

Лабораторная работа № 2

Создание модульного JAVA - приложения

Оглавление

Цель лабораторной работы	2
1 Исключительные ситуации и их обработка	2
1.1 Обработка исключений в программах на Java	2
1.2 Классы исключений и их использование	5
2 Потоки ввода-вывода данных	7
3 Коллекции	12
3.1 Иерархия базовых абстрактных классов и интерфейсов коллекций в Java	12
3.2 Коллекции, реализующие интерфейс Collection	13
3.2.1 Списки	14
3.2.2 Очереди	17
3.2.3 Множества	20
3.2.4 Организация перебора элементов коллекций, реализующих интерфейс Collection	23
3.3 Коллекции, реализующие интерфейс Map	24
3.3.1 Разновидности коллекций на основе интерфейса Map	26
3.3.2 Организация перебора элементов коллекций, реализующих интерфейс Map	28
4 Создание модульных приложений в Java	29
5 Базовое приложение для лабораторной работы	34
5.1 Структура приложения	34
5.2 Создание модуля <i>arist.lab2.datastorage</i> в Eclipse	34
5.2.1 Создание проекта DataStorage	35
5.2.2 Создание класса TextFileDataSource для загрузки данных	36
5.2.3 Экспорт содержимого модуля <i>arist.lab2.datastorage</i> для использования в других модулях	39
5.3 Создание модуля <i>arist.lab2.main</i> в Eclipse	39
5.3.1 Создание проекта MainApp	39
5.3.2 Создание класса Main для запуска и тестирования многомодульного приложения	40
5.3.3 Подключение модуля <i>arist.lab2.datastorage</i> к модулю <i>arist.lab2.main</i>	41
5.4 Запуск приложения	43
6 Задания	46
6.1 Вариант сложности А	46
6.2 Вариант сложности В	48
6.3 Вариант сложности С	53
Приложение 1. Исходный код базового приложения	54

Цель лабораторной работы

Получить практические навыки создания модульных приложений на языке Java, ознакомиться с использованием коллекций, потоков ввода/вывода, а также получить навыки обработки исключений.

1 Исключительные ситуации и их обработка

1.1 Обработка исключений в программах на Java

При выполнении кода Java-программы могут возникать исключительные ситуации, причинами которых являются особенности обрабатываемой информации (деление на 0, некорректно введенные данные), особенности поведения системы (отсутствие файла или ограничение прав на доступ к нему) и т.д. Появление исключительных ситуаций приводит к нарушению работы алгоритма программы, что по возможности должно учитываться при ее разработке, для предотвращения ее аварийного завершения.

В Java для отслеживания и обработки исключительных ситуаций используется конструкция *try-catch-finally*, шаблон для классического варианта которой выглядит следующим образом:

```
try {
    //Код, потенциально способный сгенерировать исключение
}
catch (<ТипИсключения1> <переменная>) {
    //Код для обработки исключения типа 'ТипИсключения1'
}
...
catch (<ТипИсключенияN> <переменная>) {
    //Код для обработки исключения типа 'ТипИсключенияN'
}
finally{
    //Код, выполняемый если исключение не произошло,
    //либо после его обработки
}
```

Код, потенциально способный вызвать исключение, помещается в блок *try*, после которого следует набор блоков *catch*, содержащих код, выполняемый при возникновении исключения определенного типа, и при необходимости блок *finally*, в котором размещается код, предназначенный для выполнения независимо от того, имели место исключения или нет.

Содержимое блока *finally* выполняется даже если в блоках *catch* присутствуют операторы, способные прервать выполнение текущего блока кода (*break*, *continue*, *return*), однако при вызове функции *System.exit()*, код блока *finally* не выполняется. При необходимости могут использоваться вложенные блоки *try-catch-finally*. Если возникшее исключение не

обрабатывается в коде текущего метода (нет блока *try-catch-finally*, или нет соответствующей возникшему исключению ветки *catch*), то выполнение метода прекращается и исключение будет передано вызывающему методу, который должен его обработать. При отсутствии обработки возникшего исключения во всех методах вплоть до *main()* управление будет передано исполняющей системе Java и работа программы будет прекращена.

Рассмотрим пример, обработки исключений при чтении информации из файла:

```
DataInputStream in = null;
try {
    in = new DataInputStream(new FileInputStream("D:\\data.txt"));
    double[] data = new double[in.available()/(Double.SIZE/8)];
    int i = 0;
    while (in.available()>0)
        data[i++] = in.readDouble();
}
catch (FileNotFoundException ex) {
    System.out.println("Файл не найден");
    return;
}
catch (IOException ex) {
    System.out.println("Ошибка чтения данных из файла");
    return;
}
finally {
    try {
        if(in!=null)
            in.close();
    }
    catch (IOException e) {
        System.out.println("Ошибка при закрытии файла");
    }
}
```

В данном примере в первых двух ветках *catch* обрабатываются возможные исключения, связанные с отсутствием файла с данными и ошибками чтения данных из файла. Секция *finally* используется для гарантированного освобождения ресурсов, связанных с открытием файла данных независимо от того, были сгенерированы исключения или нет. Поскольку метод *close* способен сам генерировать исключение, которое может произойти при закрытии файла, то для его обработки используется вложенный блок *try-catch*.

Для упрощения написания обработки исключений в ситуации, когда необходимо освобождать ресурсы (например, закрыть файл), используемые в коде блока *try*, начиная с JDK7 был введен оператор *try с ресурсами*. Шаблон для такого оператора выглядит следующим образом:

```

try(<ресурс1>; [<ресурс2>; ...]){
    //использование <ресурс1>, <ресурс2>, ...
}
catch(<ТипИсключения1> <переменная>) {
    //...
}
...
catch(<ТипИсключенияN> <переменная>) {
    //...
}
finally{
    //...
}

```

Ресурсы в приведенном выше шаблоне представляют собой объекты класса, реализующего интерфейс *java.lang.AutoCloseable* и реализуют его единственный метод *close*.

Порядок выполнения операций при использовании *try* с ресурсами:

1. создаются объекты ресурсов в порядке их следования, если есть соответствующий код в круглых скобках после ключевого слова *try*;
2. выполняется код блока *try*, либо до его окончания, либо до генерации исключения;
3. выполняется закрытие (вызов *close*) для всех ресурсов, перечисленных в круглых скобках после *try*, в обратном порядке. Если на шаге 1 произошло исключение, то метод *close* вызывается для всех ресурсов, предшествующих ресурсу, вызвавшему исключение;
4. порядок учета исключений:
 - **первое исключение**, появившееся при выполнении шагов 1-3 становится **основным** и его тип определяет ветку *catch*, которая будет выполнена;
 - **остальные исключения**, сгенерированные на шаге 3, **попадают в список подавленных** исключений, доступных их основного при вызове вида:

```
ex.getSuppressed(); //ex - объект основного исключения
```

5. выполняется код блока **catch**, соответствующего основному исключению;
6. выполняется код блока **finally**.

Рассмотрим пример, обработки исключений при чтении информации из файла, который рассматривался ранее и был переписан с использованием *try* с ресурсами:

```

try(var in = new DataInputStream(
    new FileInputStream("D:\\data.txt"))) {
    double[] data =
        new double[in.available()/(Double.SIZE/8)];
    int i = 0;
    while (in.available()>0)
        data[i++] = in.readDouble();
}
catch (FileNotFoundException ex) {
    System.out.println("файл не найден");
    return;
}
catch (IOException ex) {
    System.out.println("Ошибка чтения данных из файла");
    return;
}

```

Как видно из приведенного выше кода, объект потока чтения данных *in* используется в качестве ресурса в блоке *try*. По этой причине данный поток будет закрыт автоматически независимо от появления исключений в блоке *try*, что позволило избавиться по сравнению с классическим вариантом кода от секции *finally*.

Если при работе некоторого метода могут быть сгенерированы исключения, которые должны быть обработаны вызывающим кодом, то это указывается при определении метода при помощи ключевого слова *throws*, после которого через запятую перечисляются типы генерируемых методом исключений. Пример определения такого метода приведен ниже.

```

void SampleMethod() throws IOException, ArithmeticException {
    ...
}

```

1.2 Классы исключений и их использование

Различные типы исключений в Java представлены классами – потомками класса *Throwable* (рисунок 1).

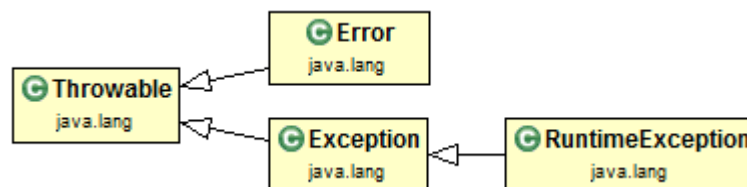


Рис 1. Иерархия базовых классов исключений а Java

Класс *Error* является базовым для определения катастрофических ошибок, после которых дальнейшая работа программы невозможна. Ошибки такого рода нельзя перехватить, используя *catch*.

Класс *Exception* и его подклассы описывают обрабатываемые исключения, которые можно разделить на две группы: непроверяемые и проверяемые. Непроверяемые исключения порождаются от класса *RuntimeException*, который является наследником *Exception*. Наличие кода, обрабатывающего такие исключения, не проверяется компилятором и решение о его необходимости принимается программистом. Остальные классы, порожденные от *Exception*, а также сам *Exception*, описывают проверяемые исключения и наличие кода для их обработки обязательно и контролируется компилятором.

Существует ряд исключений, описанных в различных пакетах Java, например, *ArithmeticException* – в пакете *java.lang*, *IOException* – в пакете *java.io* и т.д. Кроме того, можно создавать собственные классы исключений, наследуя их от *Exception*, или его подклассов. При этом имена для таких классов собственных исключений принято заканчивать на *Exception*. Ниже приведен пример класса *RangeException* для собственного исключения, которое может, например генерироваться, если некоторая величина не попадает в заданный диапазон:

```
class RangeException extends Exception {
    double value;

    public RangeException(double v) {
        this.value = v;
    }
    public String toString() {
        return "Значение " + this.value + " вне диапазона.";
    }
}
```

При написании программного кода на Java программист может самостоятельно вызвать генерацию исключения используя оператор *throw*.

Следующий пример демонстрирует применение оператора *throw* для генерации исключений, объекты которых созданы с использованием библиотечного класса *Exception* (если проверяемое значение меньше нуля) и собственного класса *RangeException* (если проверяемое значение больше десяти):

```

try {
    if(value<0)
        throw new Exception("Обнаружено отрицательное значение.");
    else if(value>10)
        throw new RangeException(value);

    System.out.println("Значение в диапазоне");
}
catch (RangeException ex) {
    System.out.println(ex);
    return;
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
    return;
}

```

Стоит отметить, что в данном примере важен порядок следования блоков *catch*. В случае если первым блоком будет блок, обрабатывающий исключения типа *Exception*, то обработка обоих сгенерированных исключений будет происходить в этом блоке, т.к. *RangeException* является подклассом *Exception*. Более того, если за блоком *catch*, обрабатывающим исключения типа *Exception*, будет следовать блок *catch*, обрабатывающий *RangeException*, то он не будет никогда срабатывать. Таким образом при наличии нескольких блоков *catch*, обрабатывающих исключения, классы которых унаследованы друг от друга, первыми должны следовать блоки, соответствующие подклассам.

2 Потоки ввода-вывода данных

Для обмена информацией с различными источниками (файл, клавиатура, сетевые соединения и т.д.), Java-программы используют *потоки данных*.

Поток данных (data stream) это некоторая последовательность блоков данных (байт, символов, и т.д.) которая передается между программой и некоторым источником или приемником, способным предоставлять или получать информацию (рисунок 2).

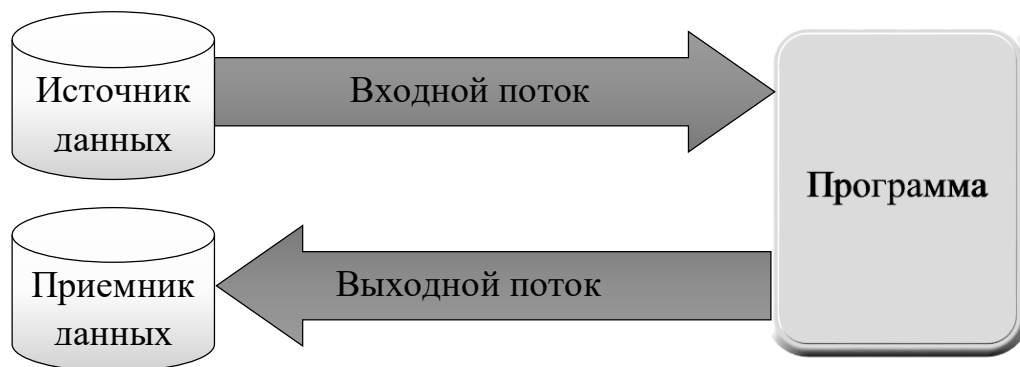


Рис 2. Потоки данных

Поток называется *входным* если он предназначен для чтения информации из некоторого источника. Если поток используется для передачи информации из программы некоторому приемнику, то он называется *выходным*. Использование концепции потоков позволяет стандартизировать обмен информацией с различными устройствами и таким образом отделить основную логику работы программы от низкоуровневых операций, различных для разных устройств.

В Java потоки ввода/вывода информации представлены объектами, классы которых содержатся в пакете *java.io* (java input/output). В зависимости от типа данных, с которыми работает поток, все потоки могут быть разделены на две большие группы:

1. *потоки байт*, представленные объектами классов *InputStream* (входные потоки) и *OutputStream* (выходные потоки) и их подклассов;
2. *потоки символов*, представленные объектами классов *Reader* (входные потоки) и *Writer* (выходные потоки) и их подклассов.

Иерархия классов для потоков байт, которые в готовом виде присутствуют в библиотеке Java, приведена на рисунке 3. На данном рисунке все классы разделены на две группы в зависимости от направления передачи информации между некоторым контейнером данных и программой.

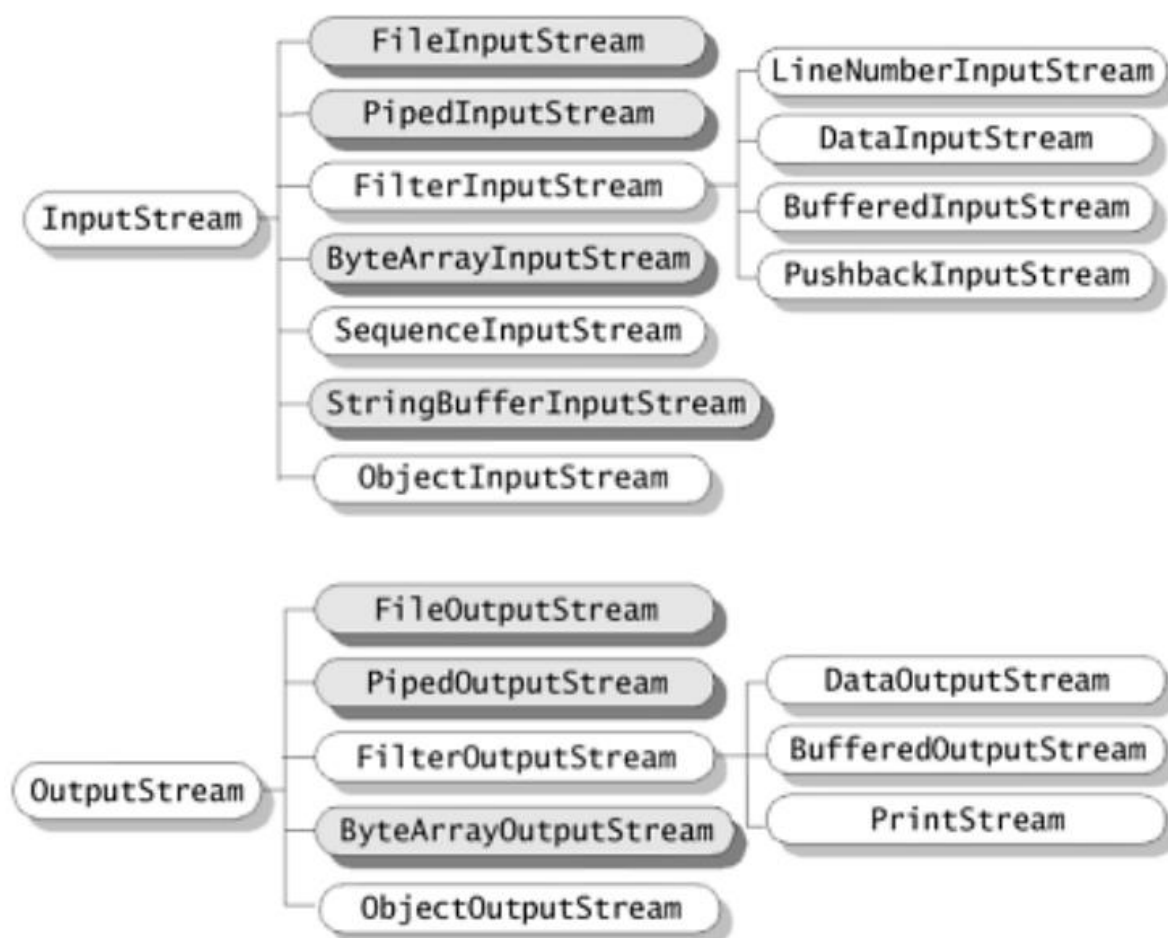


Рис 3. Иерархия классов потоков байт

Иерархия классов библиотеки Java, предназначенных для организации потоков ввода/вывода символов, приведена на рисунке 4. На данном рисунке также выделены две группы классов в зависимости от направления передачи символьной информации.

