

Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

Учебное пособие

Тема 2. Инструментальные средства языка Java

Томск
2020

Резник, Виталий Григорьевич

Распределенные вычислительные сети. Учебное пособие. Тема 2. Инструментальные средства языка Java / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2019. – 68 с.

В учебном пособии рассмотрены основные современные технологии организации распределенных вычислительных сетей, которые уже получили достаточно широкое распространение и подкреплены соответствующими инструментальными средствами реализации распределенных приложений. Представлены основные подходы к распределенной обработке информации. Проводится обзор организации распределенных вычислительных систем: методы удалённых вызовов процедур, многослойные клиент-серверные системы, технологии гетерогенных структур и одноранговых вычислений. Приводится описание концепции GRID-вычислений и сервис-ориентированный подход к построению распределенных вычислительных систем. Рассматриваемые технологии подкрепляются описанием инструментальных средств разработки программного обеспечения, реализованных на платформе языка Java.

Пособие предназначено для студентов бакалавриата по направлению 09.03.01 «Информатика и вычислительная техника» при изучении курсов «Вычислительные системы и сети» и «Распределенные вычислительные системы».

Одобрено на заседании каф. АСУ протокол №_____ от _____

УДК 004.75

© Резник В. Г., 2019

© Томск. гос. ун-т систем упр. и радиоэлектроники, 2019

Оглавление

Тема 2. Инструментальные средства языка Java.....	4
2.1 Общее описание инструментальных средств языка.....	6
2.1.1 Инструментальные средства командной строки.....	7
2.1.2 Пакетная организация языка Java.....	9
2.1.3 Инструментальные средства Eclipse.....	13
2.2 Классы и простые типы данных.....	18
2.2.1 Операторы и простые типы данных.....	19
2.2.2 Синтаксис определения классов.....	20
2.2.3 Синтаксис и семантика методов.....	22
2.2.4 Синтаксис определения интерфейсов.....	23
2.2.5 Объекты и переменные.....	24
2.3 Управляющие операторы языка.....	26
2.4 Потоки ввода-вывода.....	28
2.4.1 Стандартный ввод-вывод.....	29
2.4.2 Классы потоков ввода.....	31
2.4.3 Классы потоков вывода.....	34
2.5 Управление сетевыми соединениями.....	37
2.5.1 Адресация на базе класса InetAddress.....	37
2.5.2 Адресация на базе URL и URLConnection.....	39
2.5.3 Сокеты протокола TCP.....	41
2.5.4 Сокеты протокола UDP.....	42
2.5.5 Простейшая задача технологии клиент-сервер.....	43
2.6 Организация доступа к базам данных.....	51
2.6.1 Инструментальные средства СУБД Apache Derby.....	52
2.6.2 SQL-запросы и драйверы баз данных.....	55
2.6.3 Типовой пример выборки данных.....	58
Вопросы для самопроверки.....	68

Тема 2. Инструментальные средства языка Java

Настоящая тема посвящена краткому изучению базовых средств языка Java, который был разработан американской компанией Sun Microsystems [28]. Считается, что он начал проектироваться в начале 90-х годов, а датой официального выпуска считается 23 мая 1995 года [27]. В настоящее время, правопреемницей технологических достижений языка является корпорация Oracle [29].

Поскольку технологическая основа Java предполагает реализацию виртуальной машины JVM [26], то широкое распространение языка было под большим сомнением. Действительно, в начале 90-х годов большинство персональных компьютеров использовало 20-битную адресацию команд, что ограничивало объем основной памяти (ОП) ЭВМ размером 1 МБ, из которого только 640 КБ предназначалось для запуска программ. Тем не менее, при вероятной поддержке корпорации Oracle, выпустившей в начале 80-х годов свою знаменитую СУБД с одноимённым названием, Sun Microsystems продолжила свою работу и получила широкое признание своих достижений. Позже корпорация Oracle даже включила JVM в свою СУБД для реализации триггеров и функций ядра ориентированных на численные расчёты.

Сам язык Java, как и язык C++, унаследовал синтаксис языка C, поэтому студенты, изучавшие курс ООП, могут легко начинать программировать даже на основе статьи Википедии [27]. Мы же будем опираться на руководство [8], но предварительно сделаем ряд общих замечаний.

В декабре 1998 года вышла версия языка под номером 1.2, в которую была добавлена структура коллекций (collections framework) и ряд других незначительных изменений. Этот факт стал основой различных публикаций, использующих обозначение языка как Java 2. Такое обозначение можно найти и в современных книгах, учебных пособиях и используемых сокращениях, хотя они относятся к одному и тому же языку Java. Кроме того, часто мажорный номер версии опускается, а указывается только минорный. Поэтому словосочетание «Java 7» подразумевает версионное обозначение языка: «Java 1.7».

Большая популярность и широкое распространение технологии Java на различные платформы потребовало разделения первоначально единого программного обеспечения. В настоящее время выделяются следующие платформы:

- **J2EE** или Java EE (начиная с v.1.5) — Java Enterprise Edition, для создания программного обеспечения уровня предприятия;
- **J2SE** или Java SE (начиная с v.1.5) — Java Standard Edition, для создания пользовательских приложений, например, — для настольных систем;
- **J2ME** или Java ME или Java Micro Edition, — для использования в устройствах, ограниченных по вычислительной мощности, например, мобильных, КПК или встроенных системах;

- **Java Card** — для использования в устройствах без собственного человеко-машинного интерфейса, например, — в смарт-картах.

Дополнительно, стандартные дистрибутивы Java подразделяются на два основных вида поставок, за которыми сейчас нужно обращаться на сайты корпорации Oracle:

- **JRE** (*Java Runtime Environment*) — среда исполнения Java-приложений, обязательным компонентом которой является *JVM*;
- **JDK** (*Java Development Kit*) — дистрибутив инструментальных средств для разработки приложений на языке Java, обязательно включающий в себя базовый дистрибутив *JRE*;
- другие, альтернативные виды дистрибутивов, за которыми следует обращаться к другим поставщикам; например, **OpenJDK** — дистрибутив, обычно формируемый и поставляемый провайдерами ОС Linux.

Особо следует отметить, что дистрибутивы поставляются с фиксированным номером версии и аппаратной платформы. Например, используемый в данном учебном пособии дистрибутив: *OpenJDK_1.8* для 64-битной платформы ОС Arch Linux. Чтобы не перегружать деталями вводную часть этой главы, в подразделе 2.1 предоставляется общая вводная часть, которая касается работы с языком в командной строке интерпретатора shell и в инструментальной среде системы Eclipse EE. Более конкретная и детальная информация предоставляется в методических руководствах по лабораторным работам, например, [6, 7] и соответствующих указаниях по практическим занятиям, если таковые предусмотрены, например, [8].

