делятся на два типа:

- ***Нестатические вложенные классы*** – нестатические члены класса.
- ***Статические вложенные классы*** – статические члены класса.

Классификация вложенных классов представлена на рис. 1.

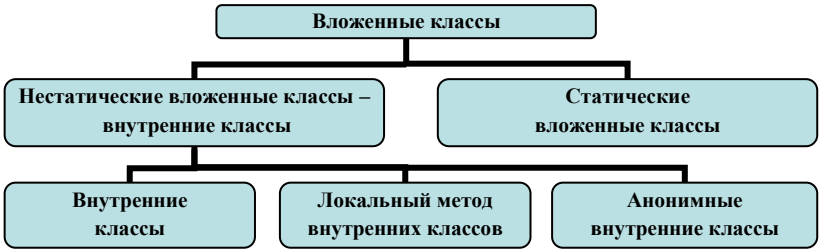

Рисунок 1 **–** Классификация вложенных классов

Внутренние классы — это механизм безопасности в Java. Класс не может быть связан с модификатором доступа **private**, но если есть класс как член другого класса, то внутренний класс может быть **private**. Это используется для доступа к закрытым (**private**) членам класса.

В Java внутренние классы имеют три типа в зависимости от того, как и где они определены: *внутренний класс*, *локальный метод внутреннего класса*, *анонимный внутренний класс*.

1.1.1. Создание внутренних классов

Для создания внутреннего класса нужно написать класс внутри класса. Внутренний класс может быть закрытым (**private**), тогда он не может быть доступен из объекта вне класса. Создадим внутренний класс **private** и получаем доступ к классу с помощью метода:

```
class Outer_Demo {
   int num;
   // Внутренний класс
   private class Inner_Demo {
      public void print() {
         System.out.println("Это внутренний класс");
      }
   }
   // Доступ к внутреннему классу из метода
   void display_Inner() {
      Inner_Demo inner = new Inner_Demo();
      inner.print();
   }
}
public class My_class {
   public static void main(String args[]) {
      // Создание внешнего класса
      Outer_Demo outer = new Outer_Demo();
      // Доступ к методу display_Inner()
      outer.display_Inner();
   }
}
```

Здесь **Outer_Demo** – внешний класс, **Inner_Demo** – внутренний класс, **display_Inner()** – метод, внутри которого мы создаем внутренний класс, и этот метод вызывается из основного метода.

Получим следующий результат:

```
Это внутренний класс
```

1.1.2. Доступ к частным (private) членам

Внутренние классы используются для доступа к закрытым членам класса. Предположим, у класса есть **private** члены. Для доступа к ним напишите в нем внутренний класс, верните частные члены из метода внутри внутреннего класса, скажем, методом **getValue()** и из другого класса (из которого нужно получить доступ к закрытым членам) вызовите метод **getValue()** внутреннего класса.

Чтобы создать экземпляр внутреннего класса, сначала необходимо создать экземпляр внешнего класса. После этого, используя объект внешнего класса, можно создать экземпляр внутреннего класса.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

Следующий пример показывает, как получить доступ к закрытым членам класса с использованием внутреннего класса.

```
   class Outer_Demo {
      // Частная переменная внешнего класса
      private int num = 2021;
      // Внутренний класс
      public class Inner_Demo {
         public int getNum() {
            System.out.println("Это метод getnum внутреннего
класса");
            return num;
         }
      }
   }
   public class My_class2 {
      public static void main(String args[]) {
         // Создание внешнего класса
         Outer_Demo outer = new Outer_Demo();
         // Создание внутреннего класса
         Outer_Demo.Inner_Demo inner = outer.new
Inner_Demo();
         System.out.println(inner.getNum());
      }
   }
```

В итоге мы увидим на консоли:

```
Это метод getnum внутреннего класса
2021
```

1.1.3. Локальный метод внутреннего класса

В Java можно написать класс внутри метода, и это будет локальный тип. Возможности внутреннего класса ограничены в рамках метода.

Локальный метод внутреннего класса может быть создан только внутри метода, где определяется внутренний класс:

```
   public class Outerclass {
      // Метод экземпляра внешнего класса
      void my_Method() {
         int num = 666;
         // Локальный метод внутреннего класса
         class MethodInner_Demo {
            public void print() {
               System.out.println("Это метод внутреннего
класса: " + num);
            }
         } // Конец внутреннего класса
         // Доступ к внутреннему классу
         MethodInner_Demo inner = new MethodInner_Demo();
         inner.print();
```

```
      }
      public static void main(String args[]) {
         Outerclass outer = new Outerclass();
         outer.my_Method();
      }
   }
```

Получим следующий результат:

```
Это метод внутреннего класса: 666
```

1.1.4. Анонимные внутренние классы в Java

Анонимный внутренний класс — это внутренний класс, объявленный без имени класса. В случае анонимных внутренних классов в Java мы объявляем и создаем их в одно и то же время. Как правило, они используются всякий раз, когда необходимо переопределить метод класса или интерфейса. Синтаксис анонимного внутреннего класса в Java выглядит следующим образом:

```
   AnonymousInner an_inner = new AnonymousInner() {
      public void my_method() {
         ........
      }
   };
```

Следующая программа показывает, как переопределить метод класса с использованием анонимного внутреннего класса.

```
   abstract class AnonymousInner {
      public abstract void mymethod();
   }
   public class Outer_class {
      public static void main(String args[]) {
         AnonymousInner inner = new AnonymousInner() {
            public void mymethod() {
               System.out.println("Это пример анонимного
внутреннего класса");
            }
         };
         inner.mymethod();
      }
   }
```

В итоге мы увидим на консоли:

```
Это пример анонимного внутреннего класса
```

1.1.5. Анонимный внутренний класс как аргумент

Если метод принимает объект интерфейса, абстрактный класс или конкретный класс, то можно реализовать интерфейс, расширить

абстрактный класс и передать объект методу. Если это класс, можно напрямую передать его методу.

Но во всех трех случаях можно передать анонимный внутренний класс методу. Синтаксис:

```
obj.my_Method(new My_Class() {
   public void Do() {
      .....
   }
});
```

Следующая программа показывает, как передать анонимный внутренний класс в качестве аргумента метода:

```
// Интерфейс
interface Message {
   String greet();
}
public class My_class {
   // Метод, который принимает объект интерфейса Message
   public void displayMessage(Message m) {
      System.out.println(m.greet() +
         ", это пример анонимного внутреннего класса в
качестве аргумента");
   }
   public static void main(String args[]) {
      // Создание класса
      My_class obj = new My_class();
      // Передача анонимного внутреннего класса в качестве
аргумента
      obj.displayMessage(new Message() {
         public String greet() {
            return "Привет";
         }
      });
   }
}
```

Получим следующий результат:

```
Привет, это пример анонимного внутреннего класса в качестве
аргумента
```

1.2. Статический вложенный класс в Java

Статический вложенный класс — это вложенный класс, который является статическим членом внешнего класса. Доступ к нему возможен без создания экземпляра внешнего класса с использованием других статических элементов. Статический вложенный класс не имеет доступа к переменным экземпляра и методам внешнего класса. Синтаксис статического вложенного класса:

```
class MyOuter {
   static class Nested_Demo {
   }
}
```

Создание экземпляра статического вложенного класса немного отличается от экземпляра внутреннего класса:

```
public class Outer {
   static class Nested_Demo {
      public void my_method() {
         System.out.println("Это мой вложенный класс");
      }
   }
   public static void main(String args[]) {
      Outer.Nested_Demo nested = new Outer.Nested_Demo();
      nested.my_method();
   }
}
```

В итоге мы увидим на консоли:

```
Это мой вложенный класс
```

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.
2. Выполните задания практической части лабораторной работы по варианту.

3. Практическая часть

3.1. Задание 1

1. Создать класс **Account** с внутренним классом, с помощью объектов которого можно хранить информацию обо всех операциях со счетом (снятие, платежи, поступления).
2. Создать класс **Зачетная Книжка** с внутренним классом, с помощью объектов которого можно хранить информацию о сессиях, зачетах, экзаменах.
3. Создать класс **Department** с внутренним классом, с помощью объектов которого можно хранить информацию обо всех должностях отдела и сотрудниках, когда-либо занимавших каждую должность.
4. Создать класс **Catalog** с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.
5. Создать класс **City** с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.

6. Создать класс **Mobile** с внутренним классом, с помощью объектов которого можно хранить информацию о моделях телефонов и их свойствах.

7. Создать класс **Художественная Выставка** с внутренним классом, с помощью объектов которого можно хранить информацию о картинах, авторах и времени проведения выставок.

8. Создать класс **Календарь** с внутренним классом, с помощью объектов которого можно хранить информацию о выходных и праздничных днях.

9. Создать класс **Shop** с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услугах.

10. Создать класс **Справочная Служба Общественного Транспорта** с внутренним классом, с помощью объектов которого можно хранить информацию о времени, линиях маршрутов и стоимости проезда.

11. Создать класс **Computer** с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.

12. Создать класс **Park** с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.

13. Создать класс **Cinema** с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени начала сеансов.

14. Создать класс **Программа Передач** с внутренним классом, с помощью объектов которого можно хранить информацию о названии телеканалов и программ.

15. Создать класс **Фильм** с внутренним классом, с помощью объектов которого можно хранить информацию о продолжительности, жанре и режиссерах фильма.

4. Содержание отчета

1. Краткие теоретические сведения о внутренних классах в Java.
2. Код программ.
3. Результаты выполнения программ.
4. Выводы по работе.

5. Контрольные вопросы

1. Что такое внутренние, вложенные и анонимные классы? Как определить вид такого класса? Как создать объекты такого класса?

2. Перечислить возможности доступа к членам внешнего класса, которым наделены вложенные классы?

3. Перечислить возможности доступа к членам внешнего класса, которым наделены внутренние классы?

4. Перечислить возможности доступа к членам внешнего класса, которым наделены анонимные классы?

5. Могут ли классы внутри классов быть базовыми, производными или реализующими интерфейсы?

6. Как решить проблему множественного наследования с применением внутренних классов?

7. Можно ли анонимный класс создать от абстрактного класса?

8. Можно ли анонимный класс создать от **final**-класса?

9. Во что компилируется анонимный внутренний класс в классе? В методе?

10. Можно ли создать анонимный статический внутренний класс?

11. Как получить доступ к внутреннему классу, объявленному внутри метода извне метода?

Занятие № 7
Строки

Цель работы: изучение классов Java, поддерживающих хранение строк, приобретение навыков использования строк в Java-программах.

1. Теоретическая часть

1.1. Введение в строки. Класс String

Для работы со строками в Java определен класс **String**, который предоставляет ряд методов. Физически объект **String** представляет собой ссылку на область в памяти, в которой размещены символы.

Для создания новой строки можно использовать один из конструкторов класса **String**, либо напрямую присвоить строку в двойных кавычках:

```
public static void main(String[] args) {
    String str1 = "Java";
    String str2 = new String(); // пустая строка
    String str3 = new String(new char[] {'h', 'e', 'l', 'l',
'o'});
    String str4 = new String(new char[]{'w', 'e', 'l', 'c',
'o', 'm', 'e'}, 3, 4);//3 -начальный индекс, 4 -кол-во символов
    System.out.println(str1); // Java
    System.out.println(str2); //
    System.out.println(str3); // hello
    System.out.println(str4); // come
}
```

Объект **String** является неизменяемым (*immutable*). При любых операциях над строкой фактически будет создаваться новая строка.