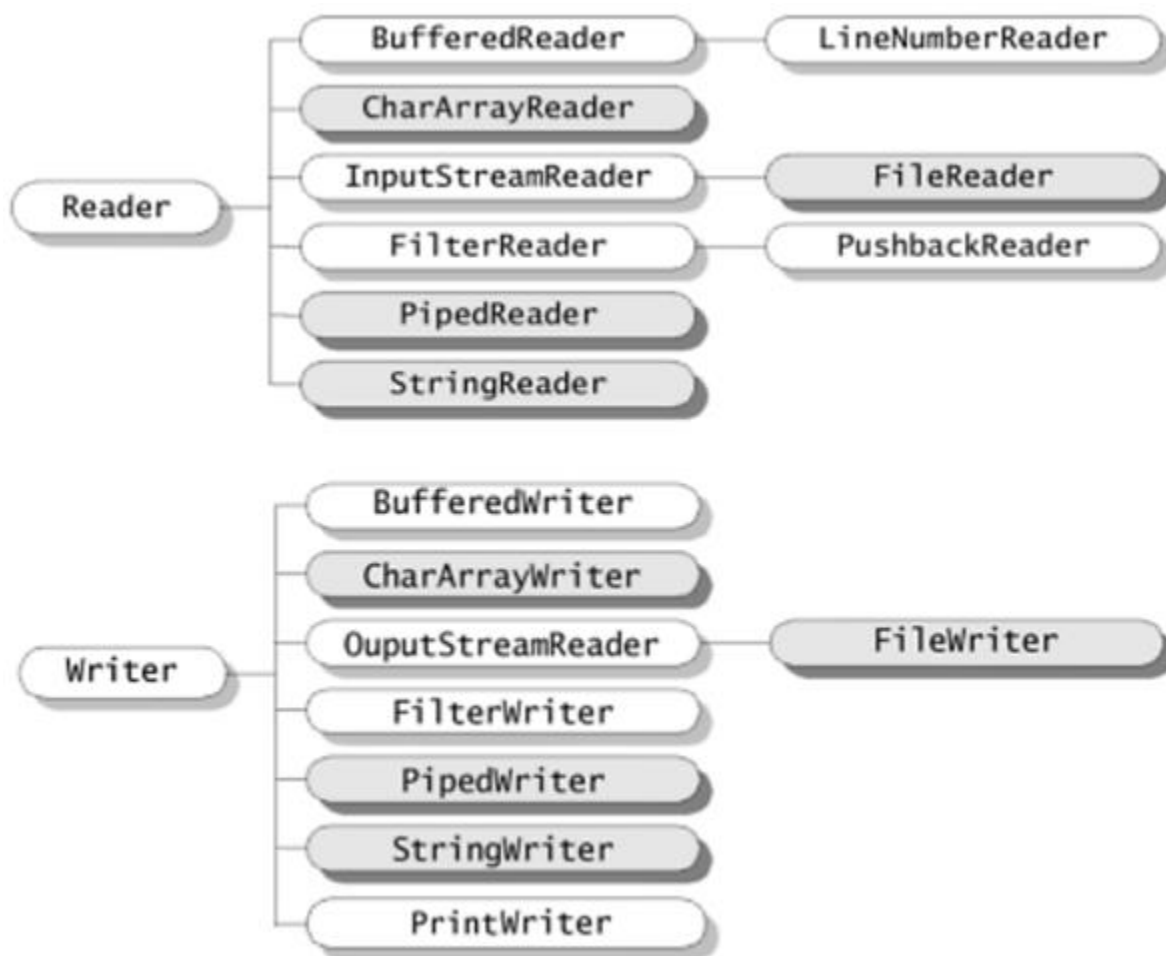


Рис 4. Иерархия классов потоков символов

Базовые классы для организации входных потоков *InputStream* и *Reader* определяют ряд схожих по названию и списку аргументов, но работающих с различными типами данных, методов, предназначенных для считывания информации из некоторого источника. Некоторые из таких методов представлены ниже.

<i>InputStream</i>	<i>Reader</i>
<code>int read()</code>	<code>int read()</code>
<code>int read(byte[] cbuf)</code>	<code>int read(char[] cbuf)</code>
<code>int read(byte[] cbuf, int offset, int length)</code>	<code>int read(char[] cbuf, int offset, int length)</code>

По аналогии с методами входных потоков, представленными выше, можно сопоставить одноименные методы классов для выходных потоков *OutputStream* и *Writer*, с помощью которых организуется запись данных. Эти методы имеют одинаковые названия, но отличаются количеством и типом своих аргументов.

<i>OutputStream</i>	<i>Writer</i>
<pre>void write(int c) void write(byte[] cbuf) void write(byte[] cbuf, int offset, int length)</pre>	<pre>void write(int c) void write(char[] cbuf) void write(char[] cbuf, int offset, int length)</pre>

Представленные выше методы в порядке их следования в списках позволяют:

1. считать / записать один байт (8 младших бит величины типа *int*) для *InputStream* и *OutputStream*, либо один символ UNICODE (16 младших бит величины типа *int*) для *Reader* и *Writer*;
2. считать в массив / записать из массива набор байт для *InputStream* и *OutputStream*, либо символов для *Reader* и *Writer*. Максимальное число байт/символов определяется длиной массива. Методы *read* входных потоков возвращают число реально считанных байт/символов;
3. аналогичны вторым в списках методам, но позволяют указать количество (*length*) байт/символов для чтения/записи и смещение на *offset* позиций в массиве *cbuf*.

Подклассы, представленные на рисунках 3 и 4, могут быть разделены на группы в зависимости от источника/приемника данных. Назначение данных подклассов приведено в таблице 1.

Таблица 1 Классы для работы с потоками данных и их назначение

Класс	Назначение
<i>FileInputStream</i> <i>FileOutputStream</i> <i>FileReader</i> <i>FileWriter</i>	Используются для создания потоков для чтения/записи информации из/в файл
<i>StringBufferInputStream</i> <i>StringReader</i> <i>StringWriter</i>	Используются для организации потоков считывания/записи информации из/в строку. При чтении байт с использованием <i>StringBufferInputStream</i> из каждого символа строки (2 байта) будет считан только младший байт. Данный класс считается устаревшим и для чтения информации из строки рекомендуется использовать <i>StringReader</i>
<i>ByteArrayInputStream</i> <i>ByteArrayOutputStream</i> <i>CharArrayReader</i> <i>CharArrayWriter</i>	Предназначены для считывания/записи данных из/в массив, расположенный в памяти

Класс	Назначение
<i>PipedInputStream</i> <i>PipedOutputStream</i> <i>PipedReader</i> <i>PipedWriter</i>	Позволяют организовать канал передачи данных между отдельными модулями (например, потоками выполнения) в рамках программы. Канал обмена данными включает пару связанных между собой потоков чтения и записи информации. Для связывания пары потоков используют либо конструкторы классов, либо метод <i>connect</i> .
<i>SequenceInputStream</i>	Позволяет объединить несколько входных потоков в один. Данные вычитываются последовательно в порядке добавления отдельных потоков в список.
<i>ObjectInputStream</i> <i>ObjectOutputStream</i>	Используются для чтения/записи из/в поток объектов классов, поддерживающих сериализацию (способность быть преобразованными в последовательность байт), для чего в них реализуется интерфейс <i>java.io.Serializable</i> .
<i>DataInputStream</i> <i>DataOutputStream</i>	Содержат методы для чтения/записи из/в поток данных всех примитивных типов (<i>int</i> , <i>double</i> и т.д.)
<i>LineNumberInputStream</i> <i>LineNumberReader</i>	Подсчитывают количество считанных из потока строк, а также позволяют перейти к некоторой строке. Под строкой при этом понимается набор байт, оканчивающийся либо '\n', либо '\r', либо их комбинацией '\r\n'. <i>LineNumberInputStream</i> считается устаревшим и рекомендуется использовать <i>LineNumberReader</i>
<i>BufferedInputStream</i> <i>BufferedOutputStream</i> <i>BufferedReader</i> <i>BufferedWriter</i>	Организуют буферизированные чтение/запись данных с/на некоторое устройство. Для минимизации количества обращений к устройству используется буфер байт, через который передаются данные. Это позволяет не обращаться к устройству при каждом вызове функций <i>read/write</i> .
<i>PushbackInputStream</i> <i>PushbackReader</i>	Используют встроенный буфер, позволяющий возвращать считанные данные обратно в поток. Это может быть полезно, если при чтении данных необходимо заглянуть вперед на несколько байтов или символов, для определения дальнейших действий.
<i>PrintStream</i> <i>PrintWriter</i>	Используются для записи в поток строк и строкового представления примитивных типов и объектов.
<i>InputStreamReader</i> <i>OutputStreamWriter</i>	Позволяют формировать мосты между потоками байт и символов. <i>InputStreamReader</i> считывает байты из <i>InputStream</i> и преобразует их в символы, используя заданную кодировку или кодировку по умолчанию. <i>OutputStreamWriter</i> преобразует символы в байты используя заданную кодировку или кодировку по умолчанию и записывает их в <i>OutputStream</i>

При создании потока с использованием одного из классов, перечисленных выше, поток автоматически открывается. После использования потока его необходимо закрыть путем вызова метода *close*.

Поскольку при выполнении обмена данными могут возникать различного рода сбои, практически все методы классов, организующие ввод-вывод информации, объявляются как потенциально генерирующие исключение, представленное классом *IOException* и его подклассами, которое относится к группе проверяемых исключений. Поэтому при вызове данных методов необходимо обеспечить обработку таких исключений.

При использовании потоков на практике для более удобной записи/чтения данных различного типа одни потоки могут служить “обертками” для других типов потоков. Это продемонстрировано в примере, приведенном на рисунке 3.2, в котором поток типа *DataInputStream* является “оберткой” для потока *FileInputStream*, поскольку предоставляет более удобный способ (метод *readDouble*) для считывания значений типа *double* из потока.

3 Коллекции

3.1 Иерархия базовых абстрактных классов и интерфейсов коллекций в Java

Для хранения и обработки наборов данных в Java могут использоваться коллекции. Коллекции предоставляют более широкие возможности по манипулированию наборами данных и не подвержены ограничениям, характерным для массивов, например, таким как фиксированная длина, задаваемая только при создании массива и возможность обращаться к элементам набора только по индексу. Особенностью, характерной для всех коллекций в Java, является то, что в качестве элементов, хранящихся внутри коллекций, могут использоваться ссылки на объекты, порожденные от класса *Object*. Хранение значений примитивных типов напрямую невозможно. Для хранения таких значений необходимо использовать соответствующие классы оболочки.

Классы и интерфейсы коллекций в Java принадлежат пакету *java.util*. Иерархия классов коллекций в Java содержит ряд базовых абстрактных классов и интерфейсов, определяющих поведение коллекций различного типа и позволяющих унифицировать работу с данными, содержащимися в коллекциях, независимо от деталей их представления. Данные классы и интерфейсы представлены на рисунке 5.

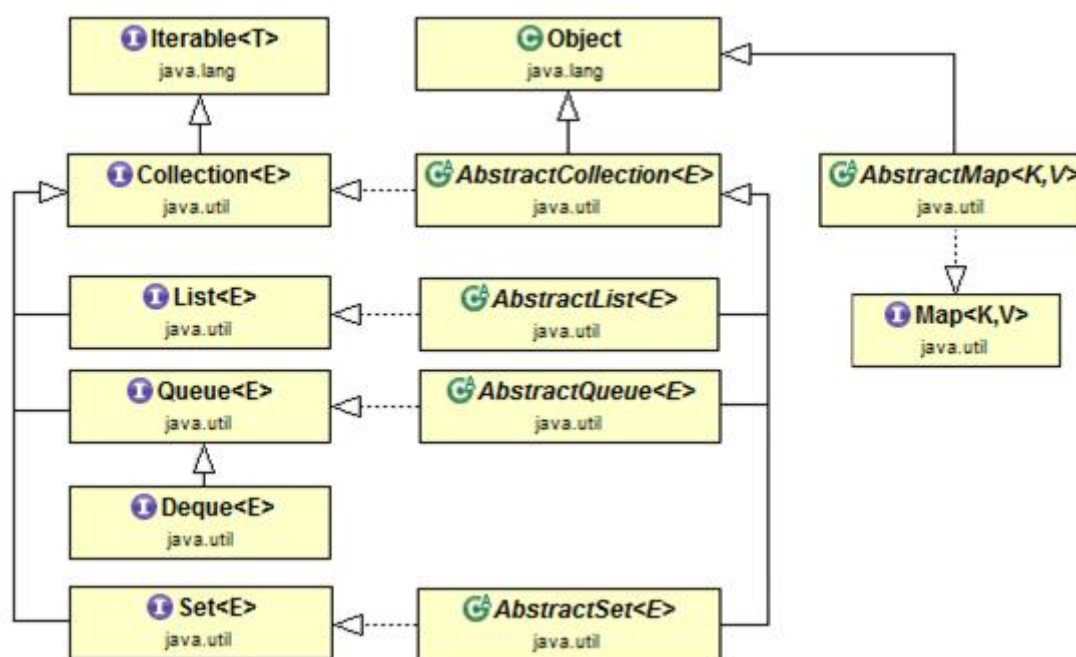


Рис 5. Иерархия базовых абстрактных классов и интерфейсов коллекций в Java

Как видно из приведенного выше рисунка все коллекции в Java можно разделить на две группы, во главе иерархий классов которых стоят абстрактные классы *AbstractCollection* и *AbstractMap*.

3.2 Коллекции, реализующие интерфейс *Collection*

Классы, порожденные от абстрактного класса *AbstractCollection*, предназначены для работы с набором объектов некоторого типа. Класс *AbstractCollection* реализует большую часть методов интерфейса *Collection* для упрощения создания подклассов, реализующих данный интерфейс, и предназначенных для создания уже конкретных типов коллекций. Методы интерфейса *Collection*, определяющие базовую функциональность коллекций, классы которых реализуют данный интерфейс, приведены в таблице 2.

Таблица 2 Методы интерфейса *Collection*

Имя метода	Назначение
<i>boolean add(T obj)</i>	Добавляет объект <i>obj</i> к вызывающей коллекции
<i>boolean addAll(Collection<? extends T> c)</i>	Добавляет объекты из коллекции <i>c</i> к вызывающей коллекции
<i>void clear()</i>	Удаляет все элементы из вызывающей коллекции
<i>boolean contains(Object obj)</i>	Возвращает <i>true</i> , если объект <i>obj</i> содержится в вызывающей коллекции, иначе – <i>false</i>
<i>boolean containsAll(Collection<?> c)</i>	Возвращает <i>true</i> , если все элементы коллекции <i>c</i> содержатся в вызывающей коллекции, иначе – <i>false</i>

Имя метода	Назначение
<i>boolean equals(Object obj)</i>	Возвращает <i>true</i> , если объект <i>obj</i> и объект вызывающей коллекции равны, иначе – <i>false</i>
<i>int hashCode()</i>	Возвращает хэш-код вызывающей коллекции
<i>boolean isEmpty()</i>	Возвращает <i>true</i> , если вызывающая коллекция пуста, иначе – <i>false</i>
<i>Iterator<T> iterator()</i>	Возвращает итератор для вызывающей коллекции
<i>boolean remove(Object obj)</i>	Удаляет одну ссылку на объект <i>obj</i> из вызывающей коллекции
<i>boolean removeAll(Collection<?> c)</i>	Удаляет все элементы, содержащиеся в коллекции <i>c</i> , из вызывающей коллекции
<i>boolean retainAll(Collection<?> c)</i>	Удаляет все элементы из вызывающей коллекции, кроме элементов, содержащиеся в коллекции <i>c</i>
<i>int size()</i>	Возвращает число элементов в вызывающей коллекции
<i>Object[] toArray()</i>	Создает и возвращает массив, содержащий ссылки на объекты, хранящиеся в вызывающей коллекции
<i><T> T[] toArray(T[] a)</i>	Возвращает массив, содержащий ссылки на объекты, хранящиеся в вызывающей коллекции, чей тип согласуется с типом элементов в массиве <i>a</i> . Если длина коллекции меньше или равна длине массива <i>a</i> , то возвращается ссылка на массив <i>a</i> , иначе создается новый массив длиной, равной длине коллекции и с базовым типом <i>T</i> . Если длина массива больше длины коллекции, то элементу массива, следующему за последним элементом, скопированным из коллекции присваивается <i>null</i> .

Как видно из рисунка 5 стандартные возможности класса *AbstractCollection*, расширяются в нескольких абстрактных подклассах, являющихся основой для последующего создания коллекций различного типа. Каждый из этих подклассов дополнительно к функциональности, наследуемой от интерфейса *Collection*, реализует свой собственный интерфейс, задающий действия с набором объектов, специфические для данного типа коллекции.

3.2.1 Списки

Абстрактный класс *AbstractList* является основой для создания коллекций, которые называются *списками*.

Отличительными чертами списков являются:

1. возможность хранения нескольких дубликатов ссылки на один и тот же объект;

2. работа с элементами коллекции с использованием индекса, значение которого отсчитывается от 0.

Дополнительная функциональность списков определяется рядом методов, определенных в интерфейсе *List*, которые представлены в таблице 3.