Остальные подразделы данной главы посвящены краткому описанию синтаксиса и семантики базовых конструкций языка Java и завершаются рассмотрением базовых сетевых приложений, а именно:

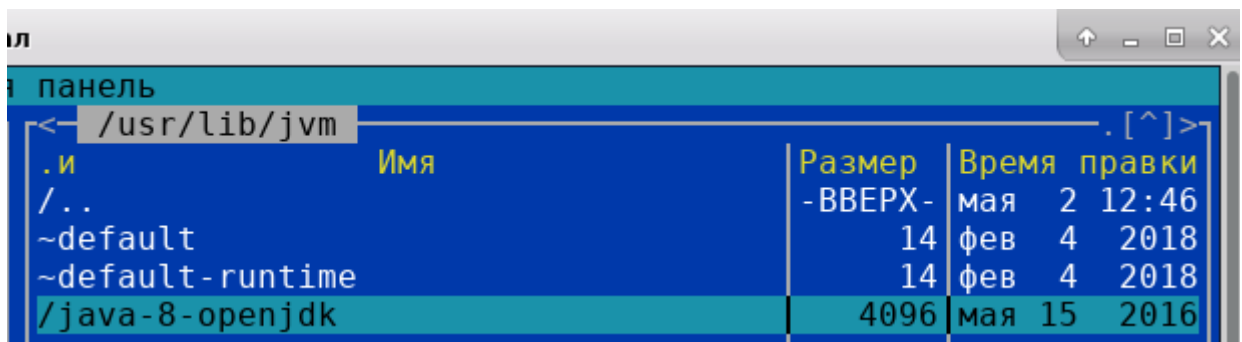
- подразделы 2.2-2.4 — содержат описания простейших типов, классов, методов и организацию ввода-вывода; это обеспечивает студентов знаниями и навыками написания обычных расчётных программ;
- подраздел 2.5 — готовит студентов к написанию простейших клиент-серверных программ на базе транспортного уровня стека протоколов TCP/IP;
- подраздел 2.6 — обеспечивает студентов навыками работы с базами данных на языке Java.

В целом, учебный материал данной главы, хоть и имеет выраженную прикладную направленность, но ориентирован на подготовку студента к освоению теоретического и практического материала последующих глав дисциплины.

2.1 Общее описание инструментальных средств языка

Поскольку технологии Java обладают высокой межплатформенной переносимостью, то среды исполнения (JRE) и инструментальные средства (JDK) работают в них практически одинаково. Тем не менее, в различных ОС обычно устанавливаются разные дистрибутивы. Различным является и их размещение, хотя программное обеспечение располагается компактно и не вызывает затруднений в его использовании.

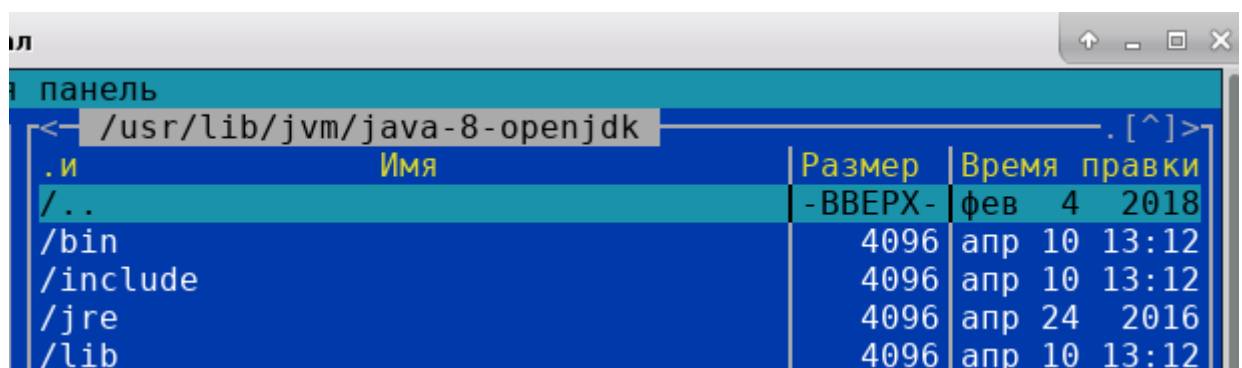
В учебном пособии используется дистрибутив OpenJDK версии 1.8, который является частью учебного программного комплекса [6] и с которым студенты хорошо знакомы в процессе изучения дисциплины «*Операционные системы*». Стандартное размещение OpenJDK находится в корне директории `/usr/lib/jvm`, как это показано на рисунке 2.1.



.и	Имя	Размер	Время	правки
/..		-ВВЕРХ-	мая 2	12:46
~default		14	фев 4	2018
~default-runtime		14	фев 4	2018
/java-8-openjdk		4096	мая 15	2016

Рисунок 2.1 — Базовая директория установки дистрибутивов Java

Теоретически, здесь может быть установлено несколько разных дистрибутивов, поэтому для удобства настроек используются две ссылки ***default*** и ***default-runtime***, конкретизирующие дистрибутив по умолчанию. На рисунке 2.1 они указывают на одну и ту же директорию ***java-8-openjdk***, содержимое которой представлено на рисунке 2.2.



.и	Имя	Размер	Время	правки
/..		-ВВЕРХ-	фев 4	2018
/bin		4096	апр 10	13:12
/include		4096	апр 10	13:12
/jre		4096	апр 24	2016
/lib		4096	апр 10	13:12

Рисунок 2.2 — Типовое содержимое директории OpenJDK, где каталог `jre` является корнем дистрибутива среды исполнения JRE

Что касается ОС MS Windows, то установка Java производится специальным инсталлятором, который скачивается с сайта корпорации Oracle. По умолчанию, инсталляция производится в базовый каталог **C:\Program Files (x86)\Java**, а структура каталогов дистрибутивов JDK оказывается такой же, как и на рисунке 2.2.

С целью избежания возможных скрытых недоразумений, студент должен учитывать, что все примеры программ, приведённые в данном пособии, реализованы в среде ОС Linux и в рабочей области пользователя *vgr*. Это сделано для того, чтобы студент приложил некоторые минимальные усилия для самостоятельной реализации всех учебных проектов.

2.1.1 Инструментальные средства командной строки

Инструментальные средства языка Java содержат некоторое множество различных программ (утилит), которыми необходимо владеть для успешного использования всех технологических возможностей языка. Две такие утилиты являются наиболее важными и часто используемыми:

- **java** — главная программа, запускающая виртуальную машину Java;
- **javac** — компилятор исходных текстов языка, входящий в дистрибутив JDK.

Хотя современные инсталляторы ОС обычно обеспечивают все необходимые настройки для конечного пользователя, разработчику приходится самому исправлять имеющиеся ошибки, проектировать и отлаживать новые дистрибутивы. К счастью ПО Java ориентировано на достаточно автономную базовую установку, что упрощает его последующую эксплуатацию. Действительно, как показано на рисунке 2.2, все дистрибутивы JDK имеют общую архитектуру, сосредоточенную в следующих директориях:

- **bin** — каталог для хранения исполняемых файлов JDK (для ОС MS Windows исполняемые файлы имеют стандартное расширение *.exe*);
- **lib** — для хранения библиотек (пакетов) Java, имеющих нативное (родное) расширение *.jar*, а также платформозависимые расширения *.so* или *.dll*;
- **jre** — каталог размещения дистрибутива JRE, также имеющий подкаталоги *bin* и *lib* с аналогичным назначением.

Дополнительные настройки инструментальной среды Java определяются с помощью четырёх переменных среды ОС:

- **PATH** — список директорий, где ОС будет искать исполняемые файлы для запуска;
- **CLASSPATH** — список директорий, где Java будет искать нужные биб-

- библиотеки (пакеты библиотек);
- **JRE_HOME** — указывает на директорию инсталляции JRE;
- **JAVA_HOME** — указывает на директорию инсталляции JDK.

Чтобы проверить правильную настройку инструментальных средств Java, следует выполнить команды:

```
$ java -version
$ javac -version
```

где знак «\$» означает, что команда запущена от имени пользователя *vgr*, а знак «#» используется, если команды запускаются от имени пользователя *root*.

Для демонстрации работы *java* и *javac* воспользуемся исходным текстом простейшей программы, представленной на листинге 2.1.