Поскольку строка рассматривается как набор символов, можно применить метод **length()** для нахождения длины строки или длины набора символов:

```
String str1 = "Java";
System.out.println(str1.length()); // 4
```

С помощью метода **toCharArray()** можно преобразовать строку обратно в массив символов:

```
String str1 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
char[] helloArray = str1.toCharArray();
```

Строка может быть пустой. Для этого ей можно присвоить пустые кавычки или удалить из строки все символы:

```
String s = "";   // строка не указывает на объект
if(s.length() == 0) System.out.println("String is empty");
```

Класс **String** имеет специальный метод, который позволяет проверить строку на пустоту - **isEmpty()**. Если строка пуста, он возвращает *true*:

```
String s = "";   // строка не указывает на объект
if(s.length() == 0) System.out.println("String is empty");
```

Переменная **String** может не указывать на какой-либо объект и иметь значение *null*:

```
String s = null;   // строка не указывает на объект
if(s == null) System.out.println("String is null");
```

Значение *null* не эквивалентно пустой строке. Например, в следующем случае мы столкнемся с ошибкой выполнения:

```
String s = null;   // строка не указывает на объект
if(s.length()==0) System.out.println("String is empty");    //
! Ошибка
```

Так как переменная не указывает ни на какой объект **String**, нельзя обращаться к методам объекта **String**, можно проверять строку на *null*:

```
String s = null;   // строка не указывает на объект
if(s!=null && s.length()==0) System.out.println("String is
empty");
```

1.1.1. Основные методы класса String

Основные операции со строками раскрывается через методы класса **String**, среди которых можно выделить следующие:

- **concat()**: объединяет строки
- **valueOf()**: преобразует объект в строковый вид

– **join()**: соединяет строки с учетом разделителя
– **compare()**: сравнивает две строки
– **charAt()**: возвращает символ строки по индексу
– **getChars()**: возвращает группу символов
– **equals()**: сравнивает строки с учетом регистра
– **equalsIgnoreCase()**: сравнивает строки без учета регистра
– **regionMatches()**: сравнивает подстроки в строках
– **indexOf()**: находит индекс первого вхождения подстроки в строку
– **lastIndexOf()**: находит индекс последнего вхождения подстроки в строку
– **startsWith()**: определяет, начинается ли строка с подстроки
– **endsWith()**: определяет, заканчивается ли строка на определенную подстроку
– **replace()**: заменяет в строке одну подстроку на другую
– **trim()**: удаляет начальные и конечные пробелы
– **substring()**: возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса
– **toLowerCase()**: переводит все символы строки в нижний регистр
– **toUpperCase()**: переводит все символы строки в верхний регистр

1.2. Основные операции со строками

1.2.1. Соединение строк

Для соединения строк можно использовать операцию сложения ("+"):

```
String str1 = "Java";
String str2 = "Hello";
String str3 = str1 + " " + str2;
System.out.println(str3); // Hello Java
```

Если в операции сложения строк используется нестроковый объект, то этот объект преобразуется к строке: `String str3 = "Год " + 2020;`.

Фактически при сложении строк с нестроковыми объектами будет вызываться метод **valueOf()** класса **String**, который имеет множество перегрузок и преобразует практически все типы данных к строке. Для преобразования объектов различных классов метод **valueOf** вызывает метод **toString()** этих классов.

Метод **concat()** принимает строку, с которой надо объединить вызывающую строку, и возвращает соединенную строку:

```
String str1 = "Java";
String str2 = "Hello";
str2 = str2.concat(str1); // HelloJava
```

Метод **join()** позволяет объединить строки с учетом разделителя. Например, две подстроки должны быть разделены пробелом:

```
String str1 = "Java";
String str2 = "Hello";
String str3 = String.join(" ", str2, str1); // Hello Java
```

Метод **join** является статическим. Первым параметром идет разделитель подстрок в общей строке, а все последующие параметры передают через запятую произвольный набор объединяемых подстрок.

1.2.2. Извлечение символов и подстрок

Для извлечения символов по индексу в классе **String** определен метод **char charAt(int index)**. Он принимает индекс и возвращает извлеченный символ:

```
String str = "Java";
char c = str.charAt(2);
System.out.println(c); // v
```

Как и в массивах, индексация начинается с нуля.

Если надо извлечь сразу группу символов или подстроку, то можно использовать метод **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**:

```
String str = "Hello world!";
int start = 6;
int end = 11;
char[] dst=new char[end - start];
str.getChars(start, end, dst, 0);
System.out.println(dst); // world
```

Он принимает следующие параметры:

- **srcBegin**: индекс в строке, с которого начинается извлечение символов
- **srcEnd**: индекс в строке, до которого идет извлечение символов
- **dst**: массив символов, в который будут извлекаться символы
- **dstBegin**: индекс в массиве dst, с которого надо добавлять извлеченные из строки символы

1.2.3. Сравнение строк

Для сравнения строк используются методы **equals()** (с учетом регистра) и **equalsIgnoreCase()** (без учета регистра). Оба метода в качестве параметра принимают строку, с которой надо сравнить:

```
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equals(str2)); // false
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

Для сравнения строк не применяется знак равенства ==. Вместо него надо использовать метод **equals()**.

Метод **regionMatches()** сравнивает отдельные подстроки в рамках двух строк:

```
boolean regionMatches(int toffset, String other, int oofset,
int len)
boolean regionMatches(boolean ignoreCase, int toffset, String
other, int oofset, int len)
```

Он принимает следующие параметры:

– **ignoreCase**: надо ли игнорировать регистр символов при сравнении. Если значение *true*, регистр игнорируется

– **toffset**: начальный индекс в вызывающей строке, с которого начнется сравнение

– **other**: строка, с которой сравнивается вызывающая

– **oofset**: начальный индекс в сравниваемой строке, с которого начнется сравнение

– **len**: количество сравниваемых символов в обеих строках

Используем метод **regionMatches()**:

```
String str1 = "Hello world";
String str2 = "I work";
boolean result = str1.regionMatches(6, str2, 2, 3);
System.out.println(result); // true
```

В данном случае метод сравнивает 3 символа с 6-го индекса первой строки ("wor") и 3 символа со 2-го индекса второй строки ("wor"). Так как эти подстроки одинаковы, то возвращается *true*.

Для сравнения двух строк (длины строк) используются методы **int compareTo(String str)** и **int compareToIgnoreCase(String str)**. Если возвращаемое значение больше 0, то первая строка больше, если меньше 0, то вторая, а если 0 - строки равны. Для сравнения длины строк используется лексикографический порядок. Например, строка "A" меньше, чем строка "B". Если первые символы строк равны, то в расчет берутся следующие символы:

```
String str1 = "hello";
String str2 = "world";
String str3 = "hell";
System.out.println(str1.compareTo(str2)); // -15 - str1 меньше
чем strt2
System.out.println(str1.compareTo(str3)); // 1 - str1 больше
чем str3
```

1.2.4. Поиск в строке

Метод **indexOf()** находит индекс первого вхождения подстроки в строку, а метод **lastIndexOf()** - индекс последнего вхождения. Если подстрока не будет найдена, то оба метода возвращают -1:

```
String str = "Hello world";
int index1 = str.indexOf('l'); // 2
int index2 = str.indexOf("wo"); //6
int index3 = str.lastIndexOf('l'); //9
```

Метод **startsWith()** позволяет определить, начинается ли строка с определенной подстроки, а метод **endsWith()** позволяет определить, заканчивается ли строка на определенную подстроку:

```
String str = "myfile.exe";
boolean start = str.startsWith("my"); //true
boolean end = str.endsWith("exe"); //true
```

1.2.5. Замена в строке

Метод **replace()** позволяет заменить в строке одну последовательность символов на другую:

```
String str = "Hello world";
String replStr1 = str.replace('l', 'd'); // Heddo wordd
String replStr2 = str.replace("Hello", "Bye"); // Bye world
```

1.2.6. Обрезка строки

Метод **trim()** позволяет удалить начальные и конечные пробелы:

```
String str = "  hello world  ";
str = str.trim(); // hello world
```

Метод **substring()** возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса:

```
String str = "Hello world";
String substr1 = str.substring(6); // world
String substr2 = str.substring(3,5); //lo
```

1.2.7. Изменение регистра

Метод **toLowerCase()** переводит все символы строки в нижний регистр, а метод **toUpperCase()** - в верхний:

```
String str = "Hello World";
System.out.println(str.toLowerCase()); // hello world
System.out.println(str.toUpperCase()); // HELLO WORLD
```

1.2.8. Split

Метод **split()** позволяет разбить строку на подстроки по определенному разделителю (символ или набор символов передается в качестве параметра в метод). Например, разобьем текст на отдельные слова:

```
String text = "FIFA will never regret it";
String[] words = text.split(" ");
for(String word : words){
    System.out.println(word);
}
```

В данном случае строка будет разделяться по пробелу:

```
FIFA
will
never
regret
it
```

1.3. StringBuffer и StringBuilder

Объекты **String** являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы в Java были добавлены классы **StringBuffer** и **StringBuilder**. Они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

Эти классы похожи. Единственное их различие состоит в том, что класс **StringBuffer** синхронизированный и потокобезопасный, поэтому его удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Класс **StringBuilder** лучше использовать в однопоточных приложениях, он будет работать быстрее, чем **StringBuffer**.

Рассмотрим работу этих классов на примере функциональности **StringBuffer**.

При всех операциях со строками **StringBuffer/StringBuilder** перераспределяет выделенную память. Чтобы избежать слишком частого перераспределения памяти, **StringBuffer/StringBuilder** заранее резервирует некоторую область памяти, которая может использоваться. Конструктор без параметров резервирует в памяти место для 16 символов.

С помощью метода **capacity()** мы можем получить количество символов, для которых зарезервирована память. А с помощью метода **ensureCapacity()** изменить минимальную емкость буфера символов:

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println("Емкость: " + strBuffer.capacity()); // 20
strBuffer.ensureCapacity(32);
System.out.println("Емкость: " + strBuffer.capacity()); // 42
System.out.println("Длина: " + strBuffer.length()); // 4
```

StringBuffer инициализируется строкой "Java", его емкость составляет 4 + 16 = 20 символов. Затем минимальную емкость буфера повышается с помощью вызова **strBuffer.ensureCapacity(32)** до 32 символов. Финальная емкость может отличаться в большую сторону. В данном случае получаем емкость 42 символа. В целях повышения эффективности Java может дополнительно выделять память.

Длина строки, которую можно получить с помощью метода **length()**, не зависит от емкости, в **StringBuffer** остается 4 символа.

Чтобы получить строку, которая хранится в **StringBuffer**, мы можем использовать стандартный метод **toString()**:

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println(strBuffer.toString()); // Java
```

1.3.1. Получение и установка символов

Метод **charAt()** получает, а метод **setCharAt()** устанавливает символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("Java");
char c = strBuffer.charAt(0); // J
System.out.println(c);
strBuffer.setCharAt(0, 'c');
System.out.println(strBuffer.toString()); // cava
```

Метод **getChars()** получает набор символов между определенными индексами:

```
StringBuffer strBuffer = new StringBuffer("world");
int startIndex = 1;
int endIndex = 4;
char[] buffer = new char[endIndex-startIndex];
strBuffer.getChars(startIndex, endIndex, buffer, 0);
System.out.println(buffer); // orl
```

1.3.2. Добавление в строку

Метод **append()** добавляет подстроку в конец **StringBuffer**:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.append(" world");
System.out.println(strBuffer.toString()); // hello world
```

Метод **insert()** добавляет строку или символ по определенному индексу в **StringBuffer**:

```
StringBuffer strBuffer = new StringBuffer("word");
strBuffer.insert(3, 'l');
System.out.println(strBuffer.toString()); //world
strBuffer.insert(0, "s");
System.out.println(strBuffer.toString()); //sworld
```

1.3.3. Удаление символов

Метод **delete()** удаляет все символы с определенного индекса о определенной позиции, а метод **deleteCharAt()** удаляет один символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.delete(0,2);
System.out.println(strBuffer.toString()); //sembler
strBuffer.deleteCharAt(6);
System.out.println(strBuffer.toString()); //semble
```

1.3.4. Обрезка строки

Метод **substring()** обрезает строку с определенного индекса до конца, либо до определенного индекса:

```
StringBuffer strBuffer = new StringBuffer("hello java!");
String str1 = strBuffer.substring(6); // обрезка строки с 6
символа до конца
System.out.println(str1); //java!
String str2 = strBuffer.substring(3, 9); // обрезка строки с 3
по 9 символ
System.out.println(str2); //lo jav
```

1.3.5. Изменение длины

Для изменения длины **StringBuffer** (не емкости буфера символов) применяется метод **setLength()**. При увеличении строка дополняется пустыми символами, при уменьшении - обрезается:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.setLength(10);
System.out.println(strBuffer.toString()); //"hello     "
strBuffer.setLength(4);
System.out.println(strBuffer.toString()); //"hell"
```

1.3.6. Замена в строке

Для замены подстроки между определенными позициями в **StringBuffer** на другую подстроку применяется метод **replace()**:

```
StringBuffer strBuffer = new StringBuffer("hello world!");
strBuffer.replace(6,11,"java");
System.out.println(strBuffer.toString()); //hello java!
```

Первый параметр метода **replace** указывает, с какой позиции надо начать замену, второй параметр - до какой позиции, а третий параметр указывает на подстроку замены.

1.3.7. Обратный порядок в строке

Метод **reverse()** меняет порядок в **StringBuffer** на обратный:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.reverse();
System.out.println(strBuffer.toString()); //relbmessa
```

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.