Таблица 3 Методы интерфейса *List*

Имя метода	Назначение
<i>void add(int index, T obj)</i>	Вставляет объект <i>obj</i> в вызывающий список в позицию с индексом <i>index</i> . Все элементы, ранее располагавшиеся в списке, начиная с позиции <i>index</i> сдвигаются вправо на 1 позицию.
<i>boolean addAll(int index, Collection<? extends T> c)</i>	Вставляет все элементы коллекции <i>c</i> в вызывающий список начиная с позиции <i>index</i> . Все элементы, ранее располагавшиеся в списке, начиная с позиции <i>index</i> сдвигаются вправо.
<i>T get(int index)</i>	Возвращает ссылку на объект с индексом <i>index</i> .
<i>int indexOf(Object obj)</i>	Возвращает индекс первой ссылки на объект <i>obj</i> . Если ссылка на <i>obj</i> отсутствует, то возвращается -1.
<i>int lastIndexOf(Object obj)</i>	Возвращает индекс последней ссылки на объект <i>obj</i> . Если ссылка на <i>obj</i> отсутствует, то возвращается -1.
<i>ListIterator<T> listIterator();</i> <i>ListIterator<T> listIterator(int index)</i>	Возвращает итератор типа <i>ListIterator</i> , установленный в начало списка (метод без аргументов) или в позицию <i>index</i> (метод с одним аргументом)
<i>T remove(int index)</i>	Удаляет элемент на позиции <i>index</i> из вызывающего списка и возвращает ссылку на удаленный элемент. Индексы всех элементов начиная с позиции <i>index+1</i> уменьшаются на 1, тем самым уплотняя список
<i>T set(int index, T obj)</i>	Присваивает ссылку на объект <i>obj</i> элементу списка с индексом <i>index</i>
<i>List<T> subList(int start, int end)</i>	Создает отображение части вызывающего списка, включающее элементы начиная с позиции <i>start</i> и заканчивая позицией <i>end-1</i> . Данное отображение имеет тип <i>List</i> , и позволяет напрямую работать с элементами исходного вызывающего списка (значения, измененные через отображение, изменяются в соответствующих позициях исходного списка)

Пакет *java.util* содержит ряд подклассов класса *AbstractList*, реализующих различные списки, которые могут быть использованы для хранения наборов данных. Иерархия этих классов представлена на рисунке 6. Классы *ArrayList* и *Vector*, а также его подкласс *Stack* унаследованы непосредственно от абстрактного класса *AbstractList*. Класс *LinkedList* является подклассом для

абстрактного класса *AbstractSequentialList*, который в свою очередь наследуется от *AbstractList*.

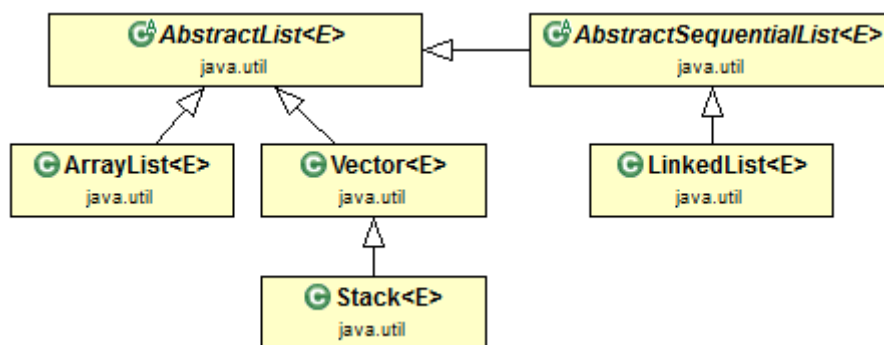


Рис 6. Иерархия классов для создания списков в Java

Класс *ArrayList* применяется для организации списка, поведение которого аналогично динамическому массиву. Такой массив может по мере необходимости увеличивать свой размер в отличие от обычных массивов в Java, для которых длина фиксирована и должна задаваться при создании массива. Изначально список *ArrayList* создается с некоторой начальной емкостью (количеством ячеек, доступных для хранения элементов в списке), которая может быть определена как параметр конструктора. Далее по мере добавления элементов к списку если первоначально заданной емкости становится недостаточно, то она автоматически увеличивается. Емкость также можно увеличить, используя метод *ensureCapacity*. Если часть объектов была удалена из списка, то его емкость может быть уменьшена с использованием метода *trimToSize* до размера, необходимого для хранения оставшегося числа элементов.

Класс *Vector* имеет то же назначение, что и класс *ArrayList*, однако в отличие от последнего может безопасно использоваться в многопоточных приложениях. В случае, если поддержка многопоточности не требуется, рекомендуется использовать более быстрый *ArrayList*.

Класс *Stack* реализует стек объектов, работающий по принципу “последний-вошел-первый-вышел” (LIFO – last-in-first-out). Данный класс добавляет к возможностям класса *Vector* пять операций, которые обеспечивают поведение, описывающее стек. Методы *push* и *pop* позволяют добавить/извлечь верхний элемент в/из стека, соответственно. Метод *peek* позволяет получить верхний элемент стека, без его извлечения. Метод *search(obj)* находит расстояние от вершины стека до позиции объекта *obj*. Расстояние до первого объекта в стеке считается равным 1. Если объект отсутствует, то возвращается значение -1. Метод *empty* позволяет проверить, является ли стек пустым.

Абстрактный класс *AbstractSequentialList* является основой для организации списков с последовательным доступом к элементам в порядке их

следования в списке. Такие списки обеспечивают быстрое выполнение операций добавления и удаления элементов по сравнению со списками, основанными на массивах, однако доступ к элементам в случайном порядке в таких списках медленнее.

Класс *LinkedList* реализует двусвязный список, в котором каждый из элементов кроме непосредственно данных содержит в себе ссылки на предыдущий и последующий элементы списка. По этой причине операции добавления и удаления элементов, для которых требуется лишь перестановка ссылок соседних элементов, проходят достаточно быстро (смотри рисунок 7).



Рис 7. Двухнаправленный список и принципы выполнения операций добавления и удаления его элементов

Доступ к первому и последнему элементам такого списка выполняются достаточно быстро, так как ссылки на них постоянно хранятся. Скорость случайного доступа к элементам будет уступать спискам, основанным на массивах.

3.2.2 Очереди

Абстрактный класс *AbstractQueue* может быть использован в качестве базового для создания коллекций с расположением элементов в определенном порядке, необходимом для последующей обработки. Такие коллекции называют *очередями*. Базовая функциональность для такого рода коллекций определена в интерфейсе *Queue*, методы которого представлены в таблице 4.

В рамках пакета *java.util* от класса *AbstractQueue* наследуется класс *PriorityQueue*, который позволяет создавать очереди, элементы которых упорядочены либо в их естественном порядке (с использованием интерфейса *Comparable*, который должны реализовывать элементы очереди), либо с использованием объекта-компаратора, реализующего интерфейс *Comparator*,

ссылка на который передается в конструктор класса *PriorityQueue*. Голова очереди указывает на наименьший элемент.

Таблица 4 Методы интерфейса *Queue*

Имя метода	Назначение
<i>boolean add(T obj)</i> <i>boolean offer(T obj)</i>	Вставляют объект <i>obj</i> в очередь если это возможно без нарушения ограничений, связанных с ее емкостью. Если элемент не может быть добавлен, то метод <i>add</i> генерирует исключение, а метод <i>offer</i> возвращает <i>false</i> без генерации исключения
<i>T remove()</i> <i>T poll()</i>	Удаляют из очереди головной элемент и возвращают ссылку на него. Если перед вызовом очередь была пуста, то метод <i>remove</i> генерирует исключение, а метод <i>poll</i> возвращает <i>null</i> , без генерации исключения
<i>T element()</i> <i>T peek()</i>	Возвращают ссылку на головной элемент очереди, без его удаления из очереди. Если перед вызовом очередь была пуста, то метод <i>element</i> генерирует исключение, а метод <i>peek</i> возвращает <i>null</i> , без генерации исключения

Если несколько элементов претендуют на роль наименьшего, то голова очереди может указывать на любой из них. Методы интерфейса *Queue*, представленные в таблице 4, работают с элементом в голове очереди. Количество элементов в очереди *PriorityQueue* не ограничивается и место для хранения ссылок на элементы в очереди резервируется автоматически при добавлении новых элементов.

Интерфейс *Deque* (double ended queue, произносится как «Deck») определяет базовую функциональность для организации очередей, позволяющих добавлять и читать данные с обеих сторон. Это позволяет поддерживать одновременно два режима работы: FIFO (First In First Out) и LIFO (Last In First Out). Методы интерфейса *Deque* описаны в таблице 5.

Таблица 5 Методы интерфейса *Deque*

Имя метода	Назначение
<i>void addFirst(T obj)</i> <i>boolean offerFirst(obj)</i> <i>void addLast(T obj)</i> <i>boolean offerLast(obj)</i>	Вставляют объект <i>obj</i> в начало (<i>addFirst</i> и <i>offerFirst</i>)/конец (<i>addLast</i> и <i>offerLast</i>) очереди, если это не нарушает ее емкость. Если элемент нельзя добавить, то <i>addFirst</i> и <i>addLast</i> генерируют исключение, а <i>offerFirst</i> и <i>offerLast</i> возвращают <i>false</i> без генерации исключения
<i>T removeFirst()</i> <i>T pollFirst()</i> <i>T removeLast()</i> <i>T pollLast()</i>	Удаляют из очереди первый (<i>removeFirst</i> и <i>pollFirst</i>)/последний (<i>removeLast</i> и <i>pollLast</i>) элемент и возвращают ссылку на него. Если очередь была пуста, то <i>removeFirst</i> и <i>removeLast</i> генерируют исключение, а <i>pollFirst</i> и <i>pollLast</i> возвращают <i>null</i> , без генерации исключения

Имя метода	Назначение
<i>T removeFirst()</i> <i>T pollFirst()</i> <i>T removeLast()</i> <i>T pollLast()</i>	Удаляют из очереди первый (<i>removeFirst</i> и <i>pollFirst</i>)/последний (<i>removeLast</i> и <i>pollLast</i>) элемент и возвращают ссылку на него. Если перед вызовом очередь была пуста, то методы <i>removeFirst</i> и <i>removeLast</i> генерируют исключение, а методы <i>pollFirst</i> и <i>pollLast</i> возвращают <i>null</i> , без генерации исключения
<i>T getFirst()</i> <i>T peekFirst()</i> <i>T getLast()</i> <i>T peekLast()</i>	Возвращают ссылку на первый (<i>getFirst</i> и <i>peekFirst</i>)/последний (<i>getLast</i> и <i>peekLast</i>) элементы очереди, без его удаления из очереди. Если перед вызовом очередь была пуста, то методы <i>getFirst</i> и <i>getLast</i> генерируют исключение, а методы <i>peekFirst</i> и <i>peekLast</i> возвращают <i>null</i> , без генерации исключения
<i>Iterator<T></i> <i>descendingIterator()</i>	Возвращает итератор по элементам очереди в обратном порядке
<i>T pop()</i>	позволяет извлечь первый элемент из очереди (эквивалентен <i>removeFirst()</i>)
<i>void push(T obj)</i>	позволяет добавить первый элемент в очередь (эквивалентен <i>addFirst(obj)</i>)
<i>boolean</i> <i>removeFirstOccurrence(Object</i> <i>obj)</i> <i>boolean</i> <i>removeLastOccurrence(Object</i> <i>obj)</i>	Позволяют удалить из очереди первую (<i>removeFirstOccurrence</i>)/последнюю (<i>removeLastOccurrence</i>) ссылку на объект эквивалентный объекту <i>obj</i> . Эквивалентность проверяется путем вызова <i>obj.equals()</i> .

Поскольку интерфейс *Deque* наследуется от *Queue*, то тип любого объекта, реализующего *Deque* может быть преобразован к *Queue*. В этом случае очередь типа *Queue* будет работать по принципу “первый -вошел-первый-вышел” (FIFO – first-in-first-out). При этом соблюдается соответствие методов интерфейсов *Queue* и *Deque*, приведенное в таблице 6.

Таблица 6 Соответствие методов интерфейсов *Queue* и *Deque*

Метод интерфейса <i>Queue</i>	Метод интерфейса <i>Deque</i>
<i>boolean add(T obj)</i>	<i>void addLast(T obj)</i>
<i>boolean offer(T obj)</i>	<i>boolean offerLast(obj)</i>
<i>T remove()</i>	<i>T removeFirst()</i>
<i>T poll()</i>	<i>T pollFirst()</i>
<i>T element()</i>	<i>T getFirst()</i>
<i>T peek()</i>	<i>T peekFirst()</i>

В рамках пакета `java.util` интерфейс *Deque* реализован в классах *LinkedList* и *ArrayDeque*. В то время как класс *LinkedList*, описанный выше, можно рассматривать как двунаправленную очередь, где каждый элемент содержит ссылки на соседние элементы, класс *ArrayDeque*, реализует аналогичную очередь, основанную на массиве. Объект *ArrayDeque* может работать быстрее при использовании в качестве стека, чем объект класса *Stack*, а также быстрее объекта *LinkedList*, при использовании в качестве очереди. В то же время *ArrayDeque* не допускает хранения значений *null*.

Кроме классов, принадлежащих пакету `java.util`, интерфейсы *Queue* и *Deque* реализуются в ряде классов, которые содержатся в пакете `java.util.concurrent` и, в отличие от перечисленных выше классов, могут безопасно использоваться в многопоточных приложениях.

3.2.3 Множества

Абстрактный класс *AbstractSet* является основой для создания коллекций, представляющих собой множество неповторяющихся элементов. Иерархия классов и интерфейсов, которые принадлежат пакету `java.util` и могут быть задействованы для создания таких коллекций представлены на рисунке 8.

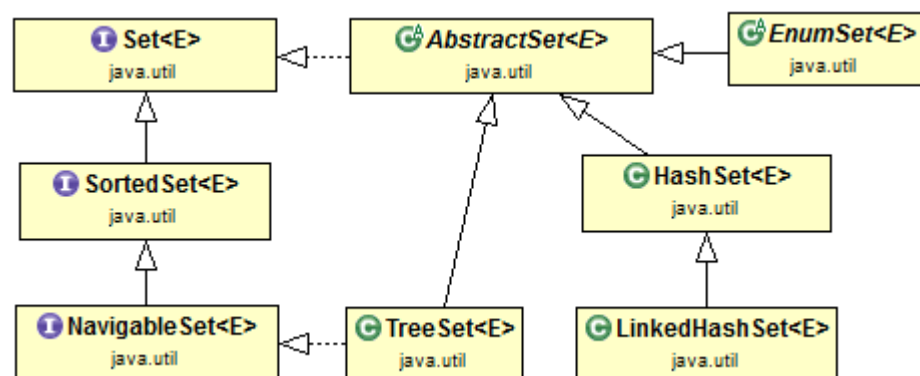


Рис 8. Иерархия классов для создания множеств неповторяющихся элементов в Java

Как видно из рисунка 8, представленная иерархия включает три интерфейса *Set*, *SortedSet* и *NavigableSet*.

Интерфейс *Set* наследуется от интерфейса *Collection*, не добавляя к нему новых методов, и служит лишь для объявления поведения коллекции, не допускающей дублирования элементов. Как следствие метод *add()* для классов, реализующих *Set* должен возвращать *false* при попытке добавить элемент, который уже есть в множестве.

Интерфейс *SortedSet* унаследован от *Set* и определяет множество, элементы которого отсортированы по возрастанию. Методы, определенные непосредственно в *SortedSet* представлены в таблице 7.

Таблица 7 Методы интерфейса *SortedSet*

Имя метода	Назначение
<i>Comparator<? super T></i> <i>comparator()</i>	Возвращает компаратор вызывающего отсортированного множества. Если множество использует естественное упорядочение, то возвращается <i>null</i>
<i>T first ()</i> <i>T last()</i>	Возвращает первый (<i>first</i>)/последний (<i>last</i>) элемент отсортированного множества
<i>SortedSet<T> headSet(T end)</i>	Возвращает объект типа <i>SortedSet</i> , содержащий элементы исходного множества, которые меньше <i>end</i>
<i>SortedSet<T> subSet(T start, T end)</i>	Возвращает объект типа <i>SortedSet</i> , содержащий элементы исходного множества, которые больше либо равны <i>start</i> и меньше <i>end</i>
<i>SortedSet<T> tailSet(T start)</i>	Возвращает объект типа <i>SortedSet</i> , содержащий элементы исходного множества, которые больше либо равны <i>start</i>

Интерфейс *NavigableSet* расширяет *SortedSet* путем добавления к его возможностям набора методов, позволяющих выполнять перемещение по элементам множества и поиск элементов, например, расположенных рядом с заданным. Данные методы представлены в таблице 8

Таблица 8 Методы интерфейса *NavigableSet*

Имя метода	Назначение
<i>T lower(T obj)</i> <i>T floor(T obj)</i> <i>T ceiling(T obj)</i> <i>T higher(obj)</i>	Возвращает элемент множества меньший (<i>lower</i>)/ меньше, либо равный (<i>floor</i>)/ больше, либо равный (<i>ceiling</i>)/ больший (<i>higher</i>) чем <i>obj</i> . Если такого элемента не найдено, то возвращается <i>null</i>
<i>Iterator<T></i> <i>descendingIterator();</i> <i>Iterator<T> iterator()</i>	Возвращает итератор по элементам множества в обратном (<i>descendingIterator</i>)/ прямом (<i>iterator</i>) порядке
<i>NavigableSet<T></i> <i>descendingSet()</i>	Возвращает ссылку на объект типа <i>NavigableSet</i> , позволяющий просматривать элементы данного множества в обратном порядке. Поскольку объект, возвращаемый методом <i>descendingSet</i> , и объект, вызывающий данный метод, ссылаются на один и тот же набор элементов, то изменения, сделанные в наборе через один из этих объектов, отражаются на содержимом второго
<i>SortedSet<T></i> <i>headSet(T end, boolean inclusive)</i>	Возвращает объект типа <i>NavigableSet</i> , содержащий элементы исходного множества, которые меньше (<i>inclusive=false</i>) / меньше либо равны (<i>inclusive= true</i>) <i>end</i>

Имя метода	Назначение
<i>NavigableSet<T></i> <i>subSet(T start, boolean startInclusive, T end, boolean endInclusive)</i>	Возвращает объект типа <i>NavigableSet</i> , содержащий элементы исходного множества, которые больше (<i>startInclusive=false</i>) / больше либо равны (<i>startInclusive=true</i>) <i>start</i> и меньше (<i>endInclusive=false</i>) / меньше равны (<i>endInclusive=true</i>) <i>end</i>
<i>SortedSet<T></i> <i>tailSet(T start, boolean inclusive)</i>	Возвращает объект типа <i>SortedSet</i> , содержащий элементы исходного множества, которые больше (<i>inclusive=false</i>)/ больше либо равны (<i>inclusive= true</i>) <i>start</i>
<i>T pollFirst()</i> <i>T pollLast()</i>	Возвращает первый (<i>pollFirst</i>)/последний (<i>pollLast</i>) элемент множества, удаляя его из набора

Абстрактный класс *EnumSet* является специализированным классом для организации множеств на основе перечислений. Все элементы такого множества должны принадлежать одному и тому же перечислению, определенному до создания множества. Особенностью данного множества с точки зрения его внутренней реализации является то, что они реализованы на базе векторов битов, что делает их одновременно компактными и быстрыми. Множество *EnumSet* не может содержать *null*.