Листинг 2.1 — Исходный текст класса Example1

```
public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир!");
    }

}
```

Здесь приведено описание публичного класса *Example1*, содержащего единственный публичный статический метод *main(...)*, который печатает на стандартный вывод (объект *System.out*) строку «Здравствуй, Мир!» с помощью метода *println(...)*. Полное определение классов и методов будет дано в следующем подразделе, а сейчас отметим, что в языке Java принято называть классы именами, начинающимися с большой буквы, а методы — с маленькой. Кроме того, исходный текст каждого класса хранится в отдельном файле с именем класса и стандартным расширением *.java*.

Учитывая эти требования, создадим рабочую директорию */home/vgr/src/rvs*, перейдём в неё и сохраним содержимое листинга 2.1 в файле *Example1.java*. Затем, проведём компиляцию данного файла командой:

```
$ javac Example1.java
```

В результате, получим файл *Example1.class*, содержащий байт-код нашей программы. Запустим этот класс на выполнение командой:

\$ java Example1

Программа выведет на терминал строку: «Здравствуй, Мир» и закончит свою работу. Чтобы продемонстрировать более сложные случаи запуска программ Java, необходимо рассмотреть его пакетную организацию.

2.1.2 Пакетная организация языка Java

Все языки программирования характеризуются некоторой логической структурой, которая проецируется на его базовые возможности и последующие расширения на прикладные области применения. Обычно это представляется в виде набора взаимосвязанных библиотек, которые необходимы не только для реализации любого прикладного проекта, но и становятся необходимой понятийной базой любого программиста, характеризующей его профессиональную подготовку.

Важной особенностью языка Java является хорошая структуризация его системных программных средств. Эта структуризация основана на базе множества пакетов, группирующих классы языка в соответствии с их основной прикладной направленностью и строгим именованием, что демонстрируется ниже:

<i>java.lang</i>	Базовый пакет, обеспечивающий основные возможности языка: объекты, классы, исключения, математические функции, интерфейс с JVM и другие.
<i>java.util</i>	Инструментальный пакет классов: коллекции, дата, время, ...
<i>java.io</i>	Операции с файлами, потоковый ввод\вывод, ...
<i>java.math</i>	Набор функций: sin, cos и другие.
<i>java.net</i>	Операции с сетью, сокет, URL, ...
<i>java.security</i>	Генерация ключей, шифрование и дешифрование, ...
<i>java.sql</i>	<i>Java Database Connectivity (JDBC)</i> доступ к базам данных
<i>java.awt</i>	Базовый пакет для работы с графикой ...
<i>javax.swing</i>	Графические компоненты для разработки приложений: кнопки, текстовые поля, ...

Практически все из перечисленных пакетов, кроме двух последних, будут использованы в учебном материале данной главы. В частности, базовый пакет ***java.lang*** уже неявно был использован в примере листинга 2.1. Это является особенностью современных реализаций языка Java. В первых версиях такое не допускалось и требовалось явное подключение всех классов, исполь-

зованных в программе.

Явное подключение используемых классов осуществляется с помощью оператора ***import***, после которого необходимо правильно указать полный путь к ним. Поскольку это в большинстве случаев является затруднительным, то возможно подключение всех классов пакета, в формате:

```
import имя_пакета.*;
```

Покажем явное подключение пакетов на примере использования объекта класса ***Date***, содержащегося в пакете ***java.util*** и задающего текущие дату и время. Для этого, модифицирует класс ***Example1***, как показано на листинге 2.2.

Листинг 2.2 — Первая модификация текста класса Example1

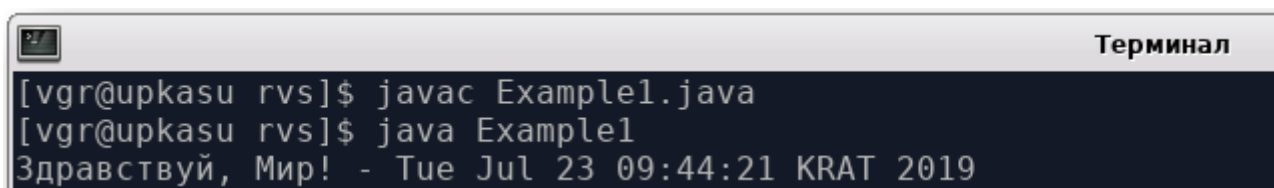
```
import java.util.*; // Подключение всех классов пакета

public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир! - " + new Date());
    }

}
```

Проведя компиляцию и запуск программы листинга 2.2, получим результат показанный на рисунке 2.3.

A screenshot of a terminal window titled "Терминал". The terminal shows the following commands and output:

```
[vgr@upkasu rvs]$ javac Example1.java
[vgr@upkasu rvs]$ java Example1
Здравствуй, Мир! - Tue Jul 23 09:44:21 KRAT 2019
```

Рисунок 2.3 — Результат компиляции и запуска программы листинга 2.2

Приведённый пример демонстрирует важные особенности программирования на языке Java, связанные с его пакетной организацией классов:

- объект ***out*** обеспечивает стандартный вывод, находится в базовом пакете ***java.lang*** и определён статическим типом класса, поэтому разрешается прямое обращение к методу ***java.lang.System.out.println(...)***, а используемое сокращение допускается наличием однозначности его вызова;

- класс **Date()** является динамическим типом, поэтому объект даты генерируется с помощью оператора **new**, а однозначность генерации обеспечивается отсутствием других пакетов, содержащих класс **Date()**;
- строка любого текста, заключённого в двойные кавычки, является объектом, имеющим свои методы и оператор конкатенации «+»;
- метод **println(...)** демонстрирует также приведение объекта типа **Date()** к объекту строкового типа.

Пакетная организация языка Java имеет простую семантику, привязанную к каталогам файловой системы ОС, что напрямую отображается синтаксисом следующего формата:

```
package имя_каталога1.имя_каталога2. ... .имя_каталогаN;
```

Строка с ключевым словом **package** помещается в начале каждого файла с исходным текстом описания класса, задавая относительную адресацию байт-кода класса в пределах реализуемого проекта. Обычно, используемые имена каталогов имеют дополнительную семантику, организующую некоторую доменную структуру и характеризующие страну, организацию, язык, автора и другие свойства целевого пакета. Например, приведённые выше имена базовых пакетов Java, начинаются именем **java**, а второе имя характеризует его целевую направленность. Для наглядности демонстрации влияния оператора **package**, проведём вторую модификацию класса *Example1*, которая представлена на листинге 2.3.

Листинг 2.3 — Вторая модификация текста класса Example1

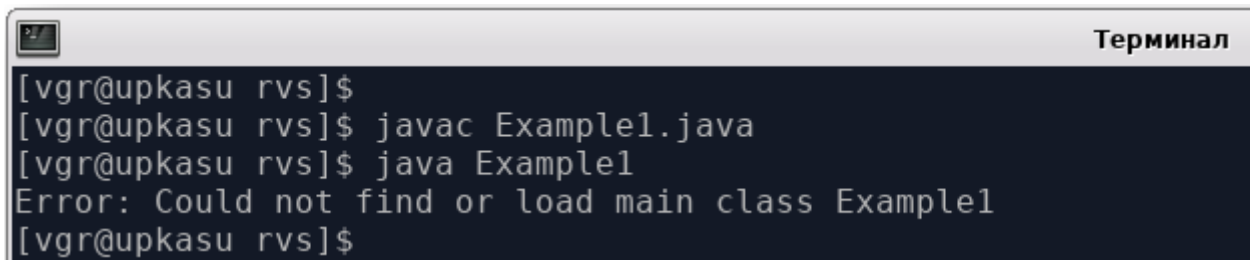
```
package ru.tusur.asu; // Описание пути для класса Example1.class

import java.util.*; // Подключение всех классов пакета

public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир! - " + new Date());
    }
}
```