2. Выполните задания практической части лабораторной работы по варианту.

3. Практическая часть

3.1. Задание 1

1. В каждом слове текста k-ю букву заменить заданным символом. Если k больше длины слова, корректировку не выполнять.

2. В тексте каждую букву заменить ее порядковым номером в алфавите. При выводе в одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.

3. В тексте после буквы Р, если она не последняя в слове, ошибочно напечатана буква А вместо О. Внести исправления в текст.

4. В тексте после k-го символа вставить заданную подстроку.

5. После каждого слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.

6. В зависимости от признака (0 или 1) в каждой строке текста удалить указанный символ везде, где он встречается, или вставить его после k-го символа.

7. Из текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.

8. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например, между скобками «(» и «)» или между звездочками «*» и т.п.).

9. Определить, сколько раз повторяется в тексте каждое слово.

10. В тексте найти и напечатать n символов (и их количество), встречающихся наиболее часто.

11. Найти, гласных или согласных букв больше в каждом предложении текста.

12. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.

13. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.

14. В тексте найти и напечатать все слова максимальной и минимальной длины.

15. В стихотворении найти одинаковые буквы, встречающиеся во всех словах.

3.2. Задание 2

1. В тексте найти первую подстроку максимальной длины, не содержащую букв.
2. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.
3. Преобразовать текст так, чтобы каждое слово, не содержащее неалфавитных символов, начиналось с заглавной буквы.
4. Подсчитать количество знаков препинания в тексте.
5. В заданном тексте найти сумму всех встречающихся цифр.
6. Все слова текста встречаются четное количество раз, за исключением одного. Определить это слово. Регистр не учитывать.
7. Определить сумму всех целых чисел, встречающихся в тексте.
8. Из текста удалить лишние пробелы (если они разделяют два различных знака препинания или рядом два пробела).
9. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции n до m.
10. Определить количество вхождений заданного слова в текст, без учета регистра, считая буквы «е»/«ё» и «и»/«й» одинаковыми.
11. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
12. Заменить все одинаковые рядом стоящие в тексте символы одним символом.
13. Вывести слова из заданного текста в алфавитном порядке.
14. Подсчитать, сколько слов в тексте начинается с заглавной буквы.
15. Подсчитать, сколько раз заданное слово входит в текст.

3.3. Задание 3

Создать программу обработки текста учебника по программированию с использованием классов: *Символ, Слово, Предложение, Абзац, Лексема, Листинг, Знак препинания* и др. Во всех задачах с формированием текста заменять табуляции и последовательности пробелов одним пробелом.

Предварительно текст следует разобрать на составные части, выполнить задание и вывести полученный результат.

1. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
2. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.

3. Найти такое слово в первом предложении, которого нет ни в одном из остальных предложений.

4. Во всех вопросительных предложениях текста найти и напечатать без повторений слова заданной длины.

5. В каждом предложении текста поменять местами первое слово с последним, не изменяя длины предложения.

6. Напечатать слова текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.

7. Рассортировать слова текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).

8. Слова текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.

9. Все слова текста рассортировать по возрастанию количества заданной буквы в слове и равные расположить в алфавитном порядке.

10. Существует текст и список слов. Для каждого слова из списка найти, сколько раз оно встречается в каждом предложении, рассортировать слова по убыванию общего количества вхождений.

11. В каждом предложении текста исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.

12. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.

13. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства — по алфавиту.

14. В тексте найти подстроку-палиндром максимальной длины, т.е. читающуюся слева направо и справа налево одинаково.

15. Преобразовать каждое слово в тексте, удалив из него все следующие (предыдущие) вхождения первой (последней) буквы слова.

3.4. Задание 4

Разработать проект управления процессами на основе создания и реализации интерфейсов для следующих предметных областей:

1. Проверить, является ли строка сильным паролем. Пароль считается сильным, если его длина больше либо равна 10 символам, он содержит как минимум одну цифру, одну букву в верхнем и одну букву в нижнем регистре. Пароль может содержать только латинские буквы и/или цифры, а также символ «_».

2. Текст из n^2 символов шифруется по следующему правилу: — все символы текста записываются в квадратную таблицу размерности n в порядке слева направо, сверху вниз; — таблица поворачивается на 90° по часовой стрелке; — 1-я строка таблицы меняется местами с последней, 2-я — с предпоследней и т.д.; — 1-й столбец таблицы меняется местами со 2-

м, 3-й — с 4-м и т.д.; — зашифрованный текст получается в результате обхода результирующей таблицы по спирали по часовой стрелке, начиная с левого верхнего угла. Зашифровать текст по указанному правилу.

3. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.

4. Вычеркнуть из текста минимальное количество предложений так, чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.

5. Осуществить сжатие английского текста, заменив каждую группу из двух или более рядом стоящих символов на один символ, за которым следует количество его вхождений в группу. К примеру, строка hellowoooorld должна сжиматься в hel2owo4rld.

6. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы «?» (произвольный символ) и «*» (произвольное количество произвольных символов).

7. Буквенная запись телефонных номеров основана на том, что каждой цифре соответствует несколько английских букв: 2 — ABC, 3 — DEF, 4 — GHI, 5 — JKL, 6 — MNO, 7 — PQRS, 8 — TUV, 9 — WXYZ. Написать программу, которая находит в заданном телефонном номере подстроку максимальной длины, соответствующую слову из словаря.

8. Осуществить форматирование заданного текста с выравниванием по левому краю. Программа должна разбивать текст на строки с длиной, не превосходящей заданного количества символов. Если слово не помещается в строке, его нужно переносить на следующую.

9. Пусть текст содержит миллион символов, необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Ускорить процесс, используя класс StringBuilder.

10. Алгоритм Барроуза-Уиллера для сжатия текстов основывается на преобразовании: для слова рассматриваются все его циклические сдвиги, которые затем сортируются в алфавитном порядке, после чего формируется слово из последних символов отсортированных циклических сдвигов. К примеру, для слова JAVA циклические сдвиги — это JAVA, AVAJ, VAJA, AJAV. После сортировки по алфавиту получим AJAV, AVAJ, JAVA, VAJA. Значит, результат преобразования — слово VJAA. Реализовать программно преобразование Барроуза-Уиллера для данного слова.

11. Восстановить слово по его преобразованию Барроуза-Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.

12. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква

каждого слова совпадала с первой буквой следующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.

13. Зашифровать текст по следующему правилу: из исходного текста выбирается 1, 4, 7, 10-й и т.д. (до конца текста) символы, затем 2, 5, 8, 11-й и т.д. (до конца текста) символы, затем 3, 6, 9, 12-й и т.д.

14. В предложении из n слов первое слово поставить на место второго, второе — на место третьего и т.д., (n-1)-е слово — на место n-го, n-е слово поставить на место первого. В исходном и преобразованном предложениях между словами должны быть или один пробел, или знак препинания и один пробел.

15. Все слова текста рассортировать в порядке убывания их длин, при этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.

4. Содержание отчета

1. Краткие теоретические сведения о строках в Java.
2. Код программ.
3. Результаты выполнения программ.
4. Выводы по работе.

5. Контрольные вопросы

1. Как создать объект класса **String**, какие существуют конструкторы класса **String**? Что такое строковый литерал? Что значит «упрощенное создание объекта **String**»?
2. Как реализован класс **String**, какие поля в нем объявлены?
3. Как работает метод **substring()** класса **String**?
4. Можно ли изменить состояние объекта типа **String**? Что происходит при попытке изменения состояния объекта типа **String**? Можно ли наследоваться от класса **String**? Почему строковые объекты *immutable*?
5. Что такое пул литералов? Как строки заносятся в пул литералов? Как занести строку в пул литералов и как получить ссылку на строку, хранящуюся в пуле литералов? В каком отделе памяти хранится пул литералов в Java 1.6 и Java 1.7?
6. В чем отличие объектов классов **StringBuilder** и **StringBuffer** от объектов класса **String**? Какой из этих классов потокобезопасный?
7. Как необходимо сравнивать на равенство объекты классов **StringBuilder** и **StringBuffer** и почему?
8. Что такое **Unicode**? Что такое *code point*? Отличия UTF-8 от UTF-16.
9. Как кодируется символ согласно кодировке UTF-8, UTF-16 и UTF-32?

10. Что такое кодировка? Какие кодировки вы знаете? Как создать строки в различной кодировке?

11. Какие методы класса **String** используются для работы с кодовыми точками? Когда следует их использовать?

12. Что представляет собой регулярное выражение? Что такое метасимволы регулярного выражения? Какие существуют классы символов регулярных выражений? Что такое квантификаторы? Какие существуют логические операторы регулярных выражений?

13. Какие классы Java работают с регулярными выражениями? В каком пакете они расположены?

14. Что такое группы в регулярных выражениях? Как нумеруются группы? Что представляет собой группа номер «0»?

15. Что такое интернационализация и локализация?

16. Что представляет собой локаль в программе? Назначение объектов класса **Locale**? Как получить локаль? Как узнать, какие локали доступны?

17. Какую информацию можно локализовать автоматически, применяя объект класса **Locale**? Как работают классы **NumberFormat** и **DateFormat**?

18. Как можно локализовать приложение, используя класс **ResourceBundle**? Для каких еще целей, кроме локализации, можно применять объекты этого класса?

Занятие № 8
Исключения и ошибки

Цель работы: изучение исключений и ошибок в Java, приобретение навыков обработки исключений в Java-программах.

1. Теоретическая часть

1.1. Исключения в Java

Исключение в Java представляет проблему, которая возникает в ходе выполнения программы. В случае возникновения в Java исключения (*exception*), или исключительного события, нормальное течение программы прекращается, и программа/приложение завершаются в аварийном режиме, что не является рекомендованным, и, как следствие, подобные случаи требуют в Java обработку исключений.

1.1.1. Причины возникновения исключения

Существует множество причин, которые могут повлечь за собой возникновение исключения. Например, файл, который необходимо открыть, не найден; пользователь ввел недопустимые данные; соединение с сетью потеряно в процессе передачи данных либо JVM

исчерпала имеющийся объем памяти. Некоторые из данных исключений вызваны пользовательской ошибкой, другие — программной ошибкой, в некоторых случаях, причиной тому может послужить сбой в материальных ресурсах.

Выделяют три типа исключений:

- **Контролируемые исключения** — исключения периода компиляции. Данные исключения не следует игнорировать.

К примеру, если используется класс **FileReader** для считывания данных из файла, а указанный файл не существует, происходит **FileNotFoundException**, и компилятор подсказывает программисту обработку данного исключения:

```
import java.io.File;
import java.io.FileReader;
public class Test {
   public static void main(String args[]) {
      File f = new File("D://java/file.txt");
      FileReader fr = new FileReader(f);
   }
}
```

При попытке компиляции программы будут выведены следующие исключения:

```
C:\>javac Test.java
Test.java:8: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
      FileReader fr = new FileReader(f);
                      ^
1 error
```

Примечание. Из-за того, что методы **read()** и **close()** класса **FileReader** вызывают **IOException**, компилятор может уведомить вас об обработке **IOException**, совместно с **FileNotFoundException**.

- **Неконтролируемые исключения** — исключения на этапе выполнения, в ходе компиляции игнорируются. Например: логические ошибки, неверный способ использования API.