Класс *HashSet* реализует неупорядоченное множество, для хранения элементов которого используется хэш-таблица (объект класса *HashMap*), в которой каждому элементу ставится в соответствие некоторый ключ. В качестве ключей используются хэш-коды элементов, добавляемых в *HashSet*, которые вычисляются путем вызова метода *hashCode*. Получаемые таким образом ключи используются только внутри объекта *HashSet* и не доступны для внешних объектов. При хранении в *HashSet* объектов собственных классов важно правильно реализовать в них методы *hashCode* и *equals* для правильного определения одинаковых объектов при их добавлении в *HashSet*. Преимуществом использования *HashSet* является постоянное время выполнения основных операций, таких как *add*, *remove*, *size* даже для больших наборов.

Класс *LinkedHashSet* является подклассом класса *HashSet*. Его основное отличие от *HashSet* состоит в том, что он хранит элементы в том порядке, в котором они были добавлены в множество, что позволяет организовать упорядоченный набор элементов.

Класс *TreeSet* реализует множество, в котором для хранения элементов используется иерархическая (древовидная) структура. Данный класс реализует интерфейс *NavigableSet* и хранит элементы в отсортированном по возрастанию виде. При добавлении объекта в дерево он сразу же размещается в

необходимую позицию с учетом сортировки. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

3.2.4 Организация перебора элементов коллекций, реализующих интерфейс *Collection*

При работе с коллекциями, реализующими интерфейс *Collection*, часто необходимо организовывать перебор их элементов. Для этого используются следующие возможности интерфейса *Iterable*, от которого унаследован интерфейс *Collection*:

1. получение при помощи вызова метода *iterator* и последующее использование ссылки на объект-итератор, реализующий интерфейс *Iterator*;
2. использование метода *foreach*.

При использовании для перебора элементов коллекции итератора, могут быть задействованы методы интерфейса *Iterator*, представленные в таблице 9.

Таблица 9 Методы интерфейса *Iterator*

Имя метода	Назначение
<i>boolean hasNext()</i>	Возвращает <i>true</i> если в коллекции присутствует следующий элемент, иначе <i>false</i>
<i>T next()</i>	Возвращает ссылку на следующий элемент, а при его отсутствии генерирует исключение
<i>void remove()</i>	Удаляет из коллекции последний элемент, возвращенный методом <i>next</i> . Если вызову данного метода не предшествовал вызов <i>next</i> , то генерируется исключение

Ниже приведен пример кода, использующего итератор с применением всех его методов.

```
List<Integer> list = new ArrayList<Integer>();
for(int i=0; i<5; i++) list.add(i);

//Получаем объект-итератор
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    System.out.println(itr.next()); //выводим элемент в консоль
    itr.remove(); //удаляем выведенный элемент
}
```

Наряду с интерфейсом *Iterator*, коллекции, реализующие интерфейс *List*, позволяют использовать обладающий большими возможностями итератор типа *ListIterator*, доступ к которому можно получить, вызвав определенный в *List*

метод *listIterator*. Интерфейс *ListIterator* порожден от интерфейса *Iterator* и дополняет его следующими возможностями:

1. позволяет добавлять элементы в коллекцию (метод *add*);
2. позволяет просматривать элементы в обратном порядке (методы *hasPrevious* и *previous*);
3. позволяет получать индексы элементов (методы *nextIndex* и *previousIndex*);
4. позволяет заменять элемент в коллекции (метод *set*).

Принцип использования *ListIterator* аналогичен использованию *Iterator*.

При использовании для перебора элементов коллекции метода *foreach* в качестве его аргумента используется объект класса, реализующего функциональный интерфейс *java.util.function.Consumer*, в котором определен метод *consume*, принимающий в качестве аргумента элемент из коллекции. Для получения объекта класса, реализующего *Consumer*, удобно использовать лямбда-выражения.

Ниже представлен программный код для вывода строкового представления элементов списка *list* в консоль, использующий метод *foreach* и полностью эквивалентный ранее приведенному примеру на основе итератора:

```
List<Integer> list = new ArrayList<Integer>();
for(int i=0; i<5; i++) list.add(i);

//используем foreach совместно с лямбда-выражением
//для вывода элементов в консоль
list.forEach(item -> System.out.println(item));
```

3.3 Коллекции, реализующие интерфейс Map

Классы, порожденные от абстрактного класса *AbstractMap*, позволяют создавать *карты отображений*, которые хранят пары «ключ–значение». Как ключ, так и значение являются объектами. Значение в каждой паре можно рассматривать как элемент хранимых данных, а ключ как идентификатор, по которому ведется поиск соответствующего значения. Ключи в рамках карты должны быть уникальными, в то время как значения могут повторяться. Иерархия интерфейсов и классов, которые принадлежат пакету *java.util* и могут быть задействованы для создания карт отображений представлены на рисунке 9.

Базовая функциональность карт отображений определена интерфейсом *Map*, некоторые методы которого представлены в таблице 10. Как видно из таблицы 10, методы интерфейса *Map* обеспечивают добавление и удаление данных в карте, получение значений по известному ключу, а также получение коллекций ключей и значений, содержащихся в карте.

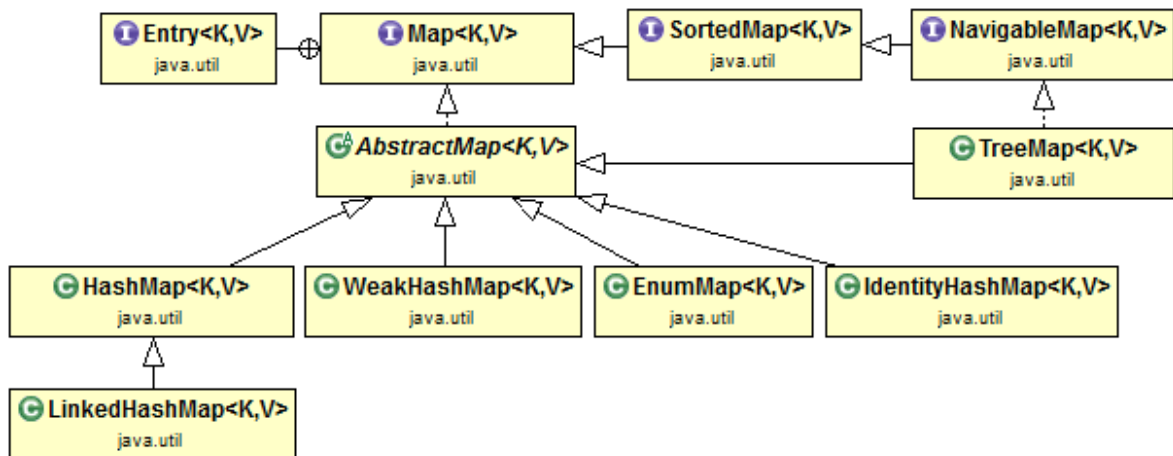


Рис 9. Иерархия классов для создания карт отображений в Java

Таблица 10 Некоторые методы интерфейса *Map*

Имя метода	Назначение
<i>void clear()</i>	Удаляет все пары ключ/значение из вызывающей карты
<i>boolean containsKey(Object k)</i>	Возвращает <i>true</i> , если вызывающая карта содержит объект <i>k</i> в качестве ключа, иначе возвращает <i>false</i>
<i>boolean containsValue(Object v)</i>	Возвращает <i>true</i> , если вызывающая карта содержит объект <i>v</i> в качестве значения, иначе возвращает <i>false</i>
<i>Set<Map.Entry<K,V>> entrySet()</i>	Возвращает объект типа <i>Set</i> , содержащий объекты типа <i>Map.Entry</i> , каждый из которых обеспечивает доступ к одной паре «ключ–значение»
<i>V get(Object k)</i>	Возвращает значение, связанное с ключом <i>k</i>
<i>boolean isEmpty()</i>	Возвращает <i>true</i> , если вызывающая карта пуста, иначе – <i>false</i>
<i>Set<K> keySet()</i>	Возвращает объект типа <i>Set</i> , который содержит множество ключей вызывающей карты
<i>V put(K k, V v)</i>	Помещает значение <i>v</i> в вызывающую карту и ставит ему в соответствие ключ <i>k</i> , переписывая любое предыдущее значение, связанное с ключом. Возвращает <i>null</i> , если ключ еще не существует. Иначе — предыдущее значение, связанное с ключом
<i>void putAll(Map<? extends K, ? extends V> m)</i>	Помещает все содержимое карты <i>m</i> в вызывающую карту
<i>V remove(Object k)</i>	Удаляет из карты пару «ключ–значение», чей ключ равен <i>k</i>
<i>int size()</i>	Возвращает размер (число пар ключ/значение) вызывающей карты
<i>Collection<V> values()</i>	Возвращает коллекцию ссылок на значения вызывающей карты в виде объекта типа <i>Collection</i> . Измерения, выполняемые через данный объект, приводят к измерениям в исходной карте

Имя метода	Назначение
<i>void forEach (BiConsumer<? super K, ? super V> action)</i>	Позволяет выполнить некоторую обработку для каждой пары ключ/значение, хранящейся в карте. Вид обработки определяется классом, который реализует интерфейс <i>BiConsumer</i> , и использовался для создания объекта <i>action</i> .

Если коллекция ключей и значений получена при помощи метода *entrySet*, то каждый элемент такой коллекции будет содержать пары «ключ–значение», представленные объектами класса, реализующего интерфейс *Map.Entry*. Методы интерфейса *Map.Entry* приведены в таблице 11.

Таблица 11 Методы интерфейса *Map.Entry*

Имя метода	Назначение
<i>boolean equals(Object obj)</i>	Возвращает <i>true</i> , если объект <i>obj</i> является <i>Map.Entry</i> -объектом, чей ключ и значение равны соответствующим элементам вызывающего объекта
<i>K getKey()</i>	Возвращает ключ для вызывающего объекта <i>Map.Entry</i>
<i>V getValue()</i>	Возвращает значение для вызывающего объекта <i>Map.Entry</i>
<i>int hashCode()</i>	Возвращает хэш-код вызывающего объекта <i>Map.Entry</i>
<i>V setValue(V v)</i>	Устанавливает значение для вызывающего объекта <i>Map.Entry</i> в <i>v</i>

3.3.1 Разновидности коллекций на основе интерфейса *Map*

Как видно из рисунка 10, в зависимости от положения в представленной на нем иерархии, можно выделить ряд разновидностей карт на основе базового интерфейса *Map*. Кроме этого, данная иерархия содержит несколько дополнительных интерфейсов с расширенной функциональностью.

Интерфейс *SortedMap* порожден от интерфейса *Map* и определяет карту, элементы которой отсортированы по возрастанию их ключей. Методы, определенные в *SortedMap*, схожи по своему назначению с методами класса *SortedSet* и добавляют следующие возможности по сравнению с *Map*:

1. получение объекта компаратора, используемого картой для сравнения элементов (метод *comparator*);
2. получение ключей первой и последней пар, хранящихся в карте (методы *firstKey* и *lastKey*);
3. создание подкарт, содержащих часть элементов исходной карты (методы *headMap*, *subMap*, *tailMap*).

Интерфейс *NavigableMap* добавляет к *SortedMap* методы, позволяющие выполнять поиск элементов, например, расположенных рядом с заданным,

получать подкарты, с подбором элементов из исходной карты по более гибким условиям, а также получать набор ключей карты в виде объекта *NavigableSet*.

Класс *HashMap* представляет собой карту отображений, основанную на хэш-таблице. Для определения позиции во внутреннем хранилище, где будет храниться пара «ключ–значение» применяется алгоритм, использующий хэш-код ключа, и обеспечивающий постоянное время доступа к элементам карты даже при относительно большом числе хранимых элементов. Карта допускает ключ и значение равные *null*. *HashMap* не гарантирует совпадение порядка добавления элементов в карту с порядком их считывания итератором, доступным после преобразования карты в объект типа *Set*.

Класс *LinkedHashMap* является подклассом *HashMap* и хранит для каждого элемента карты ссылки на соседние элементы, в том порядке, в котором они добавлялись в карту. Это требует несколько больших затрат памяти, но в то же время позволяет обеспечить упорядоченный доступ к элементам карты. Порядок прохода по элементам итератором будет совпадать с порядком их добавления в карту.

Класс *WeakHashMap* отличается от *HashMap* тем, что хранит ссылки на ключи в виде так называемых слабых ссылок, которые отличаются от обычных тем, что их наличие не препятствует удалению объекта сборщиком мусора. В результате если некоторый объект был добавлен в *WeakHashMap* в роли ключа и все обычные ссылки на него вне карты более не существуют, то компонент карты с данным ключом будет автоматически удален из карты, а сам ключ уничтожен сборщиком мусора.

Класс *IdentityHashMap* отличается от *HashMap* тем, что сравнение объектов-ключей при работе с элементами такой карты выполняется не при помощи функции *equals* (*k1.equals(k2)*) а путем сравнения ссылок на объекты (*k1==k2*). Таким образом, в карте *IdentityHashMap*, уникальность ключа определяется не по содержимому объектов, а по их адресу в памяти.

Класс *TreeMap* использует для внутреннего хранения элементов карты древообразную структуру. Объекты хранятся в отсортированном по возрастанию порядке по ключу. Применение для внутреннего хранения древообразной структуры при наличии сортировки позволяет быстро выполнять операции поиска ключа, получения и записи значений, а также удаления компонентов.

Класс *EnumMap* представляет собой специальную реализацию карты отображений, где в качестве ключей могут использоваться только элементы одного перечисления. Данные карты имеют внутреннее представление в виде массива и являются очень компактными и эффективными. В таких картах не допускается использование *null* в качестве ключа.

3.3.2 Организация перебора элементов коллекций, реализующих интерфейс Map

Для перебора элементов карт, реализующих интерфейс *Map*, можно воспользоваться их взаимосвязью с коллекциями, реализующими интерфейс *Collection*. Так, например, воспользовавшись методом *entrySet* интерфейса *Map* можно получить содержимое карты в виде множества объектов типа *Map.Entry*, каждый из которых содержит пару ключ/значение. Далее можно организовать перебор элементов этого множества одним из способов, описанных в разделе 3.2.4. Применение описанного выше подхода можно продемонстрировать на следующем примере:

```
//Создаем карту
Map<Integer, String> clients = new TreeMap();
clients.put(1, "Андрей");
clients.put(2, "Марина");
clients.put(3, "Татьяна");
clients.put(4, "Тимофей");
clients.put(5, "Татьяна");

//Получаем из карты множество пар ключ/значение
Set<Map.Entry<Integer, String>> clientsSet = clients.entrySet();

//Перебираем содержимое карты при помощи итератора
Iterator<Map.Entry<Integer, String>> itr = clientsSet.iterator();
while(itr.hasNext()) {
    Map.Entry<Integer, String> client = itr.next();
    System.out.println("ID=" + client.getKey() +
        " "; Имя: " + client.getValue());
    itr.remove();
}
```

В приведенном выше примере карта *clients* преобразуется путем вызова метода *entrySet* в множество *clientsSet* пар ключ/значение, после чего выполняется перебор элементов множества с использованием итератора.

Более простым способом перебора значений карт во многих случаях может оказаться использование метода *foreach* интерфейса *Map*. При этом обычно используется лямбда-выражение, в качестве параметров которого будут переданы ключ и значение обрабатываемой пары из карты. При использовании данного подхода приведенный выше пример примет следующий вид:

```
Map<Integer, String> clients = new TreeMap();
clients.put(1, "Андрей");
clients.put(2, "Марина");
clients.put(3, "Татьяна");
clients.put(4, "Тимофей");
clients.put(5, "Татьяна");

clients.forEach((id, clientName)->{
    System.out.println("ID=" + id +
        " "; Имя: " + clientName);
});
```

4 Создание модульных приложений в Java

Начиная с JDK 9 в Java появилась возможность создавать приложения, использующие модули. *Модуль* представляет собой совокупность программного кода и данных.

Использование модулей в Java-приложении позволяет:

1. определять связи и зависимости в коде Java-приложения;
2. управлять правами доступа к пакетам модуля и их содержимому;
3. ограничивать набор стандартных библиотек для создаваемого приложения, лишь теми, которые ему действительно необходимы.