Компиляция этой программы проходит успешно и в текущем каталоге появляется файл *Example1.class*, но как показано на рисунке 2.4, запустить на выполнение его не удаётся. Появляется сообщение, что класс не найден и не может быть загружен.



```
[vgr@upkasu rvs]$  
[vgr@upkasu rvs]$ javac Example1.java  
[vgr@upkasu rvs]$ java Example1  
Error: Could not find or load main class Example1  
[vgr@upkasu rvs]$
```

Рисунок 2.4 — Результат компиляции и запуска программы листинга 2.3

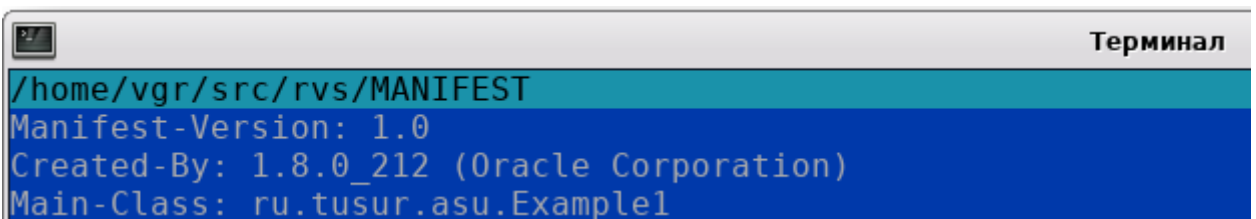
Чтобы исправить ситуацию, нужно:

- в текущем каталоге создать дерево директорий **ru/tusur/asu** и перенести туда файл *Example1.class*;
- вернувшись в исходную директорию, запустить программу командой:

```
$ java ru.tusur.asu.Example1
```

Таким образом, каждый класс, разрабатываемый в рамках какого-либо проекта, помещается в свой каталог, определяемый оператором **package**. Совокупность всех классов проекта размещается в некотором дереве каталогов, имеющем общую вершину. В последующем, такой проект оформляется в виде отдельного **jar**-файла — библиотеки языка Java, которая создаётся с помощью утилиты **jar**.

Сама утилита **jar** имеет множество параметров и показывает их запуском без параметров. Мы используем ее только для нашего примера, предварительно создав файл манифеста, представленного на рисунке 2.5.



```
/home/vgr/src/rvs/MANIFEST  
Manifest-Version: 1.0  
Created-By: 1.8.0_212 (Oracle Corporation)  
Main-Class: ru.tusur.asu.Example1
```

Рисунок 2.5 — Минимальное содержимое файла манифеста для jar-архива

Такой файл манифеста указывает статический класс архива, содержащий статический метод **main(...)**, используемый для запуска программы.

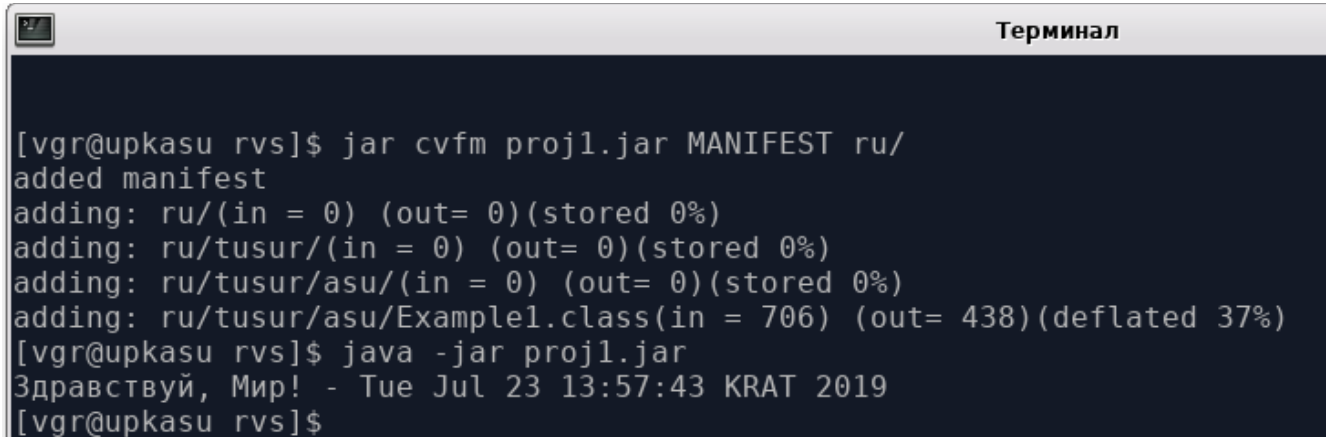
Теперь создадим архив с именем **proj1.jar** командой:

```
jar cvfm proj1.jar MANIFEST ru/
```

В текущей директории появится архив с именем **proj1.jar** и наш пример может запускаться командой:

```
java -jar proj1.jar
```

Общий результат создания архива и запуска приложения показан на рисунке 2.6.



```
Терминал
[vgr@upkasu rvs]$ jar cvfm proj1.jar MANIFEST ru/
added manifest
adding: ru/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/asu/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/asu/Example1.class(in = 706) (out= 438)(deflated 37%)
[vgr@upkasu rvs]$ java -jar proj1.jar
Здравствуй, Мир! - Tue Jul 23 13:57:43 KRAT 2019
[vgr@upkasu rvs]$
```

Рисунок 2.6 — Результат создания архива proj1.jar и запуска приложения

2.1.3 Инструментальные средства Eclipse

Возможности командной строки - достаточно широки, но в плане разработки программного обеспечения значительно уступают интегрированным инструментальным средствам IDE (*Integrated Development Environment*). В данном пособии будет использоваться IDE Eclipse EE, которое ориентировано на разработку приложений уровня предприятия средствами технологий языка Java. Мы будем придерживаться варианта ПО установленного в учебном программном комплексе [6], поэтому отметим, что упомянутый инструмент установлен опционально в среду ОС Linux и монтируется в директорию */opt/eclipseEE*. Запуск инструмента осуществляется специальным значком, размещенным на рабочем столе пользователя, или эквивалентной командой:

```
/opt/eclipseEE/eclipse -data rvs
```

Поскольку студенты хорошо знакомы с инструментальной средой Eclipse в процессе изучения курса «*Операционные системы*», программируя на языках C/C++, поэтому уделим внимание только тем аспектам технологии IDE, которые связаны с реализацией проектов на языке Java. Для конкретизации примера, создадим проект с именем *proj1*, реализующим программу с исходным текстом листинга 2.3.

В главном меню запущенной IDE, выберем: **File→New→Java Project**. Появится окно рисунка 2.7, в котором укажем имя проекта *proj1* и создадим

проект, нажав кнопку «*Finish*».