Например, в программе объявлен массив из 5 элементов, попытка вызова 6-го элемента массива повлечет за собой возникновение **ArrayIndexOutOfBoundsExceptionexception**:

```
public class Test {
   public static void main(String args[]) {
      int array[] = {1, 2, 3};
      System.out.println(array[4]);
   }
}
```

При компиляции и выполнении программы будет получено следующее исключение:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Test.main(Test.java:8)
```

- **Ошибки** — не являются исключениями, однако представляют проблемы, которые возникают независимо от пользователя либо программиста. Ошибки в коде обычно игнорируются, в том числе на этапе компиляции. К примеру, ошибка при переполнении стека.

Иерархия исключений представлена на рис. 2:

Рисунок 2 – Иерархия исключений

1.1.2. Методы исключений

В табл. 1 представлен список важных методов, доступных в классе **Throwable**.

Таблица 1 – Методы класса Throwable

№	Метод и описание
1	**public String getMessage()** Возврат подробного сообщения о произошедшем исключении. Инициализация сообщения производится в конструкторе Throwable.
2	**public Throwable getCause()** Возврат причины исключения, представленной объектом Throwable.
3	**public String toString()** Возврат имени класса, соединенного с результатом getMessage().
4	**public void printStackTrace()** Выведение результата toString() совместно с трассировкой стека в System.err, поток вывода ошибок.
5	**public StackTraceElement [] getStackTrace()** Возврат массива, содержащего каждый элемент в трассировке стека. Элемент с номером 0 представляет вершину стека вызовов, последний

	элемент массива отображает метод на дне стека вызовов.
6	**public Throwable fillInStackTrace()** Заполняет трассировку стека данного объекта Throwable текущей трассировкой стека, дополняя какую-либо предшествующую информацию в трассировке стека.

1.2. Обработка исключений

1.2.1. try и catch

Метод производит обработку исключения при использовании ключевых слов **try** и **catch**. Блок **try/catch** размещается в начале и конце кода, который может сгенерировать исключение. Код в составе блока **try/catch** является защищенным кодом. Синтаксис:

```
try {
   // Защищенный код
}catch(НазваниеИсключения e1) {
   // Блок catch
}
```

Код, предрасположенный к исключениям, размещается в блоке **try**. В случае возникновения исключения, его обработка будет производиться соответствующим блоком **catch**. За каждым блоком **try** должен немедленно следовать блок **catch** либо блок **finally**.

Оператор **catch** включает объявление типа исключения, которое предстоит обработать. При возникновении исключения в защищенном коде, блок **catch**, следующий за **try**, будет проверен. В случае, когда тип произошедшего исключения представлен в блоке **catch**, исключение передается в блок **catch** аналогично тому, как аргумент передается в параметр метода.

Например, представлен массив с заявленными двумя элементами. Попытка кода получить доступ к третьему элементу массива повлечет за собой генерацию исключения:

```
import java.io.*;
public class Test {
   public static void main(String args[]) {
      try {
         int array[] = new int[2];
         System.out.println("Доступ к третьему элементу:" +
array[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Исключение:" + e);
      }
      System.out.println("Вне блока");
   }
}
```

Будет получен следующий результат:

```
Исключение:java.lang.ArrayIndexOutOfBoundsException: 3
Вне блока
```

1.2.2. Многократные блоки catch

За блоком **try** могут следовать несколько блоков **catch**. Синтаксис:

```
try {
   // Защищенный код
}catch(ИсключениеТип1 e1) {
   // Блок catch
}catch(ИсключениеТип2 e2) {
   // Блок catch
}
```

В случае возникновения исключения в защищенном коде, исключение выводится в первый блок **catch** в списке. Если тип данных генерируемого исключения совпадает с **ИсключениеТип1**, он перехватывается в указанной области. В обратном случае исключение переходит к следующему оператору **catch**, и так до тех пор, пока не будет произведен перехват исключения, либо оно не пройдет через все операторы, в случае чего выполнение текущего метода будет прекращено, и исключение будет перенесено к предшествующему методу в стеке вызовов. Например:

```
try {
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException e1) {
   e1.printStackTrace();
   return -1;
}catch(FileNotFoundException e2) // Недействительно! {
   e2.printStackTrace();
   return -1;
}
```

1.2.3. Перехват многотипных исключений

Можно произвести обработку более чем одного исключения при использовании одного блока **catch**, данное свойство упрощает код:

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

1.2.4. Ключевые слова throws/throw

Если метод не может осуществить обработку контролируемого исключения, производится соответствующее уведомление при использовании ключевого слова **throws** в конце сигнатуры метода.

При использовании ключевого слова **throw** можно произвести обработку вновь выявленного исключения либо исключения, которое было только что перехвачено.

Следует внимательно различать ключевые слова **throw** и **throws** в Java, так как **throws** используется для отложенной обработки контролируемого исключения, а **throw**, в свою очередь, используется для вызова заданного исключения.

Представленный ниже метод отображает, что им генерируется **RemoteException**:

```
import java.rmi.RemoteException;
public class Test {
   public void deposit(double amount) throws RemoteException {
      // Реализация метода
      throw new RemoteException();
   }
   // Остаток определения класса
}
```

Метод также может объявить о том, что им генерируется более чем одно исключение, в виде перечня отделенных друг от друга запятыми исключений. К примеру, следующий метод оповещает о том, что им генерируются **RemoteException** и **InsufficientFundsException**:

```
import java.rmi.RemoteException;
public class Test {
   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Реализация метода
   }
   // Остаток определения класса
}
```

1.2.5. Блок finally

В Java **finally** следует за блоком **try** либо блоком **catch**. Блок **finally** в коде выполняется всегда независимо от наличия исключения.

Использование блока **finally** позволяет запустить какой-либо оператор, предназначенный для очистки.

Синтаксис блока **finally**:

```
try {
   // Защищенный код
}catch(ИсключениеТип1 e1) {
   // Блок catch
}catch(ИсключениеТип2 e2) {
   // Блок catch
finally {
   // Блок finally всегда выполняется.
}
```

Например:

```
public class Test {
   public static void main(String args[]) {
      int array[] = new int[2];
      try {
         System.out.println("Доступ к третьему элементу:" +
array[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Исключение:" + e);
      }finally {
         array[0] = 6;
         System.out.println("Значение первого элемента: " +
array[0]);
         System.out.println("Оператор finally выполнен.");
      }
   }
}
```

Будет получен следующий результат:

```
Исключение:java.lang.ArrayIndexOutOfBoundsException: 3
Значение первого элемента: 6
Оператор finally выполнен.
```

Следует помнить, что:

– выражение **catch** не может существовать без оператора **try**;

– при наличии блока **try/catch**, выражение **finally** не является обязательным;

– блок **try** не может существовать при отсутствии выражения **catch** либо выражения **finally**;

– существование какого-либо кода в промежутке между блоками **try**, **catch**, **finally** является невозможным.

1.2.6. Конструкция try-with-resources

При использовании различных видов ресурсов (потоки, соединения и др.), нам предстоит закрыть их при использовании блока **finally**.

Например, производится считывание данных из файла при использовании **FileReader**, после чего он закрывается блоком **finally**:

```
import java.io.FileReader;
import java.io.File;
import java.io.IOException;
public class Test {
   public static void main(String args[]) {
      FileReader fr = null;
      try {
         File f = new File("file.txt");
         fr = new FileReader(f);
```

```
         char [] array = new char[10];
         fr.read(array);   // чтение содержимого массива
         for(char c : array)
         System.out.print(c);     // вывод символов на экран,
один за одним
      }catch(IOException e1) {
         e1.printStackTrace();
      }finally {
         try {
            fr.close();
         }catch(IOException e2) {
            e2.printStackTrace();
         }
      }
   }
}
```

Конструкция **try-with-resources**, также именуемая как **автоматическое управление ресурсами**, представляет новый механизм обработки исключений, который был представлен в 7-ой версии Java, осуществляя автоматическое закрытие всех ресурсов, используемых в рамках блока **try/catch.**

Чтобы воспользоваться данным оператором, нужно разместить заданные ресурсы в круглых скобках, после чего созданный ресурс будет автоматически закрыт по окончании блока. Синтаксис:

```
try(FileReader fr = new FileReader("Путь к файлу")) {
   // использование ресурса
   }catch() {
      // тело catch
   }
}
```

Например, реализуем считывание данных в файле:

```
import java.io.FileReader;
import java.io.IOException;
public class Test {
   public static void main(String args[]) {
      try(FileReader  fr  =  new  FileReader("E://Soft/NetBeans
8.2/Projects/test/test/file.txt")) {
         char [] array = new char[10];
         fr.read(array);   // чтение содержимого массива
         for(char c : array)
         System.out.print(c);     // вывод символов на экран,
один за одним
      }catch(IOException e) {
         e.printStackTrace();
      }
   }
}
```

При работе с **try-with-resources** следует помнить следующие нюансы:

– с целью использования конструкции **try-with-resources** следует реализовать интерфейс **AutoCloseable**, после чего соответствующий метод **close()** будет вызван автоматически во время выполнения;

– в конструкции **try-with-resources** возможно указание классов;

– при указании нескольких классов в блоке **try** конструкции **try-with-resources**, закрытие классов будет производиться в обратном порядке;

– за исключением внесения ресурсов в скобки, все элементы являются равными аналогично нормальному блоку **try/catch** в составе блока **try**;

– ресурсы, внесенные в **try**, конкретизируются до запуска блока **try**;

– ресурсы в составе блока **try** указываются как окончательные.

1.3. Создание своих собственных исключений

Можно создать свои собственные исключения в среде Java. При записи собственных классов исключений следует принимать во внимание следующие аспекты:

– все исключения должны быть дочерними элементами **Throwable**;

– если вы планируете произвести запись контролируемого исключения с автоматическим использованием за счет правила обработки или объявления, вам следует расширить класс **Exception**;

– если вы хотите произвести запись исключения на этапе выполнения, вам следует расширить класс **RuntimeException**.

Определение собственного класс исключений:

```
class MyException extends Exception {
}
```

Необходимо расширить предопределенный класс **Exception** с целью создания собственного контролируемого исключения. Следующий класс **InsufficientFundsException** исключительных ситуаций, определяемых пользователем, расширяет класс **Exception**, делая его контролируемым исключением. Класс исключений, подобно всем остальным классам, содержит используемые области и методы:

```
import java.io.*;
public class InsufficientFundsException extends Exception {
   private double amount;
   public InsufficientFundsException(double amount) {
      this.amount = amount;
   }
   public double getAmount() {
      return amount;
   }
}
```

С целью демонстрации наших исключений, определяемых пользователем, следующий класс **Checking** содержит метод **withdraw()**, генерирующий **InsufficientFundsException**:

```
import java.io.*;
public class Checking {
   private int number;
   private double balance;
   public Checking(int number) {
      this.number = number;
   }
   public void deposit(double amount) {
      balance += amount;
   }
   public void withdraw(double amount) throws
InsufficientFundsException {
      if(amount <= balance) {
         balance -= amount;
      }else {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance() {
      return balance;
   }
   public int getNumber() {
      return number;
   }
}
```

Следующий пример демонстрирует вызов методов **deposit()** и **withdraw()** класса **Checking**:

```
public class Bank {
   public static void main(String [] args) {
      Checking c = new Checking(101);
      System.out.println("Депозит $300...");
      c.deposit(300.00);
      try {
         System.out.println("\nСнятие $100...");
         c.withdraw(100.00);
         System.out.println("\nСнятие $400...");
         c.withdraw(400.00);
      }catch(InsufficientFundsException e) {
         System.out.println("Извините, но у Вас $" +
e.getAmount());
         e.printStackTrace();
      }
   }
}
```

Вследствие компиляции 3-х последних программ при запуске последней будет получен следующий результат:

```
Депозит $300...

Снятие $100...