Все модули, которые могут использоваться в приложении на Java можно разделить на несколько типов, представленных на рисунке 10.



Рис 10. Типы модулей в приложении на Java

Как видно из рисунка 10 все модули можно разделить на безымянный модуль и именованные модули.

Безымянный модуль используется для размещения пакетов и классов, загружаемых на пути к классам. Для размещения Java кода на пути к классам при использовании IDE Eclipse, необходимо добавить имя проекта (закладка «Java Build Path→Projects») или .JAR файл (закладка «Java Build Path→Libraries») в список «Classpath» (рисунок 11). Классы, загруженные в безымянный модуль, имеют доступ ко всем открытым классам и интерфейсам

других именованных модулей, загруженных в приложение, как во время компиляции, так и по время выполнения через Reflection API.

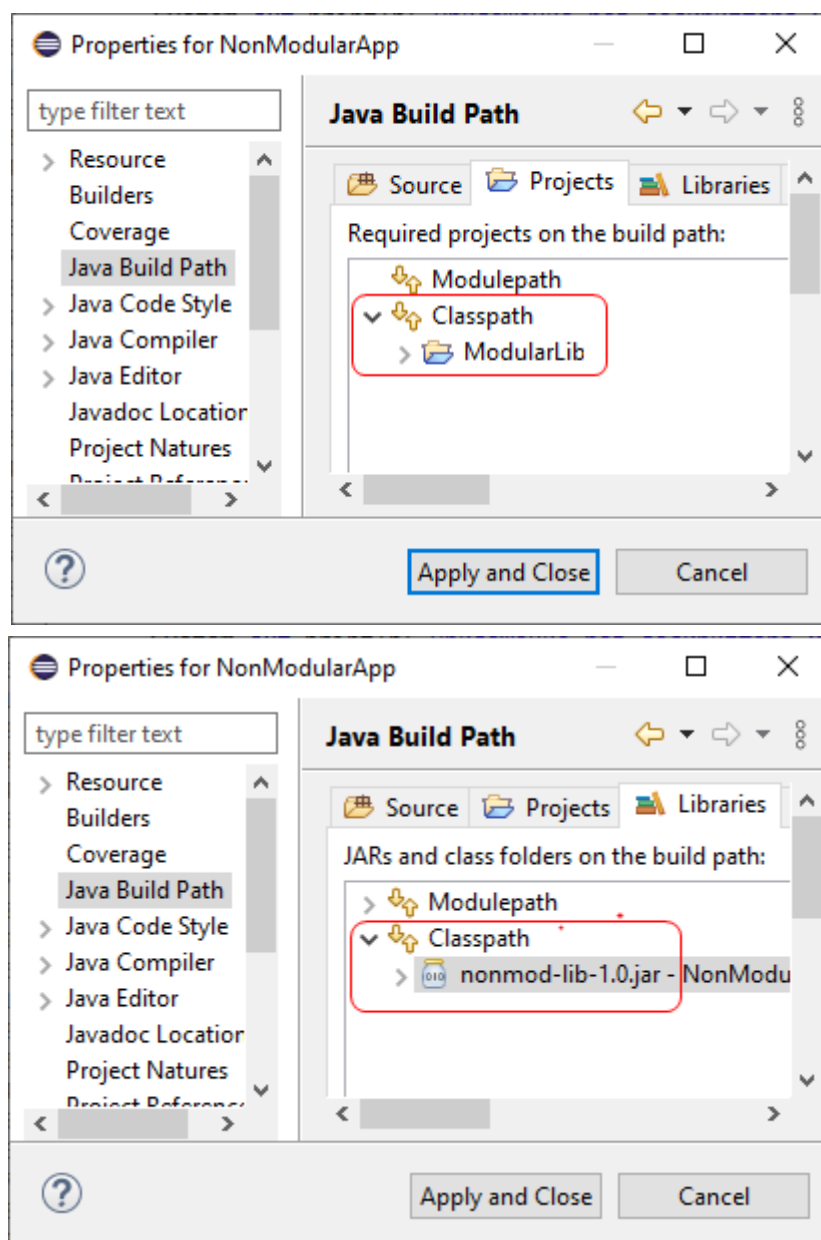


Рис 11. Размещение программного кода на пути к классам в IDE Eclipse

Обращение к классам и пакетам, загруженным в безымянный модуль из других именованных модулей невозможно по причине отсутствия у безымянного модуля имени. Роль безымянного модуля в модульных Java-приложениях состоит в обеспечении обратной совместимости с программным кодом, написанным без использования модулей на JDK до версии 9.

Вся совокупность именованных модулей (рисунок 10) может быть разделена на автоматические и явные модули.

Автоматические модули предназначены для использования кода из .JAR файлов, не содержащих модулей, в модульных приложениях. Имя

автоматического модуля формируется автоматически на основании имени .JAR файла по следующим правилам:

1. удаляется расширение .jar;
2. если в конце строки есть дефис, за которым следует хотя бы одна цифра и, возможно, точка, то берется часть строки, до последнего дефиса;
3. все символы, кроме букв и цифр, заменяются точками;
4. несколько идущих подряд точек заменяются одной;
5. начальные и конечные точки удаляются.

Так, например, на основе файла с названием *nonmod-lib-1.0.jar* будет автоматически сформировано имя модуля *nonmod.lib*. Содержимое .JAR файла будет размещено в автоматическом модуле, если данный файл не содержит явно объявленных модулей и размещен на пути к модулям (при использовании IDE Eclipse .JAR файл подключен в свойствах проекта «*Java Build Path*→*Libraries*» через список «*Modulepath*»).

Явный модуль — это модуль с явно указанным именем, для которого, помимо принадлежащих ему пакетов с классами и интерфейсами создан файл *дескриптора модуля*.

Дескриптор модуля — это файл с именем *module-info.java*, содержимое которого содержит определение модуля и набор директив в соответствии со следующим шаблоном:

```
[open] module <имя модуля> {  
    <директива модуля>  
    <директива модуля>  
    ...  
}
```

Необязательный модификатор *open* — объявляет открытый модуль все пакеты которого считаются доступными во время выполнения (через Reflection API).

Имя модуля — идентификатор, корректный для языка Java, или несколько таких идентификаторов, разделенных точкой. При задании имени модуля рекомендуется придерживаться следующих правил:

1. если модуль содержит библиотеку, предоставляемую другим разработчикам, то для обеспечения уникальности имени желательно использовать имя экспортируемого пакета верхнего уровня. Например, если модуль содержит пакеты с именами:

```
by.bsu.rct.fdp.processors  
by.bsu.rct.fdp.core  
by.bsu.rct.fdp
```

то имя модуля будет: *by.bsu.rct.fdp*

2. если модуль не предназначен для других разработчиков (например, содержит завершенное приложение), то можно использовать более короткие имена.

Директивы модуля определяют связь модуля с другими модулями. Перечень директив, которые могут использоваться в дескрипторе модуля приведен в таблице 12.

Таблица 11 Директивы, используемые в дескрипторе модуля

Директива	Назначение
requires <имя модуля>	указывает зависимость от другого модуля с именем <имя модуля> открытое содержимое модуля, от которого указана зависимость будет доступно как во время компиляции, так и во время выполнения
requires transitive <имя модуля>	аналогична предыдущей директиве, но обеспечивает зависимость от модуля с именем <имя модуля> всех других модулей, которые будут зависеть от того, в котором указана данная директива. При этом в вышеуказанных модулях явное указание зависимости от модуля с именем <имя модуля> не требуется
requires static <имя модуля>	указывает зависимость от другого модуля с именем <имя модуля>, которая необходима только во время компиляции
exports <имя пакета>	указывает на то, что пакет с именем <имя пакета> предоставляется для использования в других модулях
exports <имя пакета> to <имя модуля1>, <имя модуля2>, ...	указывает на то, что пакет с именем <имя пакета> предоставляется для использования в модулях, имена которых указаны после ключевого слова to
opens <имя пакета>	позволяет открыть пакет с именем <имя пакета> для доступа другим модулям во время выполнения
opens <имя пакета> to <имя модуля1>, <имя модуля2>, ...	позволяет открыть пакет с именем <имя пакета> для доступа во время выполнения модулям, имена которых указаны после ключевого слова to
uses <имя сервиса>	задает сервис с именем <имя сервиса>, для которого текущий модуль может обнаруживать поставщиков используя класс библиотеки Java <code>java.util.ServiceLoader</code>
provides <имя сервиса> with <имя класса1>, <имя класса2>, ...	указывает, что модуль предоставляет для использования в других модулях сервис с именем <имя сервиса>, который реализован в классах, имена которых указаны после ключевого слова with

При использовании директив, приведенных в таблице 11, в дескрипторах модулей необходимо соблюдать следующие правила:

1. в одном дескрипторе не должно быть двух и более идентичных (включая имя директивы и следующие за ним названия) директив;
2. нельзя указывать после модификатора *with* один и тот же тип более одного раза
3. модули, указанные в директивах *requires* должны быть доступны;
4. каждый экспортируемый пакет должен присутствовать в модуле;
5. нельзя указывать зависимость модуля от самого себя (непосредственно или опосредованно, через другие модули);
6. нельзя указывать после модификатора *to* один и тот же модуль более одного раза;
7. директивы *opens* не могут использоваться в дескрипторах открытых модулей (имеющих модификатор *open*);
8. классы, реализующие сервис, должны определять или общедоступный статический метод *provider* без параметров, или (явно или неявно) конструктор без параметров, возвращающий тип, соответствующий или унаследованный от типа, указанного после директивы *provides*.

При разработке Java-приложения в IDE Eclipse, содержащего более одного модуля, необходимо придерживаться следующего порядка действий:

1. для каждого явного модуля создать отдельный проект, содержащий дескриптор модуля (файл *module-info.java*);
2. в каждом созданном проекте реализовать пакеты, классы и интерфейсы, которые должны быть включены в соответствующий модуль;
3. в дескрипторах модулей указать директивы для экспорта (*exports*) пакетов, которые предоставляются для использования в других модулях, а также директивы (*requires*), указывающие на сторонние модули, необходимые для работы создаваемого модуля;
4. если в классах модуля реализуется сервис (обычно некоторый интерфейс), то:
 - в дескрипторе модуля необходимо добавить директиву *provides with*, предоставляющую возможность другим модулям использовать данный сервис;
 - если интерфейс сервиса определен в другом модуле, то модуль, где данный интерфейс используется, должен указывать зависимость от модуля, где определен интерфейс сервиса, при помощи директивы *requires*;
 - при создании интерфейса сервиса желательно в нем разместить статические методы, необходимые для загрузки данного сервиса. Такие методы должны возвращать тип, соответствующий интерфейсу сервиса, а для получения ссылок на доступные реализации сервиса использовать метод *load* класса *java.util.ServiceLoader*.
5. если в классах модуля используется сервис необходимо в дескриптор модуля добавить директиву *uses*;

6. для проектов модулей, которые зависят от других модулей:
- открыть окно свойств проекта, щелкнув правой кнопкой мыши по имени проекта и нажав в контекстном меню пункт «*Properties*»;
 - выбрать в списке параметров в левой части окна элемент «*Java Build Path*»;
 - выбрать в правой части окна закладку «*Projects*»;
 - добавить (см. раздел 5.3.3) проекты модулей, от которых зависит текущий модуль, на путь к модулям (в список «*Modulepath*»);
 - закрыть окно свойств проекта, нажав кнопку «*Apply and Close*».

5 Базовое приложение для лабораторной работы

Задание: составить многомодульную программу загрузки данных из текстового файла с использованием потоков ввода/вывода и сопутствующей обработкой исключений. Путь к файлу данных передается как параметр командной строки. Загружаемые данные должны храниться в рамках приложения в виде списка строк. В процессе загрузки данных выводить информацию о загружаемых строках в консоль. Функционал, связанный с загрузкой данных, оформить в виде отдельного явного модуля.

5.1 Структура приложения

Базовое приложение в данной лабораторной работе включает:

- проект ***DataStorage***, где определен явный модуль *arist.lab2.datastorage*, содержащий класс ***TextFileDataSource***, предназначенный для организации импорта данных из текстового файла в список строк с одновременным выводом информации о ходе процесса загрузки данных в консоль;
- проект ***MainApp***, где определен явный модуль *arist.lab2.main*, содержащий главный класс приложения ***Main***, включающий метод *main*, в котором выполняется создание списка строк для хранения данных, загружаемых из файла, а так же запускается сам процесс загрузки данных с использованием объекта класса ***TextFileDataSource***, определенного в модуле *arist.lab2.datastorage*.

5.2 Создание модуля *arist.lab2.datastorage* в Eclipse

Для создания явного модуля *arist.lab2.datastorage* в IDE Eclipse необходимо создать отдельный проект, который будет содержать данный модуль, создать дескриптор модуля, а также пакеты и классы, принадлежащие модулю.

5.2.1 Создание проекта *DataStorage*

Используя пункт главного меню Eclipse «*File→New→Java Project*» необходимо отобразить окно настройки параметров создаваемого проекта. В данном окне необходимо задать имя проекта «*DataStorage*» (рисунок 12) и, оставляя другие настройки по умолчанию, нажать кнопку «*Finish*». В появившемся после этого окне создания дескриптора модуля (рисунок 13) ввести имя модуля *arist.lab2.datastorage* и нажать кнопку «*Create*», что позволит завершить создание проекта.

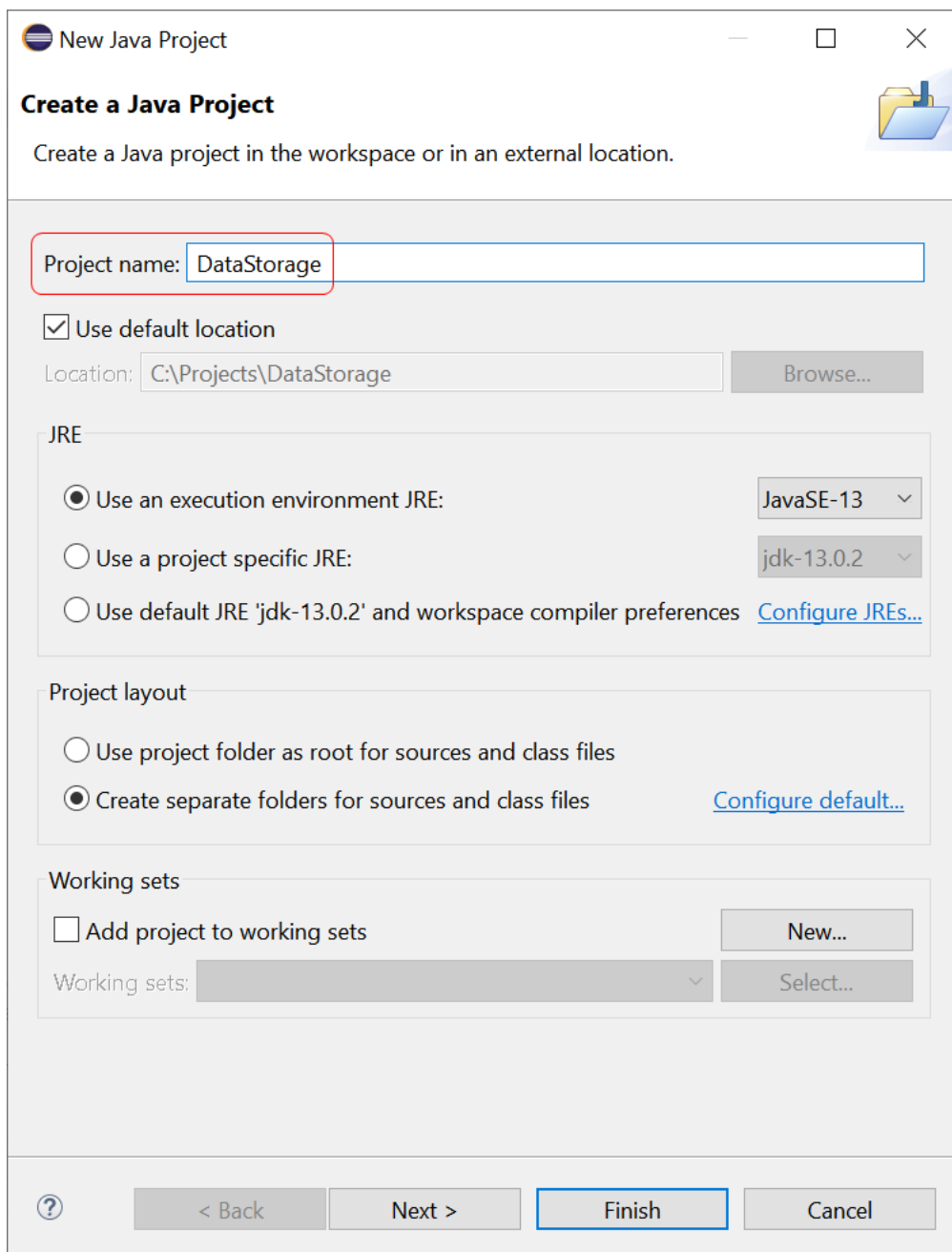


Рис 12. Окно настройки параметров проекта *DataStorage*

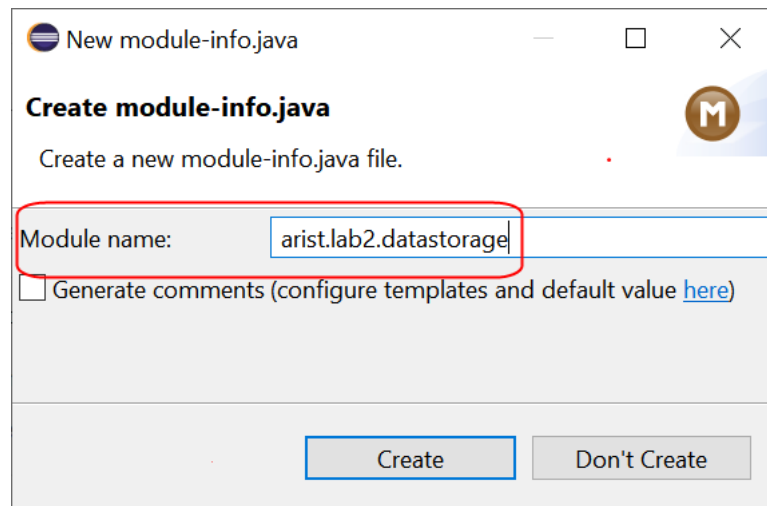


Рис 13. Окно создания дескриптора модуля *arist.lab2.datastorage*

Содержимое созданного проекта «*DataStorage*», содержащего файл «*module-info.java*» дескриптора модуля *arist.lab2.datastorage*, будет отображаться в виде «*Project explorer*» («*Project explorer*» view), пример которого представлен на рисунке 14.