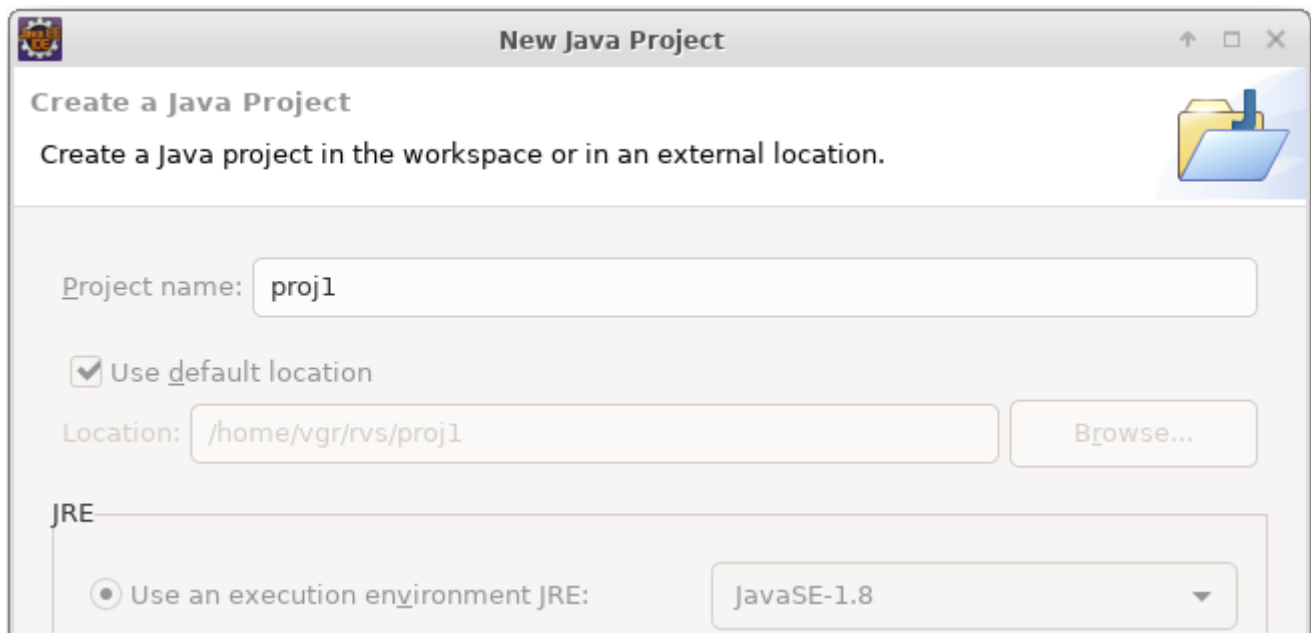


Рисунок 2.7 — Выбор названия проекта

В окне «*Package Explorer*» выделим мышкой *proj1/src* и правой кнопкой активируем меню *New*→*Class*, а появившееся окно заполним, как показано на рисунке 2.8.

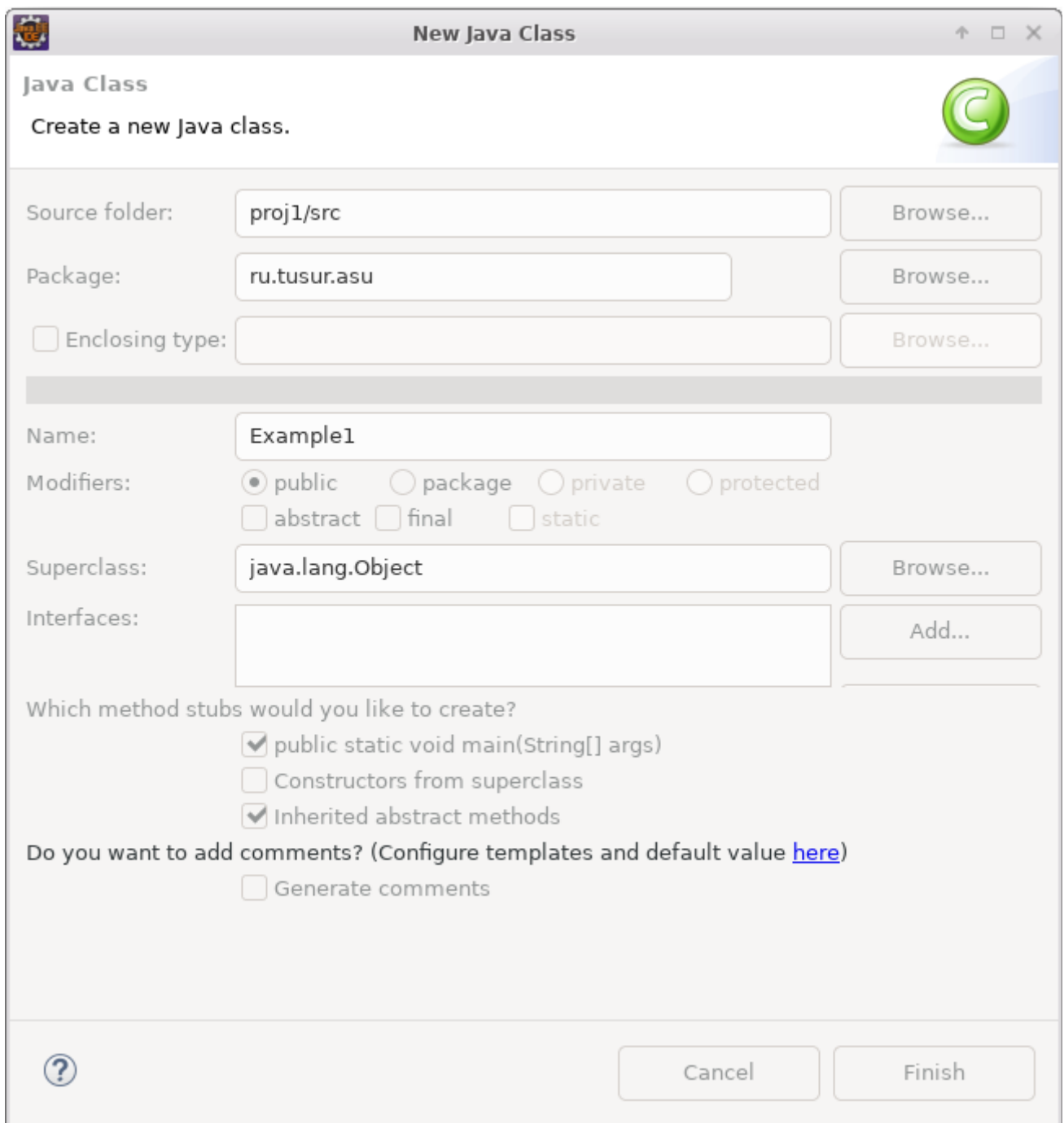


Рисунок 2.8 — Окно создания класса Example1 в проекте proj1

Завершив создание класса кнопкой «**Finish**», мы получаем в среде IDE вкладку **Example1.java** с исходным текстом шаблона класса **Example1**. Внесём в этот шаблон изменения согласно листингу 2.3 и сохраним результат. Проект готов к запуску.

Запустив программу на выполнение, мы по содержимому рисунка 2.9 убеждаемся, что результат ее работы полностью совпадает с результатом рисунка 2.3.