Снятие $400...
Извините, но у Вас $200.0
InsufficientFundsException
        at Checking.withdraw(Checking.java:25)
        at Bank.main(Bank.java:13)
```

1.4. Общие исключения

В Java можно выделить две категории исключений и ошибок.

- **Исключения JVM** — данная группа представлена исключениями/ошибками, которые вызываются непосредственно и логически со стороны JVM. Примеры: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
- **Программные исключения** — данные исключения вызываются непосредственно приложением либо программистами API. Примеры: IllegalArgumentException, IllegalStateException.

1.5. Встроенные исключения

Java определяет несколько классов исключений внутри стандартного пакета *java.lang*. Наиболее общие из этих исключений являются подклассами стандартного типа **RuntimeException**. Поскольку *java.lang* неявно импортируется во все java-программы, то большинство исключений, полученных из **RuntimeException**, автоматические. Java определяет несколько других типов исключений, которые относятся к его различным библиотекам класса.

В табл. 2 представлен список неконтролируемых исключений на этапе выполнения (*Unchecked RuntimeExceptions*).

Таблица 2 – Unchecked RuntimeExceptions

№	Метод и описание
1	**java.lang.ArithmeticException** Арифметическая ошибка, например, деление на ноль.
2	**java.lang.ArrayIndexOutOfBoundsException** Индекс массива выходит за пределы.
3	**java.lang.ArrayStoreException** Присвоение элементу массива несовместимого типа.
4	**java.lang.ClassCastException** Недопустимое приведение типов.
5	**java.lang.IllegalArgumentException** Недопустимый аргумент, используемый для вызова метода.

6	**java.lang.IllegalMonitorStateException** Недопустимая работа монитора, например, ожидание разблокированного потока.
7	**java.lang.IllegalStateException** Окружающая обстановка или приложение находится в неправильном состоянии.
8	**java.lang.IllegalThreadStateException** Запрошенная операция несовместима с текущим состоянием потока.
9	**java.lang.IndexOutOfBoundsException** Некоторый тип индекса находится за пределом.
10	**java.lang.NegativeArraySizeException** Массив создан с отрицательным размером.
11	**java.lang.NullPointerException** Недопустимое использование нулевой ссылки.
12	**java.lang.NumberFormatException** Неверное преобразование строки в числовой формат.
13	**java.lang.SecurityException** Попытка нарушить безопасность.
14	**java.lang.StringIndexOutOfBounds** Попытка индексирования за пределами строки.
15	**java.lang.UnsupportedOperationException** Была обнаружена неподдерживаемая операция.

В табл. 3 представлен список контролируемых исключений (*Checked Exceptions*) в Java, определенных в *java.lang*.

Таблица 3 – Checked Exceptions

№	Метод и описание
1	**java.lang.ClassNotFoundException** Класс не найден.
2	**java.lang.CloneNotSupportedException** Попытка клонировать объект, который не реализует Cloneable интерфейс.
3	**java.lang.IllegalAccessException** Запрещен доступ к классу.
4	**java.lang.InstantiationException** Попытка создать объект абстрактного класса или интерфейса.
5	**java.lang.InterruptedException** Один поток был прерван другим потоком.
6	**java.lang.NoSuchFieldException** Запрошенное поле не существует.
7	**java.lang.NoSuchMethodException** Запрошенный метод не существует.

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.

2. Выполните задания практической части лабораторной работы.

3. Практическая часть

3.1. Задание 1

В символьном файле находится информация об N числах с плавающей запятой с указанием локали каждого числа отдельно. Прочитать информацию из файла. Проверить на корректность, то есть являются ли числа числами. Преобразовать к числовым значениям и вычислить сумму и среднее значение прочитанных чисел. Создать собственный класс исключения. Предусмотреть обработку исключений, возникающих при нехватке памяти, отсутствии самого файла по заданному адресу, отсутствии или некорректности требуемой записи в файле, недопустимом значении числа (выходящим за пределы максимально допустимых значений) и т.д.

4. Содержание отчета

1. Краткие теоретические сведения об исключениях и ошибках в Java.
2. Код программ.
3. Результаты выполнения программ.
4. Выводы по работе.

5. Контрольные вопросы

1. Что для программы является исключительной ситуацией? Какие существуют способы обработки ошибок в программах?
2. Что такое исключение для Java-программы? Что значит «программа генерировала\выбросила исключение»? Привести пример, когда исключения генерируются виртуальной машиной (автоматически) и когда необходимо их генерировать вручную.
3. Привести иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?
4. Объяснить работу оператора **try-catch-finally**. Когда данный оператор следует использовать? Сколько блоков **catch** может соответствовать одному блоку **try**?
5. Можно ли вкладывать блоки **try** друг в друга, можно ли вложить блок **try** в **catch** или **finally**? Как происходит обработка исключений внутреннего блока **try**, если среди его блоков **catch** нет подходящего?
6. Что называют стеком операторов **try**? Как работает блок **try** с ресурсами?
7. Указать правило расположения блоков **catch** в зависимости от типов перехватываемых исключений. Может ли перехваченное исключение быть сгенерировано снова, и, если да, то как и кто в этом

случае будет обрабатывать повторно сгенерированное исключение? Может ли блок **catch** выбрасывать иные исключения, и если да, то привести пример, когда это может быть необходимо.

8. Когда происходит вызов блока **finally**? Существуют ли ситуации, когда блок **finally** не будет вызван? Может ли блок **finally** выбрасывать исключения? Может ли блок **finally** выполниться дважды?

9. Как генерировать исключение вручную? Объекты каких классов могут быть генерированы в качестве исключений? Можно ли генерировать два исключения одновременно?

10. Операторы **throw** и **throws**: отличия и принципы работы.

11. Объяснить правила реализации секции **throws** при переопределении метода и при описании конструкторов производного класса.

12. Как ведет себя блок **throws** при работе с проверяемыми и непроверяемыми исключениями?

13. Каков будет результат создания объекта, если конструктор при работе сгенерирует исключительную ситуацию?

14. Нужно ли генерировать исключения, входящие в Java SE? Как создать собственные классы исключений?

Занятие № 9
Коллекции

Цель работы: изучение коллекций Java, реализующих различные алгоритмы и структуры данных, приобретение навыков использования коллекций в Java-программах.

1. Теоретическая часть

1.1. Введение в коллекции

1.1.1. Типы коллекций

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, они имеют фиксированную длину. Эту проблему в Java решают ***коллекции***. Классы коллекций реализуют различные алгоритмы и структуры данных, такие как стек, очередь, дерево и ряд других. Классы коллекций располагаются в пакете *java.util*, поэтому перед применением коллекций следует подключить данный пакет.

В Java существует множество коллекций, все они образуют стройную и логичную систему. В основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

– **Collection**: базовый интерфейс для всех коллекций и других интерфейсов коллекций;

– **Queue**: наследует интерфейс **Collection** и представляет функционал для структур данных в виде очереди;
– **Deque**: наследует интерфейс **Queue** и представляет функционал для двунаправленных очередей;
– **List**: наследует интерфейс **Collection** и представляет функциональность простых списков;
– **Set**: также расширяет интерфейс **Collection** и используется для хранения множеств уникальных объектов;
– **SortedSet**: расширяет интерфейс **Set** для создания сортированных коллекций;
– **NavigableSet**: расширяет интерфейс **SortedSet** для создания коллекций, в которых можно осуществлять поиск по соответствию;
– **Map**: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. Не наследуется от интерфейса **Collection.**

Эти интерфейсы частично реализуются абстрактными классами:
– **AbstractCollection**: базовый абстрактный класс для других коллекций, который применяет интерфейс **Collection;**
– **AbstractList**: расширяет класс **AbstractCollection** и применяет интерфейс **List**, предназначен для создания коллекций в виде списков;
– **AbstractSet**: расширяет класс **AbstractCollection** и применяет интерфейс **Set** для создания коллекций в виде множеств;
– **AbstractQueue**: расширяет класс **AbstractCollection** и применяет интерфейс **Queue**, предназначен для создания коллекций в виде очередей и стеков;
– **AbstractSequentialList**: также расширяет класс **AbstractList** и реализует интерфейс **List**. Используется для создания связанных списков;
– **AbstractMap**: применяет интерфейс **Map**, предназначен для создания наборов по типу словаря с объектами в виде пары "ключ-значение".

С помощью интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:
– **ArrayList**: простой список объектов;
– **LinkedList**: представляет связанный список;
– **ArrayDeque**: класс двунаправленной очереди, в которой можно произвести вставку и удаление, как в начале коллекции, так и в ее конце;
– **HashSet**: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код;
– **TreeSet**: набор отсортированных объектов в виде дерева;
– **LinkedHashSet**: связанное хеш-множество;

– **PriorityQueue**: очередь приоритетов;
– **HashMap**: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение;
– **TreeMap**: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение.

1.1.2. Интерфейс Collection

Интерфейс **Collection** является базовым для всех коллекций, определяя основной функционал:

```
public interface Collection<E> extends Iterable<E>{
    // определения методов
}
```

Интерфейс **Collection** является обобщенным и расширяет интерфейс **Iterable**, поэтому все объекты коллекций можно перебирать в цикле по типу **for-each.**

Среди методов интерфейса **Collection** можно выделить следующие:
– **boolean add (E item)**: добавляет в коллекцию объект **item**. При удачном добавлении возвращает *true*, при неудачном - *false*
– **boolean addAll (Collection<? extends E> col)**: добавляет в коллекцию все элементы из коллекции **col**. При удачном добавлении возвращает *true*, при неудачном - *false*
– **void clear ()**: удаляет все элементы из коллекции
– **boolean contains (Object item)**: возвращает *true*, если объект **item** содержится в коллекции, иначе - *false*
– **boolean isEmpty ()**: возвращает *true*, если коллекция пуста, иначе - *false*
– **Iterator<E> iterator ()**: возвращает объект **Iterator** для обхода элементов коллекции
– **boolean remove (Object item)**: возвращает *true*, если объект **item** удачно удален из коллекции, иначе - *false*
– **boolean removeAll (Collection<?> col)**: удаляет все объекты коллекции **col** из текущей коллекции. Если текущая коллекция изменилась, возвращает *true*, иначе - *false*
– **boolean retainAll (Collection<?> col)**: удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции **col**. Если коллекция изменилась, возвращает *true*, иначе - *false*
– **int size ()**: возвращает число элементов в коллекции
– **Object[] toArray ()**: возвращает массив, содержащий все элементы коллекции

Все эти и остальные методы, которые имеются в интерфейсе **Collection**, реализуются всеми коллекциями.

1.2. Класс ArrayList и интерфейс List

Для создания простых списков применяется интерфейс **List**, который расширяет функциональность интерфейса **Collection**. Класс **ArrayList** представляет обобщенную коллекцию, которая наследует свою функциональность от класса **AbstractList** и применяет интерфейс **List**, т.е. **ArrayList** представляет простой список, аналогичный массиву, только количество элементов в нем не фиксировано.

Используем класс **ArrayList** и некоторые его методы в программе:

```
import java.util.ArrayList;
public class Program{
    public static void main(String[] args) {
        ArrayList<String> people = new ArrayList<String>();
        // добавим в список ряд элементов
        people.add("Tom");
        people.add("Alice");
        people.add("Kate");
        people.add("Sam");
        people.add(1, "Bob"); // добавляем элемент по индексу 1
        System.out.println(people.get(1));// получаем 2-й
объект
        people.set(1, "Robert"); // установка нового значения
для 2-го объекта
        System.out.printf("ArrayList has %d elements \n",
people.size());
        for(String person : people){
            System.out.println(person);
        }
        // проверяем наличие элемента
        if(people.contains("Tom")){
            System.out.println("ArrayList contains Tom");
        }
        // удалим несколько объектов
        people.remove("Robert"); // удаление конкретного
элемента
        people.remove(0); // удаление по индексу
        Object[] peopleArray = people.toArray();
        for(Object person : peopleArray){
            System.out.println(person);
        }
    }
}
```

Консольный вывод программы:

```
Bob
ArrayList has 5 elements
Tom
Robert
Alice
```

```
Kate
Sam
ArrayList contains Tom
Alice
Kate
Sam
```

Объект **ArrayList** типизируется классом **String**, поэтому будет список строк. С помощью метода **toArray()** можно преобразовать список в массив объектов. В объект **ArrayList** можно свободно добавлять объекты, в отличие от массива, в реальности **ArrayList** использует для хранения объектов массив для 10 объектов. Если в процессе добавляется больше, то создается новый массив, который может вместить все. Такие перераспределения памяти уменьшают производительность, поэтому можно сразу установить количество объектов: `ArrayList<String>people=newArrayList<String>(25);` либо с помощью метода: `people.ensureCapacity(25);`.

1.3. Очереди и класс ArrayDeque

Очереди представляют структуру данных, работающую по принципу ***FIFO*** (first in - first out). Чем раньше элемент был добавлен в коллекцию, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди. Особенностью классов очередей является то, что они реализуют специальные интерфейсы **Queue** или **Deque**.

1.3.1. Интерфейс Queue

Обобщенный интерфейс **Queue** расширяет базовый интерфейс **Collection** и определяет поведение класса в качестве однонаправленной очереди.

У всех классов, которые реализуют данный интерфейс, будет метод **offer** для добавления в очередь, метод **poll** для извлечения элемента из головы очереди, и методы **peek** и **element,** позволяющие получить элемент из головы очереди.

1.3.2. Интерфейс Deque

Интерфейс **Deque** расширяет интерфейс **Queue** и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу ***LIFO*** (последний вошел - первый вышел).

Наличие методов **pop** и **push** позволяет классам, реализующим этот элемент, действовать в качестве стека. Имеющийся функционал позволяет создавать двунаправленные очереди, что делает классы, применяющие данный интерфейс, довольно гибкими.

1.3.3. Класс ArrayDeque

Класс **ArrayDeque** представляет обобщенную двунаправленную очередь, наследуя функционал от класса **AbstractCollection** и применяя интерфейс **Deque**. Пример использования класса:

```
import java.util.ArrayDeque;
public class Program{
    public static void main(String[] args) {
        ArrayDeque<String> states = new ArrayDeque<String>();
        // стандартное добавление элементов
        states.add("Germany");
        states.addFirst("France"); // добавляем элемент в самое
начало
        states.push("Great Britain"); // добавляем элемент в
самое начало
        states.addLast("Spain"); // добавляем элемент в конец
коллекции
        states.add("Italy");
        // получаем первый элемент без удаления
        String sFirst = states.getFirst();
        System.out.println(sFirst);     // Great Britain
        // получаем последний элемент без удаления
        String sLast = states.getLast();
        System.out.println(sLast);      // Italy
        System.out.printf("Queue size: %d \n", states.size());
// 5
        // перебор коллекции
        while(states.peek()!=null){
            // извлечение с начала
            System.out.println(states.pop());
        }
         // очередь из объектов Person
        ArrayDeque<Person> people = new ArrayDeque<Person>();
        people.addFirst(new Person("Tom"));
        people.addLast(new Person("Nick"));
        // перебор без извлечения
        for(Person p : people){
            System.out.println(p.getName());
        }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

1.4. Класс LinkedList

Обобщенный класс **LinkedList** представляет структуру данных в виде связанного списка. Он наследуется от класса **AbstractSequentialList** и реализует интерфейсы **List**, **Dequeue** и **Queue**. То есть он соединяет функциональность работы со списком и функциональность очереди.

Класс **LinkedList** имеет следующие конструкторы:

- **LinkedList()**: создает пустой список
- **LinkedList(Collection<? extends E> col)**: создает список, в который добавляет все элементы коллекции col

LinkedList содержит все те методы, которые определены в интерфейсах **List**, **Queue**, **Deque**. Варианты использования:

- **addFirst() / offerLast()**: добавляет элемент в начало/конец списка
- **removeFirst() / pollLast():** удаляет первый элемент из начала/конца списка
- **getFirst() / peekLast()**: получает первый/ последний элемент

1.5. Интерфейс Set и класс HashSet

Интерфейс **Set** расширяет интерфейс **Collection** и представляет набор уникальных элементов. **Set** не добавляет новых методов, только вносит изменения в унаследованные. В частности, метод **add()** добавляет элемент в коллекцию и возвращает *true*, если в коллекции еще нет такого элемента.

Обобщенный класс **HashSet** представляет хеш-таблицу, наследуя свой функционал от класса **AbstractSet** и реализуя интерфейс **Set.**

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Класс **HashSet** не добавляет новых методов, реализуя лишь те, что объявлены в родительских классах и применяемых интерфейсах.

1.6. SortedSet, NavigableSet, TreeSet

Интерфейс **SortedSet** предназначен для создания коллекций, который хранят элементы в отсортированном виде (сортировка по возрастанию). **SortedSet** расширяет интерфейс **Set**, поэтому такая коллекция хранит только уникальные значения. Интерфейс **NavigableSet** расширяет интерфейс **SortedSet** и позволяет извлекать элементы на основании их значений.

Обобщенный класс **TreeSet<E>** представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. **TreeSet** является наследником класса **AbstractSet**, реализует интерфейсы **SortedSet** и **NavigableSet**. поддерживает все стандартные методы для вставки и удаления элементов.

1.7. Интерфейсы Comparable и Comparator. Сортировка

Допустим, мы хотим использовать свой класс **Person**. Чтобы объекты **Person** можно было сравнить и сортировать, они должны применять интерфейс **Comparable<E>**. При применении интерфейса он типизируется текущим классом. Применим его к классу **Person**:

```
class Person implements Comparable<Person>{
    private String name;
    Person(String name){
        this.name = name;
    }
    String getName(){return name;}
    public int compareTo(Person p){
        return name.compareTo(p.getName());
    }
}
```

Интерфейс **Comparable** содержит метод **int compareTo(E item)**, который сравнивает текущий объект с объектом, переданным в качестве параметра. Если метод возвращает отрицательное число, то первый объект будет располагаться перед вторым; если положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В данном случае мы не возвращаем явным образом никакое число, а полагаемся на встроенный механизм сравнения класса **String**. Можно определить и свою логику, например, сравнивать по длине имени:

```
public int compareTo(Person p){
    return name.length()-p.getName().length();
}
```

Теперь мы можем типизировать **TreeSet** типом **Person** и добавлять в дерево соответствующие объекты:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```

Интерфейс **Comparator** содержит ряд методов, ключевым из которых является метод **compare()**:

```
public interface Comparator<E> {
    int compare(T a, T b);
    // остальные методы
}
```

Метод **compare** возвращает числовое значение: отрицательное - объект **a** предшествует объекту **b**, иначе – наоборот, ноль - объекты равны. Для применения интерфейса нужно создать класс компаратора, который реализует этот интерфейс:

```
class PersonComparator implements Comparator<Person>{
    public int compare(Person a, Person b){
        return a.getName().compareTo(b.getName());
    }
}
```

Используем класс компаратора для создания объекта **TreeSet**:

```
PersonComparator pcomp = new PersonComparator();
TreeSet<Person> people = new TreeSet<Person>(pcomp);
people.add(new Person("Tom"));
people.add(new Person("Nick"));
people.add(new Person("Alice"));
people.add(new Person("Bill"));
for(Person  p : people){
    System.out.println(p.getName());
}
```

Версия конструктора, принимающая компаратор в качестве параметра. Теперь вне зависимости от того, реализован ли в классе **Person** интерфейс **Comparable**, будет использоваться та логика сравнения и сортировки, которая определена в классе компаратора.

1.8. Интерфейс Map и класс HashMap

1.8.1. Интерфейс Map

Интерфейс **Map<K, V>** представляет отображение (словарь), где каждый элемент представляет пару "ключ-значение". Все ключи уникальные в рамках объекта **Map**. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта. Интерфейс **Map** НЕ расширяет интерфейс **Collection**. Чтобы положить объект в коллекцию, используется метод **put**, чтобы получить его значение по ключу – метод **get**. Реализация интерфейса **Map** также позволяет получить наборы ключей и значений. Метод **entrySet()** возвращает набор всех элементов в виде объектов **Map.Entry<K, V>** с ключом типа K и значением типа V.

1.8.2. Классы отображений. HashMap

Наиболее распространенным классом отображений является **HashMap**, который реализует интерфейс **Map** и наследуется от класса **AbstractMap**. Чтобы добавить/заменить элемент, используются методы **put/replace**. С помощью других методов интерфейса **Map** производятся перебор элементов, получение ключей, значений, удаление.

Пример использования класса:

```
import java.util.*;
public class Program{
```

```
    public static void main(String[] args) {
        Map<Integer, String> states = new HashMap<Integer,
String>();
       states.put(1, "Germany");
       states.put(2, "Spain");
       states.put(4, "France");
       states.put(3, "Italy");
       // получим объект по ключу 2
       String first = states.get(2);
       System.out.println(first);
       // получим весь набор ключей
       Set<Integer> keys = states.keySet();
       // получить набор всех значений
       Collection<String> values = states.values();
       //заменить элемент
       states.replace(1, "Poland");
       // удаление элемента по ключу 2
       states.remove(2);
       // перебор элементов
       for(Map.Entry<Integer, String> item :
states.entrySet()){
           System.out.printf("Key: %d  Value: %s \n",
item.getKey(), item.getValue());
       }
       Map<String, Person> people = new HashMap<String,
Person>();
       people.put("1240i54", new Person("Tom"));
       people.put("1564i55", new Person("Bill"));
       people.put("4540i56", new Person("Nick"));
       for(Map.Entry<String, Person> item : people.entrySet()){
           System.out.printf("Key: %s  Value: %s \n",
item.getKey(), item.getValue().getName());
       }
    }
}
class Person{
    private String name;
    public Person(String value){
        name=value;
    }
    String getName(){return name;}
}
```

1.9. Интерфейсы SortedMap и NavigableMap. Класс TreeMap

Для создания отображений Java также предоставляет интерфейсы **SortedMap** и **NavigableMap**. Интерфейс **SortedMap** расширяет **Map** и создает отображение, в котором все элементы отсортированы в порядке возрастания их ключей. Интерфейс **NavigableMap** расширяет интерфейс **SortedMap** и обеспечивает возможность получения элементов отображения относительно других элементов.

Класс **TreeMap<K, V>** представляет отображение в виде дерева. Он наследуется от класса **AbstractMap** и реализует интерфейсы **NavigableMap** и **SortedMap**. В отличие от коллекции **HashMap**, в **TreeMap** все объекты автоматически сортируются по возрастанию их ключей.

1.10. Итераторы

Одним из ключевых методов интерфейса **Collection** является метод **Iterator<E> iterator()**, возвращающий итератор - объект, реализующий интерфейс **Iterator**:

```
public interface Iterator <E>{
    E next();
    boolean hasNext();
    void remove();
}
```

Реализация интерфейса предполагает использование методов:

– **next()**: возвращает следующий элемент, при достижении конца коллекции выбрасывает исключение **NoSuchElementException**;

– **hasNext():** показывает, не достигнут ли конец коллекции. Если следующий элемент имеется, то метод вернет значение *true*;

– **remove():** удаляет текущий элемент, который был получен последним вызовом **next()**.

Используем итератор для перебора коллекции **ArrayList**:

```
import java.util.*;
public class Program {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        states.add("Germany");
        states.add("France");
        states.add("Italy");
        states.add("Spain");
        Iterator<String> iter = states.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next());
        }
    }
}
```

Гораздо больший набор методов предоставляет интерфейс **ListIterator**, расширяющий интерфейс **Iterator**. Он используется классами, реализующими интерфейс **List** (**LinkedList**, **ArrayList** и др.)

```
import java.util.*;
public class Program {
    public static void main(String[] args) {
```

```
        ArrayList<String> states = new ArrayList<String>();
        states.add("Germany");
        states.add("France");
        states.add("Italy");
        states.add("Spain");
        ListIterator<String> listIter = states.listIterator();
        while(listIter.hasNext()){
            System.out.println(listIter.next());
        }
        // сейчас текущий элемент - Испания
        // изменим значение этого элемента
        listIter.set("Португалия");
        // пройдемся по элементам в обратном порядке
        while(listIter.hasPrevious()){
            System.out.println(listIter.previous());
        }
    }
}
```

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.

2. Выполните задания практической части лабораторной работы по варианту.