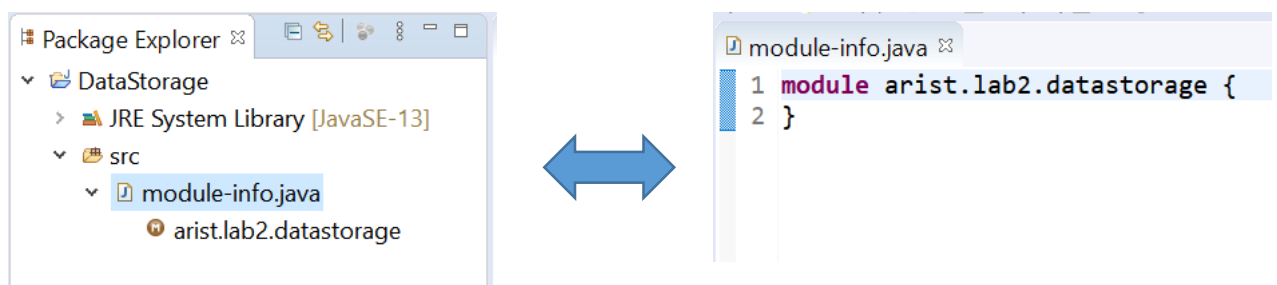


Рис 14. Представление содержимого проекта *DataStorage* в среде Eclipse, после создания проекта

5.2.2 Создание класса *TextFileDataSource* для загрузки данных

Поскольку в рамках разрабатываемого приложения модуль *arist.lab2.datastorage* отвечает за ввод/вывод данных, то в данном модуле необходимо создать класс для импорта данных из текстового файла *TextFileDataSource*. Данный класс будет принадлежать пакету *arist.lab2.datastorage.dataimport*. Для создания вышеуказанного пакета необходимо щелкнуть правой кнопкой мыши на папке «*Src*» проекта *DataStorage* в виде «*Project explorer*» и в появившемся контекстном меню выбрать пункт «*New*→*Package*». В окне создания нового пакета необходимо ввести его имя (рисунок 15), и далее нажать кнопку «*Finish*».

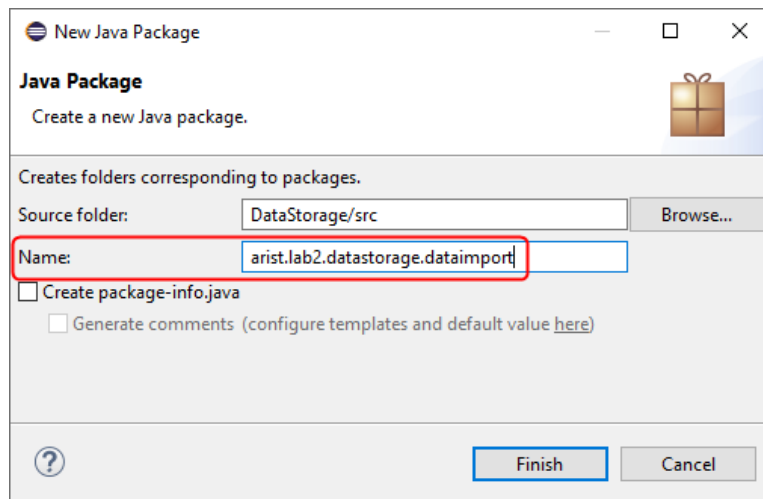


Рис 15. Окно создания нового пакета

Для создания каркаса класса *TextFileDataSource* необходимо щелкнуть правой кнопкой мыши на имени пакета *arist.lab2.datastorage.dataimport* в виде «Project explorer» и в появившемся контекстном меню выбрать пункт «New→Class». Активизация данного пункта меню позволяет отобразить окно создания нового класса (рисунок 16), в котором необходимо указать имя создаваемого класса и затем нажать кнопку «Finish».

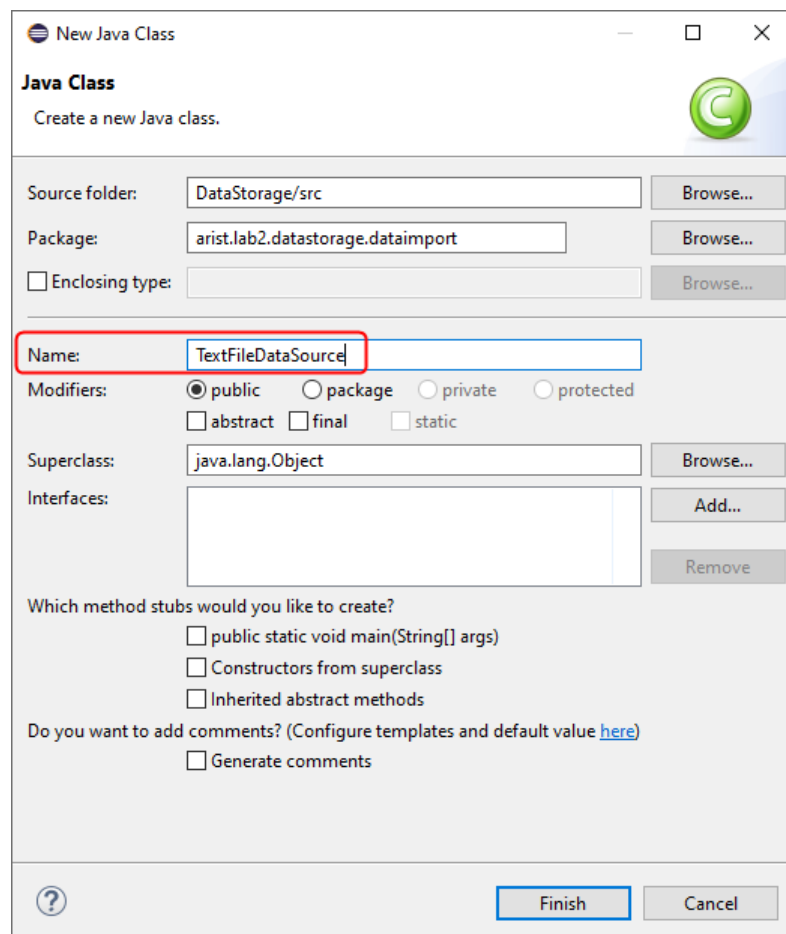


Рис 16. Окно создания класса *TextFileDataSource*

Сразу после создания исходный код класса *TextFileDataSource* будет иметь вид:

```
package arist.lab2.datastorage.dataimport;

public class TextFileDataSource {

}
```

Для организации импорта данных из текстового файла добавим в класс *TextFileDataSource* метод *LoadData*:

```
public void LoadData(String filename, List<String> data) {
    try(BufferedReader src = new BufferedReader(new FileReader(filename))){
        String line;
        while((line = src.readLine()) != null) {
            data.add(line);
            System.out.println(" " + line + " -> загружен");
        }
    } catch (FileNotFoundException e) {
        System.out.println("Файл " + filename + " не найден.");
    } catch (IOException e) {
        System.out.println("Ошибка при работе с файлом " + filename + "'.");
    }
}
```

Данный метод принимает два аргумента, первый из которых *filename* используется для передачи имени файла данных, а второй *data* содержит ссылку на список строк, в который будут добавлены импортированные данные.

Для получения данных из файла в текстовом виде в методе *LoadData* создается объект класса *FileReader*, однако возможностей данного класса недостаточно для считывания из файла отдельных строк текста. Такую возможность обеспечивает класс *BufferedReader*. Поэтому объект потока для считывания информации из файла данных *src* будет объектом класса *BufferedReader*. При создании данного объекта, для его связывания с файлом данных, в конструктор класса *BufferedReader* передается ссылка на объект класса *FileReader*.

Поскольку при создании, использовании и закрытии потока чтения данных из файла может быть сгенерирован ряд исключительных ситуаций, необходимо использовать блок обработки исключений *try/catch*. В коде метода *LoadData* используется блок *try/catch* с ресурсами, в качестве ресурса для которого используется объект потока *src*. Это позволяет обеспечить автоматическое закрытие потока после окончания работы блока *try*, независимо от генерации в нем исключительных ситуаций. Если исключительная ситуация появится при создании потока чтения данных из файла (конструктор класса *FileReader*), то она будет обработана в первой ветке *catch*, обрабатывающей исключение типа *FileNotFoundException*. Все остальные исключения, которые могут быть сгенерированы в процессе чтения данных и при закрытии потока обрабатываются во второй ветке *catch*, обрабатывающей исключение типа *IOException*.

Для получения строки данных из текстового файла используется метод *readLine* класса *BufferedReader*. Если все данные были считаны, то данный метод вернет значение *null*, что используется для окончания цикла чтения данных. Для добавления считанной из файла строки данных в список *data* используется метод *add*.

Явный конструктор в классе *TextFileDataSource* не определен, поэтому автоматически в данный класс добавляется конструктор по умолчанию без параметров.

5.2.3 Экспорт содержимого модуля *arist.lab2.datastorage* для использования в других модулях

Для обеспечения доступности программного кода, импортирующего данные из текстового файла, для других модулей необходимо дополнить файл дескриптора *module-info.java* модуля *arist.lab2.datastorage*. В данный файл необходимо добавить директиву *exports* для пакета *arist.lab2.datastorage.dataimport*, которому принадлежит класс *TextFileDataSource*. В результате файл *module-info.java* для модуля *arist.lab2.datastorage* приобретает следующий окончательный вид:

```
module arist.lab2.datastorage {  
    exports arist.lab2.datastorage.dataimport;  
}
```

5.3 Создание модуля *arist.lab2.main* в Eclipse

Для создания явного модуля *arist.lab2.main* в IDE Eclipse необходимо создать отдельный проект, который будет содержать данный модуль, создать дескриптор модуля, а также пакеты и классы, принадлежащие модулю.

5.3.1 Создание проекта *MainApp*.

Создание проекта *MainApp* выполняется в порядке, аналогичном созданию проекта *DataStorage*, который был описан в разделе 3.2.1. Создание проекта включает следующие шаги:

1. с использованием пункта главного меню Eclipse «*File→New→Java Project*» необходимо отобразить окно настройки параметров создаваемого проекта, где указать имя проекта «*MainApp*» и, оставляя другие настройки по умолчанию, нажать кнопку «*Finish*»;
2. в появившемся после выполнения шага 1 окне создания дескриптора модуля необходимо ввести имя модуля *arist.lab2.main* и нажать кнопку «*Create*», что позволит завершить создание проекта.

Содержимое созданного проекта «*MainApp*», содержащего файл «*module-info.java*» дескриптора модуля *arist.lab2.main*, будет отображаться в виде «*Project explorer*» («*Project explorer*» view) (рисунок 17).

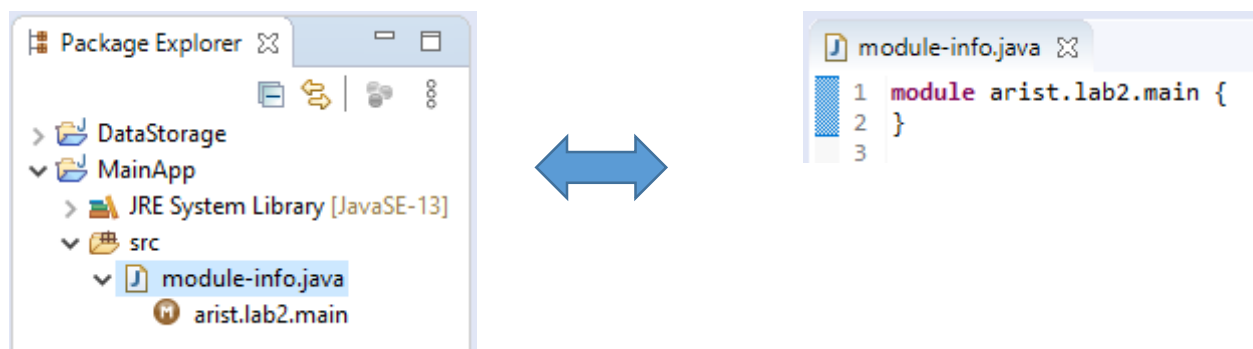


Рис 17. Представление содержимого проекта *MainApp* в среде Eclipse, после создания проекта

5.3.2 Создание класса *Main* для запуска и тестирования многомодульного приложения

Модуль *arist.lab2.main* в рамках многомодульного приложения содержит код, позволяющий запустить приложение на выполнение и задействовать при этом другие модули. Для организации точки входа в приложение необходимо реализовать метод *main*. Данный метод будет принадлежать классу *Main*, расположенному в пакете *arist.lab2.main*. Процесс создания вышеуказанных пакета и класса включает следующие шаги:

1. для создания пакета *arist.lab2.main* необходимо в виде «*Project explorer*» щелкнуть правой кнопкой мыши на папке «*Src*» проекта *MainApp* и в появившемся контекстном меню выбрать пункт «*New→Package*». В окне создания нового пакета необходимо ввести его имя, и далее нажать кнопку «*Finish*»;
2. для создания каркаса класса *Main* необходимо щелкнуть правой кнопкой мыши на имени пакета, созданного на шаге 1 и в появившемся контекстном меню выбрать пункт «*New→Class*». Далее в окне создания нового класса необходимо указать имя *Main* и нажать кнопку «*Finish*». После создания класса *Main* необходимо добавить в него метод *main* вида:


```

public static void main(String[] args) {
    //Проверяем передан ли параметр командной строки
    if(args.length == 0) {
        System.out.println("Задайте путь к файлу данных как параметр командной строки!");
        System.exit(0);
    }

    //Создаем список строк для хранения данных из файла
    ArrayList<String> data = new ArrayList<String>();

    //Загружаем данные
    TextFileDataSource dataSource = new TextFileDataSource();

    System.out.println("Файл данных: '" + Path.of(args[0]) + "'");
    System.out.println("Чтение исходных данных:");
    dataSource.LoadData(args[0], data);
}

```

В приведенном выше коде метода *main* предполагается, что имя файла данных для импорта в приложение будет передано как параметр командной строки и, соответственно, будет содержаться в массиве строк *args*, который является входным аргументом для данного метода. Поэтому для проверки наличия переданного имени файла используется код вида:

```

if(args.length == 0) {
    System.out.println("Задайте путь к файлу данных как параметр командной строки!");
    System.exit(0);
}

```

Если передаваемых через параметр командной строки данных нет, то будет выведено сообщение о необходимости передать имя файла и далее выполняется завершение работы приложения.

Далее в коде метода *main* выполняется создание списка строк, в который должны будут быть добавлены данные из импортируемого файла. Ссылка на данный список хранится в переменной *data*.

Последняя часть кода в приведенном выше методе *main* выполняет запуск импорта данных из файла, для чего создается объект *datasource* класса *TextFileDataSource*, для которого далее вызывается метод *LoadData*.

5.3.3 Подключение модуля *arist.lab2.datastorage* к модулю *arist.lab2.main*

Для обеспечения доступа из модуля *arist.lab2.main* к классу *TextFileDataSource*, код которого расположен в модуле *arist.lab2.datastorage*, необходимо выполнить подключение модуля *arist.lab2.datastorage* к модулю *arist.lab2.main*.

Для этого необходимо в файл дескриптора *module-info.java* модуля *arist.lab2.main* добавить директиву *requires* с указанием после нее требуемого модуля. В результате файл *module-info.java* для модуля *arist.lab2.main* приобретает следующий окончательный вид:

```

module arist.lab2.main {
    requires arist.lab2.datastorage;
}

```

Приведенный выше вид дескриптора модуля *arist.lab2.main* только лишь указывает на необходимость использования модуля *arist.lab2.datastorage* в модуле *arist.lab2.main*. Однако дополнительно необходимо указать, где именно располагается программный код модуля *arist.lab2.datastorage*. Для этого проект *DataStorage*, в рамках которого создан модуль *arist.lab2.datastorage*, необходимо добавить на путь к модулям для проекта *MainApp*. С этой целью необходимо в виде «*Project explorer*» щелкнуть правой кнопкой мыши на проекте *MainApp*, после чего в контекстном меню выбрать пункт «*Properties*». Данный пункт меню позволяет открыть окно конфигурации проекта, где в списке, расположенном в левой части данного окна, необходимо выбрать элемент «*Java Build Path*» (рисунок 18). Справа на странице настроек, соответствующих элементу «*Java Build Path*», необходимо выбрать закладку «*Projects*», далее в окне «*Required projects on the build path:*» выделить пункт «*Modulepath*» и нажать справа кнопку «*Add*».