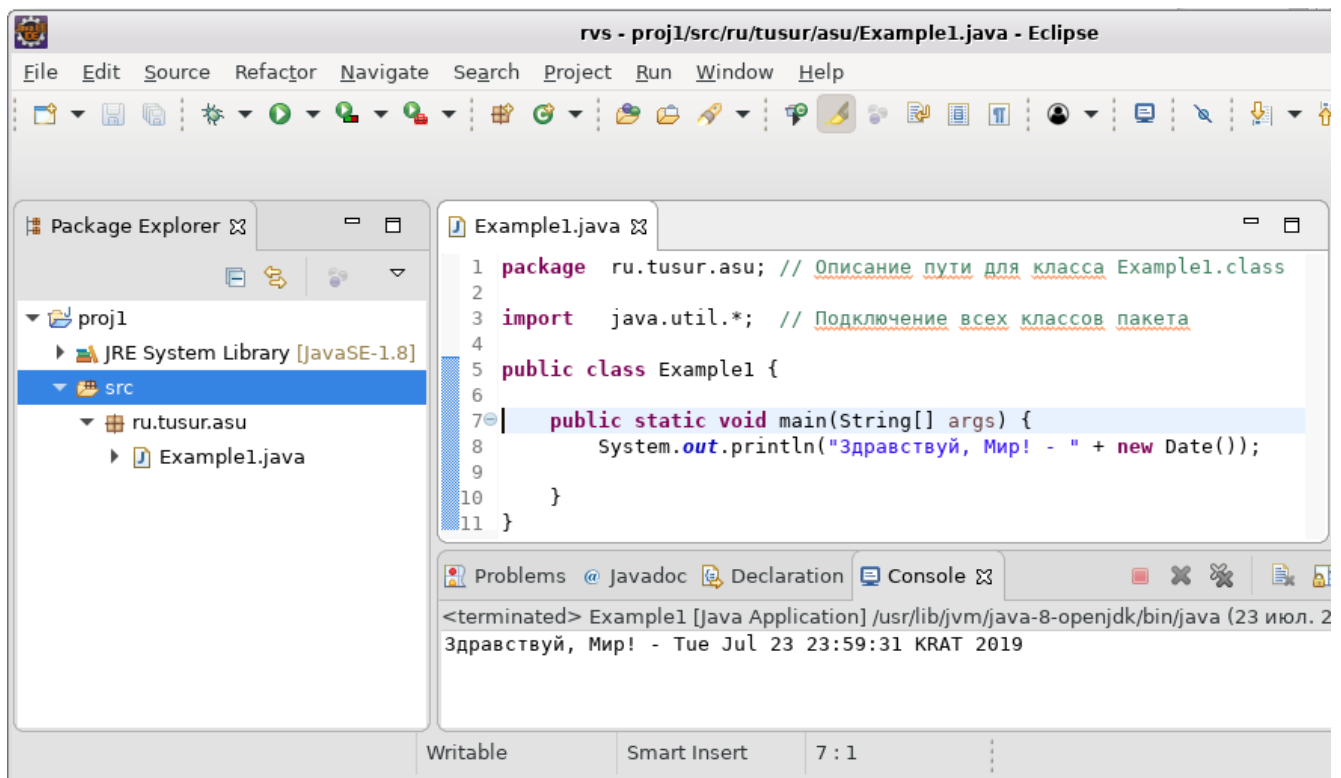


Рисунок 2.9 — Результат запуска проекта `proj1`

Теперь проведём анализ каталога `/home/vgr/rvs/proj1`, где расположены каталоги нашего проекта:

- директория ***bin*** соответствует дереву каталогов для классов проекта;
- директория ***src*** соответствует дереву каталогов для исходных текстов проекта;
- рисунок 2.10 показывает места размещения файлов ***Example1.class*** и ***Example1.java***.

Терминал					
Левая панель		Файл	Команда	Настройки	Правая панель
~ /rvs/proj1/bin/ru/tusur/asu		.	[^]		~ /rvs/proj1/src/ru/tusur/asu
		.и	Размер	Время правки	.и
		/..	-ВВЕРХ-	июл 23 22:59	/..
		Example1.class	769	июл 23 23:59	Example1.java
					319
					июл 23 23:59

Рисунок 2.10 — Места хранения файлов `Example1.class` и `Example.java`

Убедившись, что Eclipse также придерживается правил хранения классов, задаваемых оператором ***package***, и приступим к созданию ***jar***-архива нашего проекта. Для этого, в окне «***Package Explorer***» выделим мышкой проект ***proj1*** и правой кнопкой активируем меню «***Export...***».

Далее, в появившемся окне «***Export***» выбираем «***Runnable JAR file***» и кнопкой «***Next***» переходим к окну, показанному на рисунке 2.11 и заполняем

его с учётом того, что архив имеет имя **proj1.1.jar**, а директория - **/home/vgr/src/rvs**.

Теперь, нажав кнопку «**Finish**», можно сравнить полученный архив с архивом **proj1.jar**.

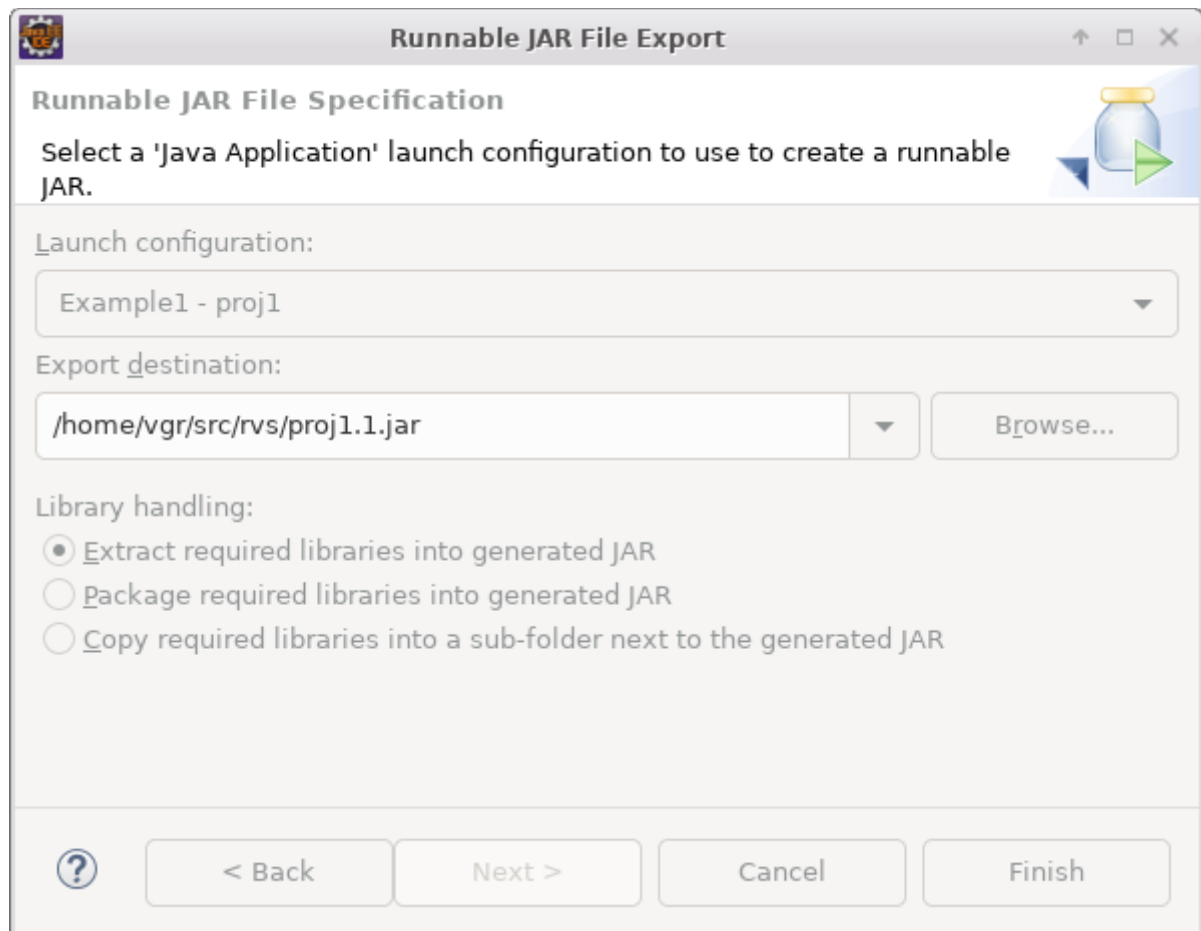


Рисунок 2.11 — Выбор местоположения архива проекта proj1

Запустив программу командой **java -jar proj1.1.jar**, убеждаемся, что она работает также, как и программа архива **proj1.jar**. Можно ещё распаковать архивы командами:

```
jar xf proj1.jar
```

```
jar xf proj1.1.jar
```

и сравнить их содержимое.

Теперь, завершая данный подраздел, можно находиться в уверенности, что студенту представлены все инструменты языка Java, необходимые для изучения последующего материала этой главы.