3. Практическая часть

3.1. Задание 1

1. Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.

2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.

3. Создать список из элементов каталога и его подкаталогов.

4. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.

5. Задать два стека, поменять информацию местами.

6. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.

7. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **HashMap**.

8. Умножить два многочлена заданной степени, коэффициенты которых хранятся в различных списках.

9. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные — в начало списка.

10. Ввести строки из файла, записать в список **ArrayList**. Выполнить сортировку строк, используя метод **sort()** из класса **Collections**.

11. Задана строка, состоящая из символов «(», «)», «[», «]», «{», «}». Проверить правильность расстановки скобок. Использовать стек.

12. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **HashSet**.

13. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс **HashMap**.

14. Заполнить **HashMap** 10 объектами. Найти строки, у которых *ключ* > 5. Если *ключ* = 0, вывести строки через запятую. Перемножить все ключи, где длина строки > 5.

15. Написать функцию, получающую итераторы на начало и конец отсортированного **List** и символ. Возвращать функция должна начало и конец диапазона, строки в котором начинаются с заданного символа.

3.2. Задание 2

1. В кругу стоят N человек, пронумерованных от 1 до N. При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из программ должна использовать класс **ArrayList**, а вторая — **LinkedList**. Какая из двух программ работает быстрее? Почему?

2. Задан список целых чисел и некоторое число X. Не используя вспомогательных объектов и методов сортировки и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие X, а затем числа, больше X.

3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **PriorityQueue**.

4. Реализовать класс **Graph**, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.

5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций: добавление/удаление числа; поиск числа, наиболее близкого к заданному (т.е. модуль разницы минимален).

6. Реализовать класс, моделирующий работу N-местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится свободное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.

7. Во входном файле хранятся две разреженные матрицы — A и B. Построить циклически связанные списки CA и CB, содержащие ненулевые элементы соответственно матриц A и B. Просматривая списки, вычислить: а) сумму S = A + B; б) произведение P = A × B.

8. Во входном файле хранятся наименования некоторых объектов. Построить список Z, элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возрастанию шифров. Затем «сжать» список Z, удаляя дублирующие наименования объектов.

9. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка C1 и C2, элементы которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возрастанию. Затем объединить списки C1 и C2 в один упорядоченный список, изменяя только значения полей ссылочного типа.

10. Один из способов шифрования данных, называемый «двойным шифрованием», заключается в том, что исходные данные при помощи некоторого преобразования последовательно шифруются на некоторые два ключа — K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному варианту. Проверить, оказался ли разработанный способ действительно эффективным, протестировав программу для случая, когда оба ключа являются 20-битными (время ее работы не должно превосходить одной минуты).

11. На плоскости задано N точек. Вывести в файл описания всех прямых, которые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **HashMap**.

12. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клетки до центра круга. Использовать класс **PriorityQueue.**

13. На плоскости задано N отрезков. Найти точку пересечения двух отрезков, имеющую минимальную абсциссу. Использовать класс **TreeMap**.

14. На клетчатом листе бумаги закрашена часть клеток. Выделить все различные фигуры, которые образовались при этом. Фигурой считается набор закрашенных клеток, достижимых друг из друга при движении в четырех направлениях. Две фигуры являются различными, если их нельзя совместить поворотом на угол, кратный 90 градусам, и параллельным переносом. Используйте класс **HashSet.**

15. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Использовать класс **ArrayDeque.**

4. Содержание отчета

1. Краткие теоретические сведения о коллекциях в Java.
2. Код программ.

3. Результаты выполнения программ.

4. Выводы по работе.

5. Контрольные вопросы

1. Назвать основные интерфейсы коллекций. Какие бывают коллекции?
2. В чем особенности разных видов коллекций? Когда и какие коллекции следует применять?
3. Сравнить **ArrayList** и **LinkedList**.
4. Сравнить **HashMap** и **Hashtable**.
5. Как устроены **HashSet**, **TreeMap**, **TreeSet**.
6. Принцип работы и реализации **HashMap**. Изменения в java 8.
7. Чем отличается **ArrayList** от **Vector**?
8. Особенности интерфейса **Set**.
9. Как добавляются объекты в **HashSet**?
10. Какими способами можно отсортировать коллекцию (3 способа)?
11. Как удалить элемент из коллекции при итерации в цикле?
12. Как удалить элемент из **ArrayList** (или другой коллекции) при поиске этого элемента в цикле?
13. Коллекции из **пакета java.util.concurrent**. Их особенности.
14. Как реализовано добавление в **ArrayList** нового элемента?
15. Метод для преобразования потоконебезопасной коллекции в потокобезопасную.
16. Какие коллекции более «быстрые» — *legacy* (**Vector**, **Hashtable**) или из пакета **java.util.concurrent**?
17. Если в коллекцию часто добавлять элементы и удалять, какую лучше использовать? Почему? Как они устроены?
18. Как получить копию коллекции? Записать код преобразования.
19. Чем **Stream** отличается от коллекции?
20. Промежуточные и терминальные операции в *stream*.
21. Методы: **map()** *vs* **flatMap()** в *stream*.
22. Что такое потоковая обработка данных?

Занятие № 10
Java DataBase Connectivity

Цель работы: изучение взаимодействия языка программирования Java с базами данных, приобретение навыков работы с БД в Java-программах.

1. Теоретическая часть

1.1. Java и базы данных. Строение и элементы JDBC

Java Database Connectivity – это стандартный API для независимого соединения языка программирования Java с различными базами данных (далее – БД).

JDBC решает следующие задачи: создание соединения с БД, создание SQL выражений, выполнение SQL – запросов, просмотр и модификация полученных записей.

В целом, JDBC – это библиотека, которая обеспечивает целый набор интерфейсов для доступа к различным БД. Для доступа к каждой конкретной БД необходим специальный JDBC – драйвер, который является адаптером Java – приложения к БД.

JDBC поддерживает как 2-звенную, так и 3-звенную модель работы с БД, но в общем виде, JDBC состоит из двух слоев:

- JDBC API: обеспечивает соединение “приложение – JDBC Manager”.
- JDBC Driver API: обеспечивает соединение “JDBC Manager – драйвер”.

JDBC API использует менеджер драйверов и специальные драйверы БД для обеспечения подключения к различным БД.

JBDC Manager проверяет соответствие драйвера и конкретной БД, поддерживает возможность одновременного использования нескольких драйверов для одновременной работы с несколькими видами БД.

Ключевыми пакетами JDBC являются *java.sql* и *javax.sql*.

В табл. 4 представлен список элементов JDBC API.

Таблица 4 – Элементы JDBC API

№	Элемент и описание
1	**Менеджер драйверов (Driver Manager)** Управляет списком драйверов БД. Каждой запрос на соединение требует соответствующего драйвера. Первое совпадение даёт нам соединение.
2	**Драйвер (Driver)** Отвечает за связь с БД. Работать с ним нам приходится крайне редко. Вместо этого мы чаще используем объекты DriverManager, которые управляют объектами этого типа.
3	**Соединение (Connection)** Обеспечивает нас методами для работы с БД. Все взаимодействия с БД происходят исключительно через Connection.
4	**Выражение (Statement)** Для подтверждения SQL-запросов мы используем объекты, созданные с использованием этого интерфейса.
5	**Результат (ResultSet)** Экземпляры этого элемента содержат данные, полученные в результате выполнения SQL – запроса. Он работает как итератор и “пробегает” по полученным данным.
6	**Исключения (SQL Exception)** Обрабатывает все ошибки, которые могут возникнуть при работе с БД.

1.2. JDBC. Базовый синтаксис SQL

Язык структурированных запросов (**S**tructured **Q**uery **L**anguage) **SQL –** это стандартизированный язык, который позволяет выполнять операции с базами данных: создание и удаление БД, таблиц; создание, редактирование, удаление и чтение записей из таблиц и т.д.

SQL поддерживается большинством используемых БД. Для выполнения операций с БД применяется следующий синтаксис:

- Создание БД:

```
CREATE DATABASE PROSELYTE_DATABASE;
```

- Удаление БД:

```
DROP DATABASE PROSELYTE_DATABASE;
```

После создания БД нам необходимо создать таблицу, в которой будут храниться записи.

- Создание таблицы:

```
CREATE TABLE developers{
   id INT,
   name VARCHAR(50),
   specialty VARCHAR(50),
   salary INT
}
```

- Удаление таблицы:

```
DROP TABLE developer;
```

- Добавление записи в таблицу:

```
INSERT INTO developers VALUES (1, 'Proselyte', 'Java', 2000);
```

- Получение записи из таблицы:

```
SELECT * FROM developers;
```

Так мы получим все записи из таблицы. Если же нам нужно вывести только определенные столбцы (напр., id и имя) с условием (напр., со специальностью Java), то запрос будет выглядеть так:

```
SELECT id, name FROM developers WHERE specialty LIKE '%java%';
```

- Редактирование записи в таблице:

```
UPDATE developers SET salary = 3000 WHERE specialty LIKE
'%java%';
```

Эта запись установит зарплату 3000 для всех записей, в которых специальность содержит слово JAVA.

- Удаление записи из таблицы:

```
DELETE FROM developers WHERE name = 'PETER';
```

Этот код удалит из таблицы запись с именем PETER.

1.3. Пример простого приложения

Создадим Java-приложение с использованием JDBC, в котором мы создадим соединение с базой данных (далее – БД), выполним

несколько SQL–запросов и отобразим результат в консоли. Для создания данного приложения нам нужно выполнить следующие шаги: 1) создать простое Java приложение → 2) создать базу данных и таблицу в ней → 3) импортировать пакет **java.sql.*** → 4) использовать JDBC драйвер → 5) создать соединение → 6) выполнить запрос → 7) получить данные из БД → 8) закрыть соединения.

Рассмотрим пример создания простого приложения:

1. Добавление зависимости MySQL Connector в POM.xml

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.38</version>
</dependency>
```

2. Создание таблицы developers в БД

```
CREATE TABLE PROSELYTE_TUTORIALS.developers (
  id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(50) NOT NULL,
  specialty VARCHAR(50) NOT NULL,
  salary INT NOT NULL,
  PRIMARY KEY (id));
```

3. Добавление записей в таблицу developers

```
INSERT INTO PROSELYTE_TUTORIALS.developers (name, specialty,
salary) VALUES ('Proselyte', 'Java', '2000');
INSERT INTO PROSELYTE_TUTORIALS.developers (name, specialty,
salary) VALUES ('Peter','C++', '3000');
INSERT INTO PROSELYTE_TUTORIALS.developers (name, specialty,
salary) VALUES ('AsyaSmile', 'UI/UX', '2000');
```

4. Класс DevelopersJdbcDemo

```
import java.sql.*;
public class DevelopersJdbcDemo {
    /**
     * JDBC Driver and database url
     */
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static       final       String       DATABASE_URL       =
"jdbc:mysql://localhost/PROSELYTE_TUTORIALS";
    /**
     * User and Password
     */
    static final String USER = "ВВЕДИТЕ ВАШЕ ИМЯ ПОЛЬЗОВАТЕЛЯ";
    static final String PASSWORD = "ВВЕДИТЕ ВАШ ПАРОЛЬ";
    public    static    void    main(String[]    args)    throws
ClassNotFoundException, SQLException {
        Connection connection = null;
        Statement statement = null;
```

```
        System.out.println("Registering JDBC driver...");
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Creating database connection...");
        connection  =  DriverManager.getConnection(DATABASE_URL,
USER, PASSWORD);
        System.out.println("Executing statement...");
        statement = connection.createStatement();
        String sql;
        sql = "SELECT * FROM developers";
        ResultSet resultSet = statement.executeQuery(sql);
        System.out.println("Retrieving data from database...");
        System.out.println("\nDevelopers:");
        while (resultSet.next()) {
            int id = resultSet.getInt("id");
            String name = resultSet.getString("name");
            String                  specialty                  =
resultSet.getString("specialty");
            int salary = resultSet.getInt("salary");
            System.out.println("\n================\n");
            System.out.println("id: " + id);
            System.out.println("Name: " + name);
            System.out.println("Specialty: " + specialty);
            System.out.println("Salary: $" + salary);
        }
        System.out.println("Closing  connection  and  releasing
resources...");
        resultSet.close();
        statement.close();
        connection.close();
    }
}
```

5. В результате работы программы мы получим следующий результат:

```
/*Some System Messages*/

Registering JDBC driver...
Creating database connection...
Executing statement...
Retrieving data from database...