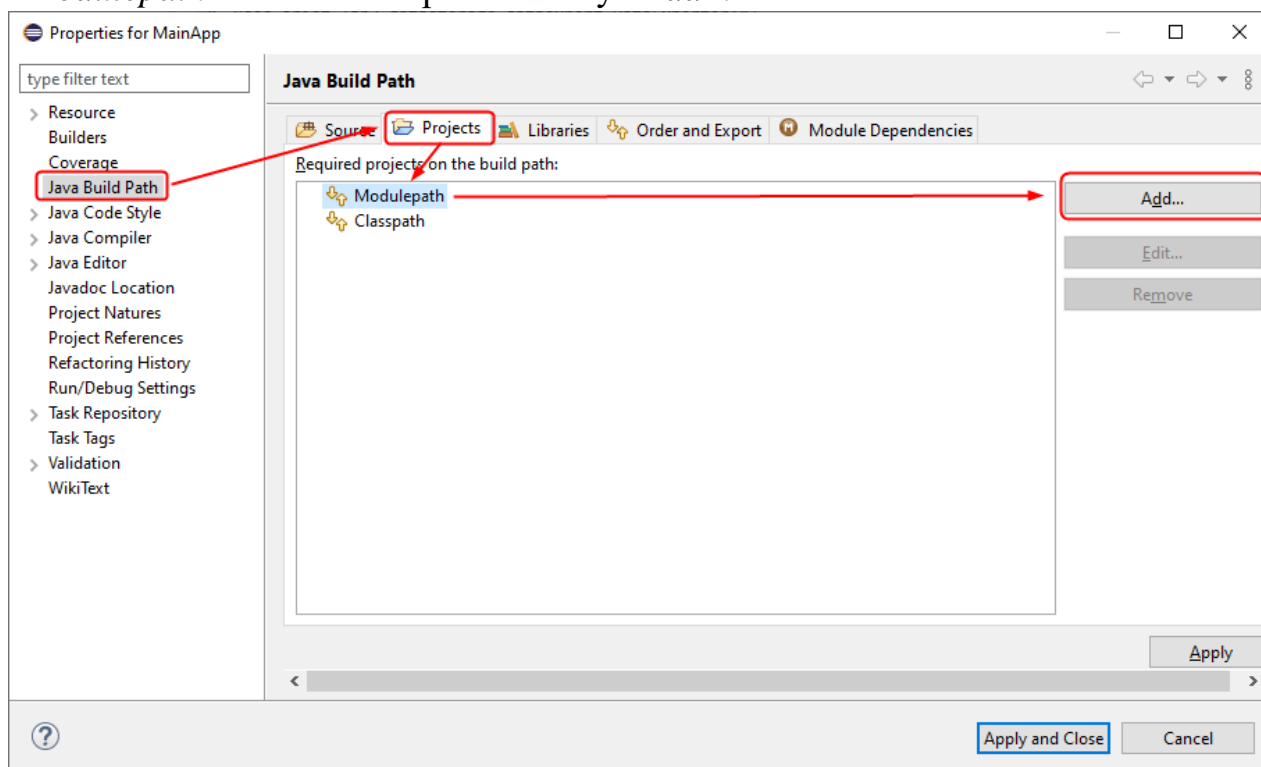


Рис 18. Порядок действий в окне свойств проекта *MainApp* для вызова окна добавления других проектов на путь к модулям

Нажатие кнопки «*Add*» в окне, представленном на рисунке 18, приведет к открытию окна выбора требуемых проектов (рисунок 19). В данном окне необходимо выбрать проект *DataStorage* и нажать кнопку «*Ok*».

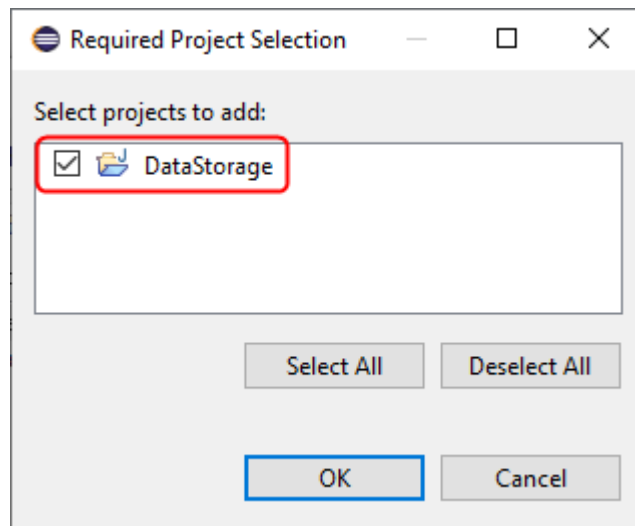


Рис 19. Диалоговое окно выбора требуемых проектов

После закрытия окна выбора требуемых проектов, проект *DataStorage* появится в списке проектов, добавленных на путь к модулям для проекта *MainApp* (рисунок 20)

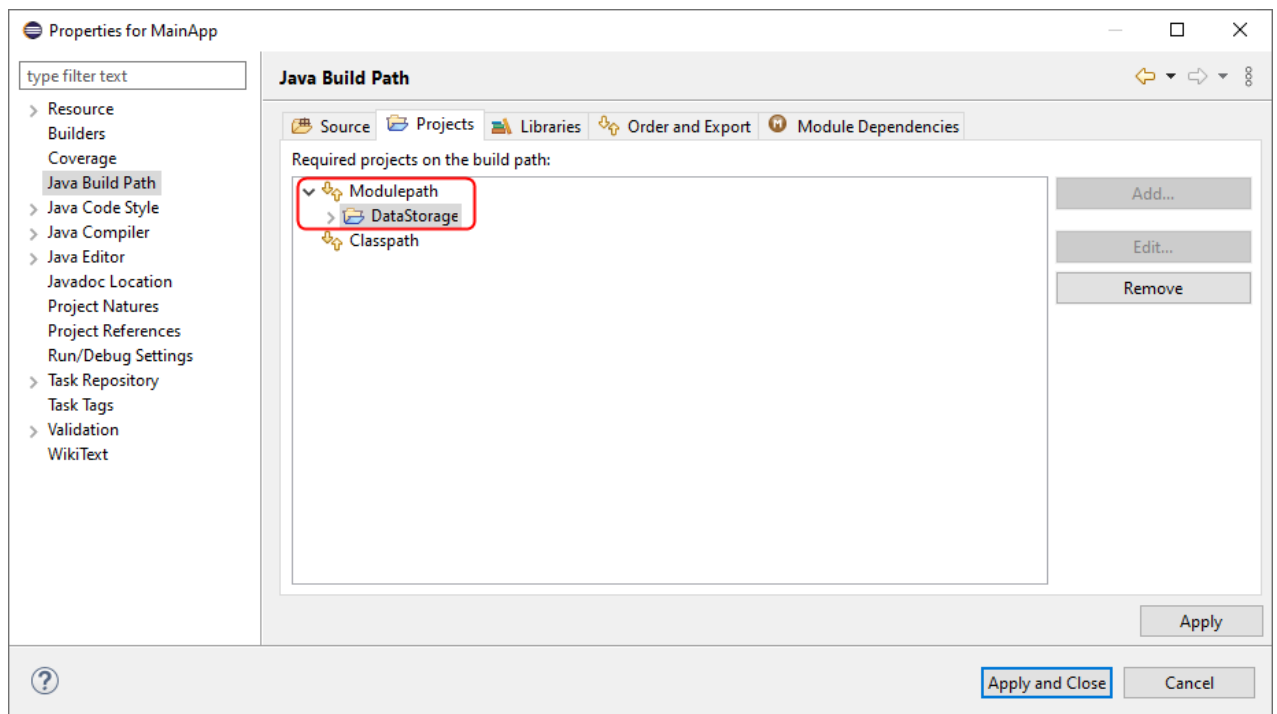



Рис 20. Результат добавления проекта *DataStorage* на путь к модулям для проекта *MainApp*

Для завершения работы с окном конфигурации проекта *MainApp* необходимо нажать кнопку «*Apply and Close*».

5.4 Запуск приложения

Полные версии исходных кодов классов базового приложения данной лабораторной работы (без реализации заданий, предложенных в пункте 4) размещены в Приложении 1.

Для задания импортируемого файла данных как аргумента командной строки необходимо определить конфигурацию запуска приложения. Для этого следует активировать пункт меню «*Run → Run Configuration*», в открывшемся диалоговом окне (рисунок 21) в списке слева выбрать пункт «*Java Application*» и нажать кнопку «*New launch configuration*»  (первая слева кнопка на панели инструментов, расположенной в левой части окна над списком).

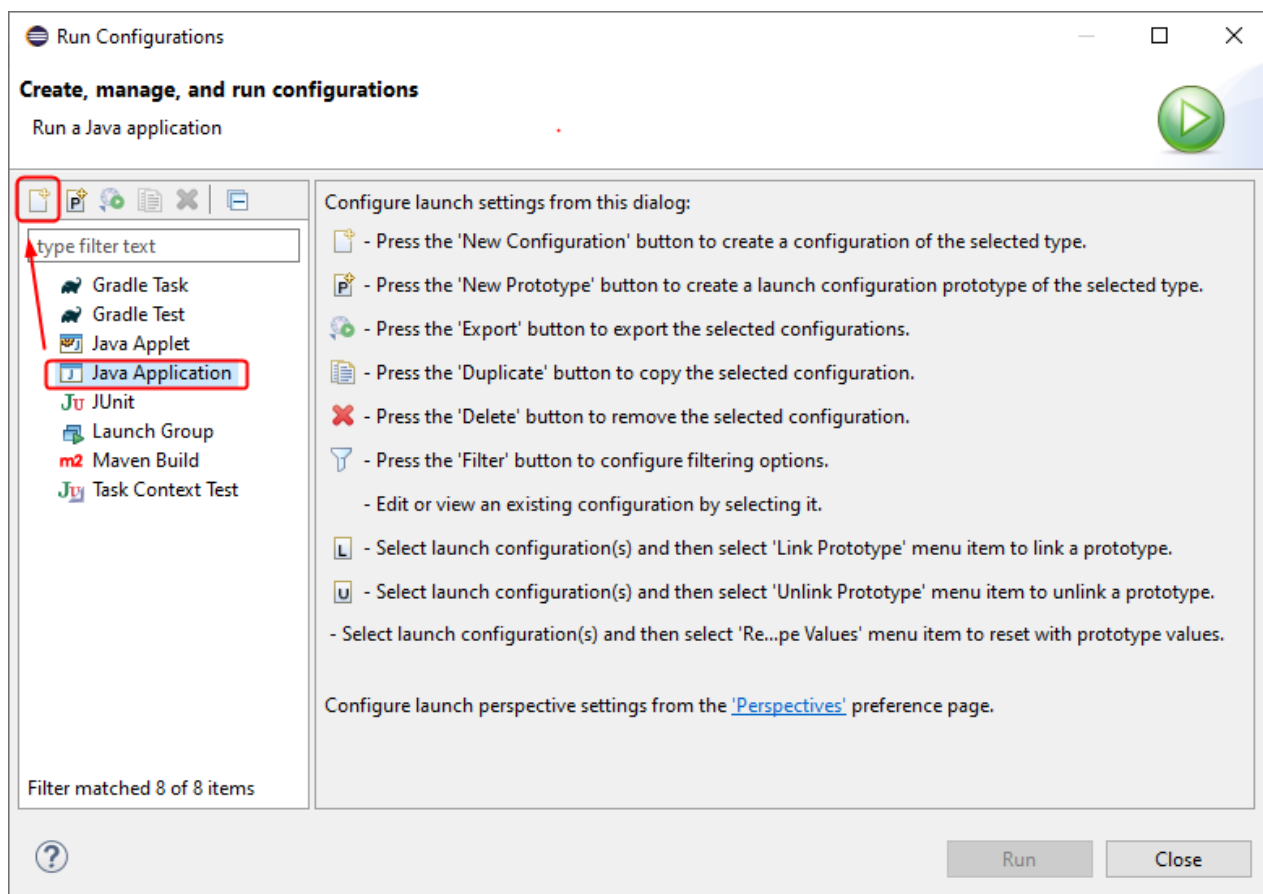


Рис 21. Создание новой конфигурации запуска приложения

После создания новой конфигурации запуска приложения перейти на вкладку «*Arguments*», и в поле ввода «*Program arguments*» указать путь и имя файла данных (рисунок 22). После этого следует нажать кнопки «*Apply*» и «*Run*».

Замечание: аргумент командной строки рекомендуется помещать в двойные кавычки, в противном случае, при наличии пробелов, он будет интерпретирован как несколько различных аргументов. Указанный путь к файлу данных должен соответствовать его местоположению в файловой системе.

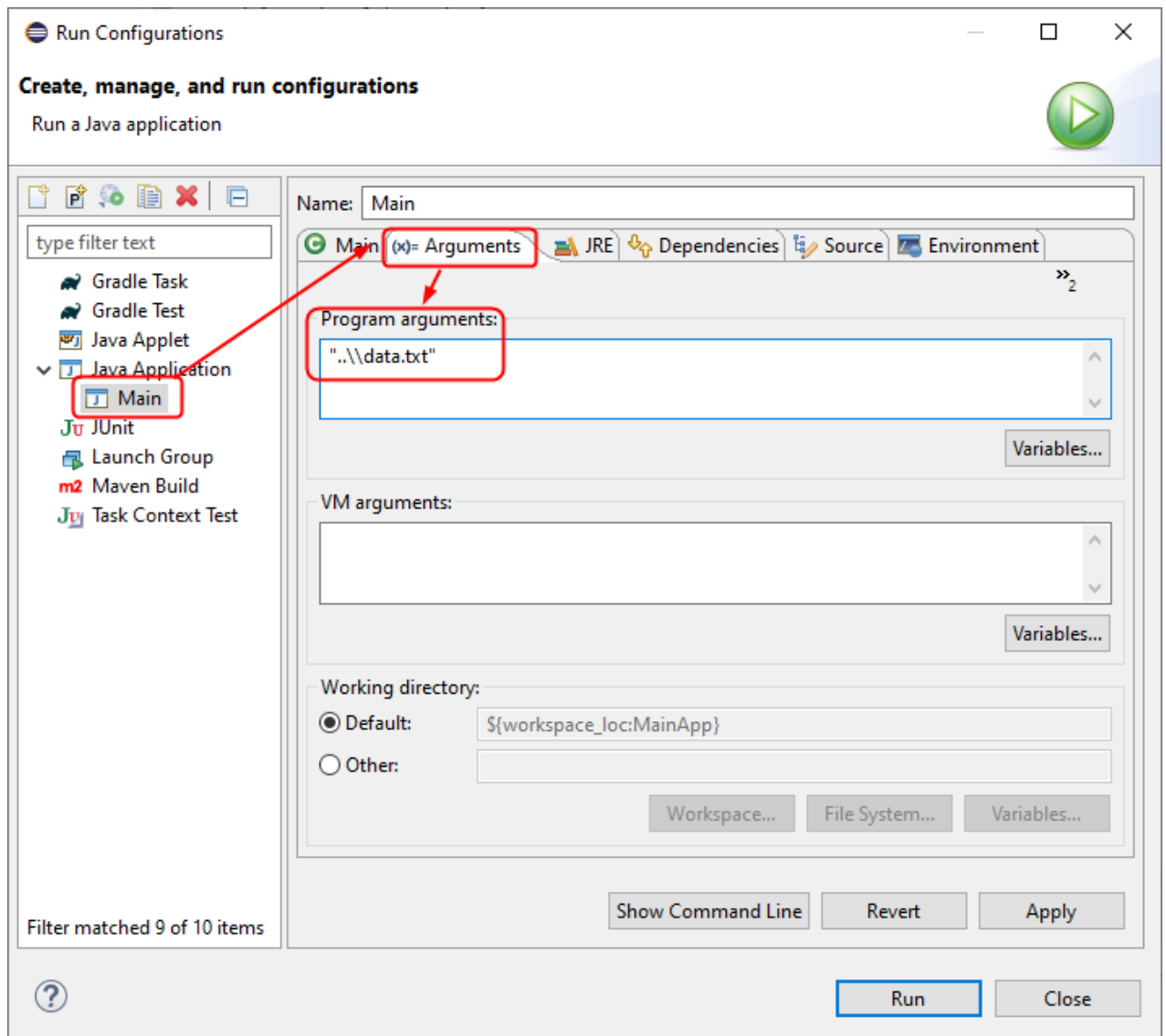


Рис 22. Задание пути и имени файла данных как аргумента командной строки в диалоговом окне настройки конфигурации запуска

Результаты работы приложения будут показаны в виде «Console».

6 Задания

6.1 Вариант сложности А

- а) Выполнить загрузку данных из файла (по вариантам) с использованием стандартного программного кода, приведенного в Приложении 1. Проконтролировать правильность загрузки по результатам, выведенным в консоль.
- б) Реализовать в пакете *arist.lab2.datastorage.dataimport* класс *DataChecker*, где создать метод *CheckDataItem*, проверяющий корректность (по вариантам) отдельной строки из файла с данными. Если строка не корректна, генерировать исключение типа *Exception* с сообщением, соответствующим данной строке.
- в) Задействовать класс *DataChecker* для проверки строк данных, при их загрузке из файла. Выводить в консоль информацию о корректности или некорректности загружаемых строк. Некорректные данные не добавлять в список загруженных данных.
- г) Реализовать дополнительный модуль *arist.lab2.processors* куда добавить одноименный пакет, в котором создать класс *DataStatistics*. В данном классе создать метод вида:

```
public Collection<String> collect (ArrayList<String> data) {  
    ...  
}
```

В методе *collect* создать новую коллекцию с информацией (по вариантам), на основе данных, загруженных из файла (передаются через аргумент *data*).