2.2 Классы и простые типы данных

Как было отмечено ранее, Java является сильно типизированным языком объектно-ориентированного программирования, программное обеспечение которого группируется в виде специализированных пакетов, а дистрибутивы группируются по четырём платформам: *J2EE*, *J2SE*, *J2ME* и *Java Card*. Общая взаимосвязь ПО языка является сложной, но хорошо продуманной системой. Например, официальная документация на платформу Java SE версии 6 группирует все технологии в виде концептуальной диаграммы [30], представленной на рисунке 2.12. Следует также заметить, что на самом сайте эта диаграмма является интерактивной и позволяет легко переходить на соответствующие разделы документации.

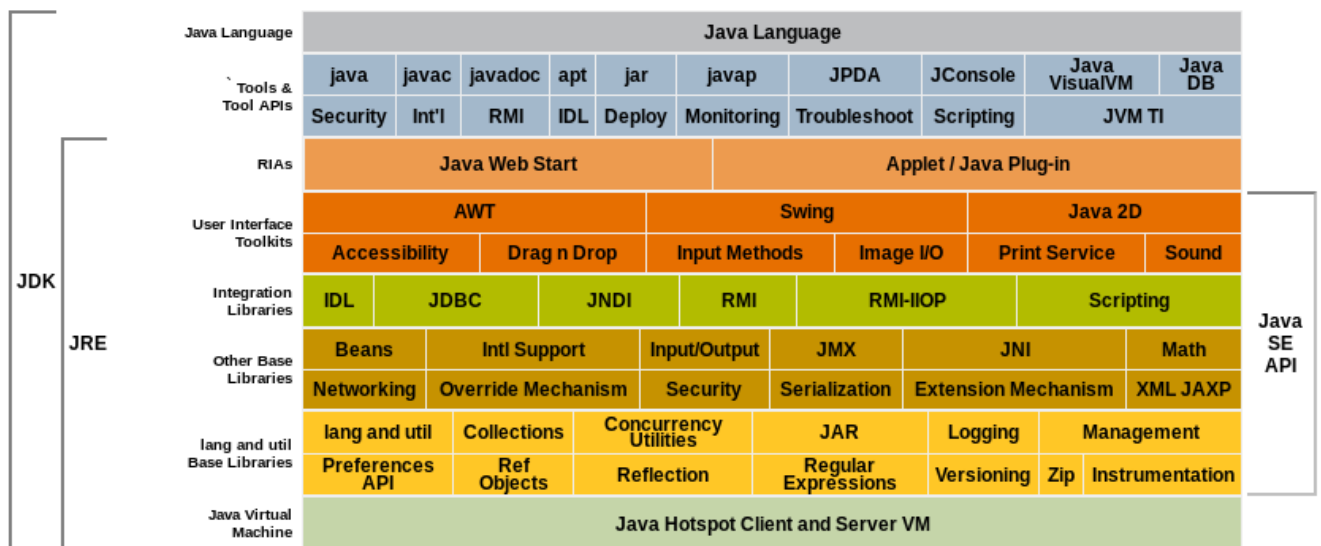


Рисунок 2.12 — Концептуальная диаграмма технологий Java SE версии 6 [30]

Тема данного подраздела посвящена синтаксическим конструкциям языка Java, соответствующим общему понятию ООП «класс». Простейшие примеры использования этого понятия были рассмотрены в предыдущем подразделе, где также отмечалось, что инструментальная поддержка соответствующих конструкций сосредоточена в пакете *java.lang*. Если говорить более точно, то базовой конструкцией языка Java является:

```
public class Object,
```

где *Object* адресуется как *java.lang.Object* и является суперклассом для всех других классов. Все объекты, созданные классами языка Java, включают методы класса *Object*.

В целом учебный материал излагается в справочном стиле. Расчёт идёт

на знание студентом теории и синтаксических конструкций языка C++. Подраздел также не содержит примеров, предполагая дополнительные занятия по лабораторным работам. Изложение учебного материала начинается с представления операторов и семантики простых типов данных, имеющих соответствующий эквивалент во всех языках программирования.

2.2.1 Операторы и простые типы данных

Язык Java имеет восемь простейших типов данных, имеющих аналоги в других языках и перечисленных в таблице 2.1.

Таблица 2.1 - Простые типы языка Java

<i>Тип</i>	<i>Длина (байт)</i>	<i>Диапазон или набор значений</i>
boolean	не определено	true, false
byte	1	-128..127
char	2	0..2 ¹⁶ -1, или 0..65535
short	2	-2 ¹⁵ ..2 ¹⁵ -1, или -32768..32767
int	4	-2 ³¹ ..2 ³¹ -1, или -2147483648..2147483647
long	8	-2 ⁶³ ..2 ⁶³ -1, или примерно -9.2·10 ¹⁸ ..9.2·10 ¹⁸
float	4	-(2·2 ⁻²³)·2 ¹²⁷ ..(2·2 ⁻²³)·2 ¹²⁷ , или примерно -3.4·10 ³⁸ ..3.4·10 ³⁸
double	8	-(2·2 ⁻⁵²)·2 ¹⁰²³ ..(2·2 ⁻⁵²)·2 ¹⁰²³ , или примерно -1.8·10 ³⁰⁸ ..1.8·10 ³⁰⁸

Типы *float* и *double* имеют также специальные значения бесконечности и NaN, которые мы не будем здесь рассматривать. Кроме того, имеется набор зарезервированных слов, которые перечислены в таблице 2.2.

Таблица 2.2 - Зарезервированные слова языка Java

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short

static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

Перечень операторов языка Java представлен в таблице 2.3.

Таблица 2.3 - Операторы языка Java

+	+=	-	-=
*	*=	/	/=
	=	^	^=
&	&=	%	%=
>	>=	<	<=
!	!=	++	--
>>	>>=	<<	<<=
>>>	>>>=	&&	
==	=	~	?:
	instanceof	[]	

Назначение перечисленных обозначений можно уточнить по любому руководству языка Java, мы отметим, что неявные преобразования встроенных типов полностью совпадает с преобразованием типов в языках C/C++, согласно правилам:

1. Если один операнд имеет тип *double*, другой тоже преобразуется к типу *double*.
2. Иначе, если один операнд имеет тип *float*, другой тоже преобразуется к типу *float*.
3. Иначе, если один операнд имеет тип *long*, другой тоже преобразуется к типу *long*.
4. Иначе оба операнда преобразуются к типу *int*.

2.2.2 Синтаксис определения классов

Кроме перечисленных восьми простых типов данных, все остальные типы языка Java относятся к классам, а чтобы иметь возможность преобразо-

вания между ними, используются классы-обёртки представленные в таблице 2.4.

Таблица 2.4 - Простые типы данных и классы-обёртки языка Java

boolean	byte	char	short	int	long	float	double
Boolean	Byte	Character	Short	Integer	Long	Float	Double

Общее определение класса задаётся следующим синтаксическим шаблоном при негласном соглашении, что имя класса должно начинаться с *заглавной* буквы:

```
[модификаторы] class имя-класса
    [extends суперкласс]
    [implements список_интерфейсов]
{
    ...
    // переменные и методы класса
    ...
}
```

В данном и последующих определениях, ключевые слова выделяются полужирным шрифтом, а необязательные элементы конструкции выделены скобками «[...]». Соответственно, в шаблоне класса — выделены три ключевых слова:

- **class** – слово, являющееся обязательным при объявлении класса;
- **extends** – слово, которое используется при объявлении нового класса, на основе уже созданного класса (суперкласса); объявляемый класс называется классом-потомком уже существующего класса;
- **implements** – слово, которое указывает, что класс поддерживает методы, определённые в последующем списке интерфейсов; отдельные элементы списка интерфейсов разделяются запятыми.