Developers:

================

id: 1
Name: Proselyte
Specialty: Java
Salary: $2000

================

id: 2
```

```
Name: Peter
Specialty: C++
Salary: $3000

================

id: 3
Name: AsyaSmile
Specialty: UI/UX
Salary: $2000
Closing connection and releasing resources...
```

Так будет выглядеть наша таблица в БД:

```
mysql> SELECT * FROM developers;
+----+-----------+-----------+--------+
| id | name      | specialty | salary |
+----+-----------+-----------+--------+
|  1 | Proselyte | Java      |   2000 |
|  2 | Peter     | C++       |   3000 |
|  3 | AsyaSmile | UI/UX     |   2000 |
+----+-----------+-----------+--------+
```

2. Порядок выполнения работы

1. Изучите теоретическую часть лабораторной работы.

2. Выполните задания практической части лабораторной работы по варианту.

3. Практическая часть

3.1. Задание 1

В каждом из заданий необходимо выполнить следующие действия:

• организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение;

• создать БД. Привести таблицы к одной из нормальных форм;

• создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов;

• создать класс на модификацию информации.

1. **Файловая система**. В БД хранится информация о дереве каталогов файловой системы — каталоги, подкаталоги, файлы.

Для каталогов необходимо хранить: родительский каталог, название.

Для файлов необходимо хранить: родительский каталог, название, место, занимаемое на диске.

• Определить полный путь заданного файла (каталога).

• Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и каталоги.

• Подсчитать место, занимаемое на диске содержимым заданного каталога.

• Найти в базе файлы по заданной маске с выдачей полного пути.

• Переместить файлы и подкаталоги из одного каталога в другой.

• Удалить файлы и каталоги заданного каталога.

2. **Видеотека**. В БД хранится информация о домашней видеотеке: фильмы, актеры, режиссеры.

Для фильмов необходимо хранить: название, имена актеров, дату выхода, страну, в которой выпущен фильм.

Для актеров и режиссеров необходимо хранить: ФИО, дату рождения.

• Найти все фильмы, вышедшие на экран в текущем и прошлом году.

• Вывести информацию об актерах, снимавшихся в заданном фильме.

• Вывести информацию об актерах, снимавшихся как минимум в N фильмах.

• Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.

• Удалить все фильмы, дата выхода которых была более заданного числа лет назад.

3. **Расписание занятий**. В БД хранится информация о преподавателях и проводимых ими занятиях.

Для предметов необходимо хранить: название, время проведения (день недели), аудитории, в которых проводятся занятия.

Для преподавателей необходимо хранить: ФИО; предметы, которые он ведет; количество пар в неделю по каждому предмету; количество студентов, занимающихся на каждой паре.

• Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.

• Вывести информацию о преподавателях, которые не ведут занятия в заданный день недели.

• Вывести дни недели, в которых проводится заданное количество занятий.

• Вывести дни недели, в которых занято заданное количество аудиторий.

• Перенести первые занятия заданных дней недели на последнее место.

4. **Письма**. В БД хранится информация о письмах и отправителях.

Для людей необходимо хранить: ФИО, дату рождения.

Для писем необходимо хранить: отправителя, получателя, тему письма, текст письма, дату отправки.

• Найти пользователя, длина писем которого наименьшая.

• Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.

• Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.

• Вывести информацию о пользователях, которые не получали сообщения с заданной темой.

• Направить письмо заданного человека с заданной темой всем адресатам.

5. **Сувениры**. В БД хранится информация о сувенирах и производителях.

Для сувениров необходимо хранить: название, реквизиты производителя, дату выпуска, цену.

Для производителей необходимо хранить: название, страну.

• Вывести информацию о сувенирах заданного производителя.

• Вывести информацию о сувенирах, произведенных в заданной стране.

• Вывести информацию о производителях, чьи цены на сувениры меньше заданной.

• Вывести информацию о производителях заданного сувенира, произведенного в заданном году.

• Удалить заданного производителя и его сувениры.

6. **Заказ**. В БД хранится информация о заказах магазина и товарах в них.

Для заказа необходимо хранить: номер заказа, товары в заказе, дату поступления.

Для товаров в заказе необходимо хранить: товар, количество.

Для товара необходимо хранить: название, описание, цену.

• Вывести полную информацию о заданном заказе.

• Вывести номера заказов, сумма которых не превосходит заданную, и количество различных товаров равно заданному

• Вывести номера заказов, содержащих заданный товар.

• Вывести номера заказов, не содержащих заданный товар и поступивших в течение текущего дня.

• Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.

• Удалить все заказы, в которых присутствует заданное количество заданного товара.

7. **Продукция**. В БД хранится информация о продукции компании.

Для продукции необходимо хранить: название, группу продукции (телефоны, телевизоры и др.), описание, дату выпуска, значения параметров.

Для групп продукции необходимо хранить: название, перечень групп параметров (размеры и др.).

Для групп параметров необходимо хранить: название, перечень параметров.

Для параметров необходимо хранить: название, единицу измерения.

• Вывести перечень параметров для заданной группы продукции.

• Вывести перечень продукции, не содержащий заданного параметра.

• Вывести информацию о продукции для заданной группы.

• Вывести информацию о продукции и всех ее параметрах со значениями.

• Удалить из базы продукцию, содержащую заданные параметры.

• Переместить группу параметров из одной группы товаров в другую.

8. **Погода**. В БД хранится информация о погоде в различных регионах.

Для погоды необходимо хранить: регион, дату, температуру, осадки.

Для регионов необходимо хранить: название, площадь, тип жителей.

Для типов жителей необходимо хранить: название, язык общения.

• Вывести сведения о погоде в заданном регионе.

• Вывести даты, когда в заданном регионе шел снег и температура была ниже заданной отрицательной.

• Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.

• Вывести среднюю температуру за прошедшую неделю в регионах с площадью больше заданной.

9. **Магазин часов**. В БД хранится информация о часах.

Для часов необходимо хранить: марку, тип (кварцевые или механические), стоимость, количество, реквизиты производителя.

Для производителей необходимо хранить: название; страна.

• Вывести марки заданного типа часов.

• Вывести информацию о механических часах, стоимость которых не превышает заданную.

• Вывести марки часов, изготовленных в заданной стране.

• Вывести производителей, общая сумма часов которых в магазине не превышает заданную.

10. **Города**. В БД хранится информация о городах и их жителях.

Для городов необходимо хранить: название, год основания, площадь, количество населения для каждого типа жителей.

Для типов жителей необходимо хранить: город проживания, название; язык общения.

• Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.

• Вывести информацию обо всех городах, в которых проживают жители выбранного типа.

• Вывести информацию о городе с заданным количеством населения и всех типах жителей, в нем проживающих.

• Вывести информацию о самом древнем типе жителей.

11. **Планеты**. В БД хранится информация о планетах, их спутниках и галактиках.

Для планет необходимо хранить: название, радиус, температуру ядра, наличие атмосферы, наличие жизни, спутники.

Для спутников необходимо хранить: название, радиус, расстояние до планеты.

Для галактик необходимо хранить: название, планеты.

• Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.

• Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.

• Вывести информацию о планете, галактике, в которой она находится, и ее спутниках, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников.

• Найти галактику, сумма ядерных температур планет которой наибольшая.

12. **Точки**. В БД хранится некоторое конечное множество точек с их координатами.

• Вывести точку из множества, наиболее приближенную к заданной.

• Вывести точку из множества, наиболее удаленную от заданной.

• Вывести точки из множества, лежащие на одной прямой с заданной прямой.

13. **Треугольники**. В БД хранятся треугольники и координаты их точек на плоскости.

• Вывести треугольник, площадь которого наиболее приближена к заданной.

• Вывести треугольники, сумма площадей которых наиболее приближена к заданной.

• Вывести треугольники, которые помещаются в окружность заданного радиуса.

• Вывести все равнобедренные треугольники.

• Вывести все равносторонние треугольники.

• Вывести все прямоугольные треугольники.

• Вывести все тупоугольные треугольники с площадью больше заданной.

14. **Стихотворения**. В БД хранятся несколько стихотворений с указанием автора и года создания. Для хранения использовать объекты типа Clob. Клиент выбирает автора и критерий поиска.

• Вывести стихотворение, в котором больше всего восклицательных предложений?

• В каком из стихотворений меньше всего повествовательных предложений?

• Есть ли среди стихотворений сонеты и сколько их?

15. **Четырехугольники**. В БД хранятся координаты вершин выпуклых четырехугольников на плоскости.

• Вывести координаты вершин параллелограммов.

• Вывести координаты вершин трапеций.

4. Содержание отчета

1. Краткие теоретические сведения о Java DataBase Connectivity.
2. Код программ.
3. Результаты выполнения программ.
4. Выводы по работе.

5. Контрольные вопросы

1. Что такое JDBC? Перечислить основные классы и интерфейсы, входящие в состав JDBC, указать их назначение. Какие еще существуют технологии Java, работающие с БД?
2. Привести алгоритм получения соединения с БД, выполнения запроса и обработки результатов. Как загрузить драйвер БД и что он собой представляет. Какие существуют типы драйверов БД в JDBC? Нужно ли регистрировать драйвер БД? Если да, то как это сделать?
3. Как правильно закрыть **Connection**?
4. Какие есть типы драйверов для соединения с СУБД?
5. Чем отличается **Statement** от **PreparedStatement**? Где сохраняется запрос после первого вызова **PreparedStatement**? Будет ли результат идентичным **PreparedStatement**, если формировать запрос просто в строке и отправлять его в **Statement**?
6. Защищены ли **Statement** и **PreparedStatement** от *sql-injection*? Можно ли работать с несколькими объектами *statement* или *prepared statement*, полученными от одного объекта *connection* одновременно и может ли такое использование быть небезопасным?
7. Зачем нужен **CallableStatement**? Как выполняется вызов хранимых процедур из Java-программы? Что называется *batch*-командой, как выполнить *batch*-команду?
8. Чем отличаются метод **executeUpdate()** от **executeQuery()**?
9. Для чего JDBC использует объекты типа **ResultSet**?
10. Что означает прокручиваемый/непрокручиваемый **ResultSet**, обновляемый/необновляемый **ResultSet**?
11. Как можно получить такие различные типы объектов **ResultSet**? Можно ли через объект **ResultSet** изменить значения в БД и, если да, то каким образом?
12. Как в объекте **ResultSet** вернуться в предыдущую строку? Всегда ли можно вернуться в предыдущую строку?

13. Как получить сгенерированный СУБД первичный ключ без выполнения дополнительного запроса к БД?

14. Как узнать, в какие типы Java будут конвертированы при выборке *sql*-типы данных?

15. Привести определение транзакции, *commit* и *rollback*. Как JDBC работает с транзакциями по умолчанию? Как отменить *autocommit*, как в этом случае следует работать с БД?

16. Что такое точка сохранения и как ее создать? Как откатить транзакцию до точки сохранения или до предыдущего *commit*?

17. Что такое пул соединений с БД, для чего он необходим? Каковы основные принципы создания пула соединений к БД?

18. Что означает термин «метаданные»? Какую информацию предоставляют объекты классов **DatabaseMetaData**, **ResultsSetMetaData** и для чего она может быть использована?

19. Что означает термин *уровень изоляции транзакции*? Какие уровни изоляции транзакций поддерживает JDBC? Как задать уровень изоляции транзакций?

Библиографический список

1. Java. Промышленное программирование : практ. пособие / И.Н. Блинов, В.С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.

2. Java. Методы программирования : уч.-мет. пособие / И.Н. Блинов, В.С. Романчик. — Минск : издательство «Четыре четверти», 2013. — 896 с.

3. Java from EPAM : учеб.-метод. пособие / И.Н. Блинов, В.С. Романчик. — Минск : Четыре четверти, 2020. — 560 с.