- д) Использовать класс *DataStatistics* в классе *Main*, модуля *arist.lab2.main*, для сбора статистики с последующим ее выводом в консоль.

Таблица 12 Описание данных и заданий для реализации по вариантам (для уровня сложности А)

№ п/п	Файл данных	Критерий корректности строк в файле данных	Описание собираемой информации для задания г)
1	data1.txt	строка данных состоит из марки автомобиля и его цвета, разделенных символом ':'	Коллекция строк по шаблону «марка автомобиля»: «количество автомобилей», отсортированных в алфавитном порядке, марки автомобилей не повторяются
2	data2.txt	строка данных состоит из названия продукта, после которого сразу следует его вес в круглых скобках	Коллекция строк по шаблону «название продукта»: «общий вес всех партий этого продукта», отсортированных в алфавитном порядке, названия продуктов не повторяются

№ п/п	Файл данных	Критерий корректности строк в файле данных	Описание собираемой информации для задания г)
3	data3.txt	строка данных состоит из названия пиломатериала, после которого следует его метраж, отделенный разделителем вида “->”	Коллекция строк по шаблону «название пиломатериала»: «общий метраж всех его партий», отсортированных в алфавитном порядке, названия пиломатериалов не повторяются
4	data4.txt	строка данных состоит из названия предмета мебели и его материала, разделенных символом ‘;’	Коллекция строк по шаблону «предмет мебели»: «количество предметов мебели», отсортированных в алфавитном порядке, названия предметов мебели не повторяются
5	data5.txt	строка данных состоит из названия устройства и его производителя, разделенных символом ‘\’	Коллекция строк по шаблону «устройство»: «количество устройств», отсортированных в алфавитном порядке, названия устройств не повторяются
6	data6.txt	строка данных состоит из типа продукта, после которого следует его объем, отделенный символом ‘>’	Коллекция строк по шаблону «тип продукта»: «общий объем всех таких продуктов», отсортированных в алфавитном порядке, типы продуктов не повторяются
7	data7.txt	строка данных состоит из элемента одежды, после которого следует его размер, отделенный символом ‘-‘	Коллекция строк по шаблону «вид одежды»: «количество единиц одежды такого вида», отсортированных в алфавитном порядке, виды одежды не повторяются
8	data8.txt	строка данных состоит из марки грузовика, после которого следует грузоподъемность, отделенная символом ‘=‘	Коллекция строк по шаблону «марка грузовика»: «количество грузовиков такой марки», отсортированных в алфавитном порядке, марки грузовиков не повторяются
9	data9.txt	строка данных состоит из вида обуви, после которого следует размер, отделенный символом ‘~‘	Коллекция строк по шаблону «вид обуви»: «количество пар обуви такого вида», отсортированных в алфавитном порядке, тип обуви не повторяется

№ п/п	Файл данных	Критерий корректности строк в файле данных	Описание собираемой информации для задания г)
10	data10.txt	строка данных состоит из вида общественного транспорта, после которого следует число мест для сидения, отделенное символом '\$'	Коллекция строк по шаблону « <i>вид транспорта</i> »: « <i>суммарное число мест для сидения</i> », отсортированных в алфавитном порядке, тип транспорта не повторяется
11	data11.txt	строка данных состоит из вида канцелярского товара, после которого следует его формат, отделенный символом '&'	Коллекция строк по шаблону « <i>вид товара</i> »: « <i>количество товаров такого вида</i> », отсортированных в алфавитном порядке, вид товара не повторяется
12	data12.txt	строка данных состоит из названия цветка, после которого следует его длина, отделенный символом '#'	Коллекция строк по шаблону « <i>название цветка</i> »: « <i>количество цветов с таким названием</i> », отсортированных в алфавитном порядке, название цветка не повторяется

6.2 Вариант сложности В

- а) Выполнить задание а) варианта сложности А;
- б) Реализовать дополнительный модуль *arist.lab2.datacontainers*, куда добавить одноименный пакет, в котором создать:
 1. класс (по вариантам) для хранения строк данных, загруженных из файла, который должен иметь отдельный конструктор, принимающий строку из файла, извлекающий из нее данные для заполнения полей и, если данные не корректны, генерирующий исключение с использованием класса, описанного ниже;
 2. класс исключения (по вариантам), который должен использоваться для генерации исключений в конструкторе класса для хранения данных при попытке сохранения некорректных данных в объект.

Модуль *arist.lab2.datacontainers* подключить к модулю *arist.lab2.datastorage* таким образом, чтобы он автоматически мог использоваться во всех модулях, требующих *arist.lab2.datastorage*.
- в) Изменить загрузку данных из файла, создавая список объектов класса, описанного в задании б)-1, вместо списка строк. Выводить в консоль информацию о корректности или некорректности загружаемых строк. Некорректные данные не добавлять в список загруженных данных.
- г) Реализовать дополнительный модуль *arist.lab2.processors*, куда добавить одноименный пакет, в котором создать класс *DataStatistics*. В данном

классе создать два метода (по вариантам) для сбора статистики по данным, загруженным из файла.

- д) Использовать оба метода класса *DataStatistics* в классе *Main*, модуля *arist.lab2.main*, для сбора статистики.
- е) Создать в модуле *arist.lab2.datastorage* пакет *arist.lab2.datastorage.dataexport*, в котором реализовать класс *TextFileDataExporter* с методами, позволяющими сохранять собранную при выполнении пункта д) статистику в текстовые файлы, и использовать эти методы в классе *Main* модуля *arist.lab2.main*.

Таблица 13 Описание заданий для реализации по вариантам (для уровня сложности В)

№ п/п	Классы модуля <i>arist.lab2.datacontainers</i>	Методы класса <i>DataStatistics</i>
1	<p>Класс для хранения данных: <i>CarDataContainer</i></p> <p>Поля: <i>String model</i> – хранит марку автомобиля; <i>String color</i> – хранит цвет автомобиля</p> <p>Класс исключения: <i>IncorrectCarDataException</i></p>	<p>1. <i>amountByModel</i> входной параметр: <i>ArrayList<CarDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>carsByColor</i> входной параметр: <i>ArrayList<CarDataContainer> data</i> возвращаемое значение: коллекция цветов автомобиля, имеющих в файле данных, каждому из которых поставлен в соответствие список имеющихся марок автомобилей такого цвета и их количество, для каждой марки.</p>
2	<p>Класс для хранения данных: <i>ProductDataContainer</i></p> <p>Поля: <i>String name</i> – хранит название продукта; <i>int amount</i> – хранит вес партии продукта</p> <p>Класс исключения: <i>IncorrectProductDataException</i></p>	<p>1. <i>amountByProduct</i> входной параметр: <i>ArrayList<ProductDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>productsByAmountRange</i> входной параметр: <i>ArrayList<ProductDataContainer> data</i> возвращаемое значение: коллекция диапазонов веса продукта с шагом 5, начиная с 0 (например, 0-4, 5-9 и т.д.), сформированных на основании веса продуктов, имеющих в файле данных, каждому из которых поставлен в соответствие список имеющихся продуктов с весом, попадающим в данный диапазон.</p>

№ п/п	Классы модуля <i>arist.lab2.datacontainers</i>	Методы класса <i>DataStatistics</i>
3	<p>Класс для хранения данных: <i>MaterialDataContainer</i></p> <p>Поля: <i>String name</i> – хранит название пиломатериала; <i>int footage</i> – хранит метраж пиломатериала</p> <p>Класс исключения: <i>IncorrectMaterialDataException</i></p>	<p>1. <i>footageByMaterial</i> входной параметр: <i>ArrayList<MaterialDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>materialsByFootageRange</i> входной параметр: <i>ArrayList<MaterialDataContainer> data</i> возвращаемое значение: коллекция диапазонов метража с шагом 10, начиная с 0 (например, 0-9, 10-19 и т.д.), сформированных на основании метража пиломатериалов, имеющих в файле данных, каждому из которых поставлен в соответствие список имеющихся пиломатериалов с метражом, попадающим в данный диапазон.</p>
4	<p>Класс для хранения данных: <i>FurnitureDataContainer</i></p> <p>Поля: <i>String name</i> – хранит название предмета мебели; <i>String material</i> – хранит материал, из которого изготовлен предмет мебели</p> <p>Класс исключения: <i>IncorrectFurnitureDataException</i></p>	<p>1. <i>amountByName</i> входной параметр: <i>ArrayList<FurnitureDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>furnitureByMaterial</i> входной параметр: <i>ArrayList<FurnitureDataContainer> data</i> возвращаемое значение: коллекция названий материалов из файла данных, каждому из которых соответствует список предметов мебели, изготовленных из такого материала, для каждого из которых указано их количество.</p>
5	<p>Класс для хранения данных: <i>DeviceDataContainer</i></p> <p>Поля: <i>String device</i> – хранит название устройства; <i>String manufacturer</i> – хранит имя производителя</p> <p>Класс исключения: <i>IncorrectDeviceDataException</i></p>	<p>1. <i>amountByDevice</i> входной параметр: <i>ArrayList<DeviceDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>deviceByManufacturer</i> входной параметр: <i>ArrayList<DeviceDataContainer> data</i> возвращаемое значение: коллекция имен производителей из файла данных, каждому из которых поставлен в соответствие список имеющихся устройств данного производителя, для каждого из которых указано их количество.</p>

№ п/п	Классы модуля <i>arist.lab2.datacontainers</i>	Методы класса <i>DataStatistics</i>
6	Класс для хранения данных: <i>FoodDataContainer</i> Поля: <i>String type</i> – хранит тип продукта; <i>double volume</i> – хранит объем продукта Класс исключения: <i>IncorrectFoodDataException</i>	1. <i>amountByFoodType</i> <u>входной параметр:</u> <i>ArrayList<FoodDataContainer> data</i> <u>возвращаемое значение:</u> смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>foodByVolume</i> <u>входной параметр:</u> <i>ArrayList<FoodDataContainer> data</i> <u>возвращаемое значение:</u> коллекция диапазонов объема с шагом 0.25, начиная с 0 (например, “от 0 до 0.25”, “от 0.25 до 0.5” и т.д.), сформированных по объемам продуктов из файла данных, каждому из которых соответствует список продуктов с объемом из данного диапазона.
7	Класс для хранения данных: <i>ClothesDataContainer</i> Поля: <i>String type</i> – хранит тип одежды; <i>int size</i> – хранит размер одежды Класс исключения: <i>IncorrectClothesDataException</i>	1. <i>amountByClothesType</i> <u>входной параметр:</u> <i>ArrayList< ClothesDataContainer> data</i> <u>возвращаемое значение:</u> смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>foodByVolume</i> <u>входной параметр:</u> <i>ArrayList<ClothesDataContainer> data</i> <u>возвращаемое значение:</u> коллекция размеров одежды, сформированных на основании размеров из файла данных, каждому из которых соответствует список имеющихся элементов одежды такого размера, с указанием их количества.
8	Класс для хранения данных: <i>TruckDataContainer</i> Поля: <i>String brand</i> – хранит марку грузовика; <i>int carrying</i> – хранит грузоподъемность Класс исключения: <i>IncorrectTruckDataException</i>	1. <i>amountByBrand</i> <u>входной параметр:</u> <i>ArrayList< TruckDataContainer > data</i> <u>возвращаемое значение:</u> смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>foodByVolume</i> <u>входной параметр:</u> <i>ArrayList< TruckDataContainer > data</i> <u>возвращаемое значение:</u> коллекция диапазонов грузоподъемности с шагом 10, начиная с 0 (например, 0-9, 10-19 и т.д.), сформированных на базе грузоподъемности из файла данных, которым поставлен в соответствие список грузовиков с грузоподъемностью из данного диапазон.

№ п/п	Классы модуля <i>arist.lab2.datacontainers</i>	Методы класса <i>DataStatistics</i>
9	Класс для хранения данных: <i>FootwearDataContainer</i> Поля: <i>String kind</i> – хранит вид обуви; <i>int size</i> – хранит размер Класс исключения: <i>IncorrectFootwearDataException</i>	1. <i>amountByKind</i> входной параметр: <i>ArrayList< FootwearDataContainer > data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>footwearBySize</i> входной параметр: <i>ArrayList< FootwearDataContainer > data</i> возвращаемое значение: коллекция размеров обуви, сформированных на основании имеющихся в файле данных, каждому из которых соответствует список имеющихся видов обуви такого размера, с указанием их количества.
10	Класс для хранения данных: <i>TransportDataContainer</i> Поля: <i>String type</i> – хранит вид транспорта; <i>int seatsCount</i> – хранит количество мест для сидения Класс исключения: <i>IncorrectTransportDataException</i>	1. <i>amountByTransportType</i> входной параметр: <i>ArrayList< TransportDataContainer > data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>transportBySeatsCount</i> входной параметр: <i>ArrayList< TransportDataContainer > data</i> возвращаемое значение: коллекция диапазонов числа мест для сидения с шагом 10, начиная с 20 (например, 20-29, 30-39 и т.д.), сформированных на базе данных из файла, которым соответствует список транспортных средств с числом мест для сидения, попадающим в данный диапазон.
11	Класс для хранения данных: <i>StationeryDataContainer</i> Поля: <i>String kind</i> – хранит вид канцелярского товара; <i>String format</i> – хранит размер канцелярского товара Класс исключения: <i>IncorrectStationeryDataException</i>	1. <i>amountOfStationery</i> входной параметр: <i>ArrayList<StationeryDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А. 2. <i>stationeryByFormat</i> входной параметр: <i>ArrayList<StationeryDataContainer > data</i> возвращаемое значение: коллекция форматов канцелярских товаров, сформированных на основании имеющихся в файле данных, каждому из которых поставлен в соответствие список имеющихся видов канцелярских товаров такого формата, с указанием их количества.

№ п/п	Классы модуля <i>arist.lab2.datacontainers</i>	Методы класса <i>DataStatistics</i>
12	<p>Класс для хранения данных: <i>FlowerDataContainer</i></p> <p>Поля: <i>String name</i> – хранит название цветка; <i>int length</i> – хранит длину цветка</p> <p>Класс исключения: <i>IncorrectFlowerDataException</i></p>	<p>1. <i>amountOfFlowers</i> входной параметр: <i>ArrayList< FlowerDataContainer> data</i> возвращаемое значение: смотри описание возвращаемого значения метода <i>collect</i> в задании г) варианта сложности А.</p> <p>2. <i>flowersByLength</i> входной параметр: <i>ArrayList< FlowerDataContainer > data</i> возвращаемое значение: коллекция диапазонов длин цветков с шагом 20, начиная с 10 (например, 10-29, 30-49 и т.д.), сформированных на основании данных из файла, каждому из которых поставлен в соответствие список имеющихся цветов с длиной, попадающей в данный диапазон.</p>

6.3 Вариант сложности С

- а) Выполнить задание а) варианта сложности В;
- б) Выполнить задание б) варианта сложности В;
- в) Выполнить задание в) варианта сложности В;
- г) Выполнить задание г) варианта сложности В;
- д) Выполнить задание г) варианта сложности В;
- е) В модуле *arist.lab2.datastorage* создать пакет *arist.lab2.datastorage.dataexport*, в котором определить интерфейс сервиса *DataExporter*. В интерфейсе *DataExporter* определить абстрактные методы для идентификации сервиса по имени и экспорта данных, собранных при выполнении пункта д), а также статический метод для обеспечения поиска сервисов, реализующих данный интерфейс.
- ж) В модуле *arist.lab2.datastorage* создать отдельный пакет в который добавить два отдельных класса *FileExportService* и *ConsoleExportService*, реализующих сервисы на основе интерфейса *DataExporter*. Класс *FileExportService* должен обеспечивать экспорт данных в текстовый файл, а *ConsoleExportService* в консоль.
- з) Использовать сервисы, созданные при выполнении пункта ж) в классе *Main* модуля *arist.lab2.main* для экспорта данных, собранных при выполнении пункта д).

Приложение 1. Исходный код базового приложения

Проект *DataStorage*

Класс для загрузки данных из текстового файла *TextFileDataSource*

```
package arist.lab2.datastorage.dataimport;

import java.io.*;
import java.util.List;

public class TextFileDataSource {

    public void LoadData(String filename, List<String> data) {
        try(BufferedReader src = new BufferedReader(new FileReader(filename))){
            String line;
            while((line = src.readLine()) != null) {
                data.add(line);
                System.out.println(" '" + line + "' -> загружен");
            }
        } catch (FileNotFoundException e) {
            System.out.println("Файл '" + filename + "' не найден.");
        } catch (IOException e) { //для исключений метода readline и при закрытии ресурса
            System.out.println("Ошибка при работе с файлом '" + filename + "'");
        }
    }
}
```

Дескриптор модуля *arist.lab2.datastorage*

```
module arist.lab2.datastorage {

    exports arist.lab2.datastorage.dataimport;

}
```

Проект *MainApp*

Главный класс приложения *Main*

```
package arist.lab2.main;

import java.nio.file.Path;
import java.util.*;
import arist.lab2.datastorage.dataimport.*;

public class Main {

    public static void main(String[] args) {
        //Проверяем передан ли параметр командной строки
        if(args.length == 0) {
            System.out.println("Задайте путь к файлу данных как параметр командной строки!");
        }
    }
}
```



```

        System.exit(0);
    }

    //Создаем список строк для хранения данных из файла
    ArrayList<String> data = new ArrayList<String>();

    //Загружаем данные
    TextFileDataSource dataSource = new TextFileDataSource();

    System.out.println("Файл данных: '" + Path.of(args[0]) + "'");
    System.out.println("Чтение исходных данных:");
    dataSource.LoadData(args[0], data);
}
}

```

Дескриптор модуля *arist.lab2.main*

```

module arist.lab2.main {

    requires arist.lab2.datastorage;

}

```