Объявляемый класс может не иметь модификатора, тогда «*областью его видимости*» является файл, в котором он определён. Если модификатор присутствует, то он может быть одним из трёх видов:

- **abstract** — класс, имеющий хотя бы один абстрактный метод (метод объявлен, но не имеет реализации); может иметь классы-потомки;
- **public** — открытые (публичные) классы, которые можно использовать: внутри программы, внутри пакета программ или за пределами пакета программ;
- **final** — класс, который не может иметь потомков; примерами таких классов являются: **String** и **StringBuffer**.

2.2.3 Синтаксис и семантика методов

Общее определение метода класса задаётся следующим синтаксическим шаблоном при негласном соглашении, что имя его метода должно начинаться со *строчной* буквы:

```
[спецификатор_доступа]  
  [static] [abstract] [final] [native] [synchronized]  
  тип_данных имя_метода  
  ([параметр1], [параметр2], ...)  
  [throws список_исключений]
```

Спецификатор_доступа — может отсутствовать, но в любом случае он определяет область видимости не только методов, но и переменных:

- **public** - имеется видимость переменной константы или метода из любого класса;
- **private** — доступ разрешается только внутри класса;
- **protected** — доступ разрешается как внутри класса, в котором даётся определение, так и внутри классов-потомков;
- **<пробел>** - доступ разрешается для любых классов пакета, в котором класс определён.

Модификаторы — определяют особенности использования методов, если к ним имеется доступ:

- **static** — метод, принадлежащий классу и используемый всеми объектами класса; обращение к методу производится указанием имени класса и, после точки, указывается метод (оператор **new** указывать не нужно);
- **abstract** — объявляет метод, но не даёт его реализации;
- **final** — метод нельзя будет переопределять (перегружать) в классах-потомках;
- **native** — определяет метод, реализованный на других языках, отличных от языка Java, например, - на языке C;
- **synchronized** — при обращении к методу, доступ к остальным методам класса прекращается до завершения работы данного метода.

Тип_данных — возвращаемый методом тип результата, к которым относятся: типы объявленных классов, простейшие типы данных или **void**, если метод ничего не возвращает.

throws список_исключений — если в методе используются классы или ситуации, приводящие к исключениям, то указывает список необрабатываемых

мых в методе исключений; имена исключений в списке разделяются запятыми.

В качестве замечания отметим, что в языке Java, как и в языках C/C++, имеется особый метод *main(...)*, синтаксис которого имеет стандартный вид:

```
public static void main(String[] args) {  
    // Команды языка Java  
}
```

Если такой метод присутствует в каком-либо классе, то такой класс может быть запущен как самостоятельная программа. Обратите внимание, что этот метод должен быть публичным (*public*) и статическим (*static*), чтобы загрузчик программ мог его увидеть и запустить на выполнение.

Теперь, рассмотрим особые конструкции Java — *интерфейсы*, которые можно рассматривать как специальные типы классов.

2.2.4 Синтаксис определения интерфейсов

Интерфейсы были предложены как альтернатива *абстрактным* типам классов, чтобы расширить возможности последних. Общий синтаксис объявления интерфейсов имеет вид:

```
[public] interface имя_интерфейса  
[extends интерфейс1, интерфейс2, ...]  
{  
    [тип_переменной1 имя_переменной1 = значение1;]  
    [тип_переменной2 имя_переменной2 = значение2;]  
    ...  
    [метод_доступа тип_метода название_метода1([тип_аргумента1 аргумент1], [тип_аргумента2 аргумент2], ...);  
    [метод_доступа тип_метода название_метода2([тип_аргумента1 аргумент1], [тип_аргумента2 аргумент2], ...);  
    ...  
}
```

В определении интерфейса может присутствовать необязательный модификатор *public* и два ключевых слова:

- **interface** — слово, являющееся обязательным при объявлении интерфейса;
- **extends** — слово, которое используется при объявлении нового интер-

фейса, включающего один или более уже объявленных интерфейсов.

Тело интерфейса может содержать:

- объявленные и проинициализированные типы данных;
- описание не реализованных методов.

Интерфейсы можно интерпретировать как незавершённые классы, подобные абстрактным классам, но в отличие от последних они допускают объявление объединений других интерфейсов. Как показано выше, список интерфейсов может присутствовать при объявлении класса, подключая к классу методы, которые должны быть в нем реализованы.

2.2.5 Объекты и переменные

В отличие от языка C++, в языке Java имеются только динамически создаваемые объекты и все переменные языка являются объектными ссылками. С другой стороны, в языке Java отсутствует понятие ссылки и соответствующие операции адресации, доступные в языках C/C++. Чтобы наглядно показать сказанное, обратимся к исходному тексту класса `Example2`, представленному на листинге 2.4.

Листинг 2.4 — Исходный текст класса `Example2` из среды Eclipse EE

```
package ru.tusur.asu;

public class Example2 {

    // Объявление переменной типв String
    String text1;

    // Объявление статического массива из двух целых чисел
    static int[] im = new int[2];

    // Конструктор класса
    Example2(String text2, int n) {
        // Присваиваем значение части переменных
        text1 = text2;
        im[0] = n;
    }

    // Первый (обычный) метод
    public void print1() {
        System.out.println(text1 + ":"
            + " im[0]=" + im[0]
            + " im[1]=" + im[1]);
    }

    // Второй (статический) метод
    public static void print2() {
```



```

        System.out.println("Статический метод: "
            + " im[0]=" + im[0]
            + " im[1]=" + im[1]);
    }
}

```

Этот пример демонстрирует объявление класса *Example2* со следующими особенностями:

- объявляется объектная переменная *text1*, которая неявно инициализируется «пустой» строкой;
- объявляется *статический* массив из двух целых чисел *im*, который неявно инициализируется нулевыми значениями и становится объектом класса;
- объявляется специальный метод *Example2(...)*, который является конструктором класса и, при создании объекта, будет инициализировать значение объекта *text1*, а также — первый элемент массива *im*; таких конструкторов может быть много, но они должны отличаться списком аргументов;
- объявляется обычный *публичный* метод *print1*, который с помощью *статического* метода *println(...)* объекта *java.lang.System.out* распечатывает содержимое строки *text1* и массива *im*;
- объявляется *публичный статический* метод *print2*, который с помощью того же метода *println(...)* распечатывает содержимое массива *im*.

Теперь, чтобы показать детали использования перечисленных особенностей класса *Example2*, определим класс *Example3*, представленный на листинге 2.5.

Листинг 2.5 — Исходный текст класса *Example3* из среды *Eclipse EE*

```

package ru.tusur.asu;

public class Example3 {

    public static void main(String[] args) {

        // Объявляем и создаем объект obj1 класса Example2
        Example2 obj1 =
            new Example2("Первый вызов обычного метода", 1);

        // Вызываем обычный метод объекта obj1
        obj1.print1();

        // Вызываем статический метод класса Example2
        Example2.print2();

        // Объявляем и инициализируем объект obj2 класса Example2
        Example2 obj2 = obj1;

        // Вызываем обычный метод объекта obj2
    }
}

```

```

        obj2.print1();

        // Пересоздаем объект obj1 класса Example2
        obj1 =
            new Example2("Второй вызов обычного метода", 2);

        // Вызываем обычный метод объекта obj1
        obj1.print1();

        // Вызываем статический метод класса Example2
        Example2.print2();

        // Удаляем объект для obj1 и obj2: "сборка мусора"
        obj1 = null;
        obj2 = null;
    }
}

```

Реализуем оба класса *Example2* и *Example3* в одном проекте **proj2** среды IDE Eclipse EE. Запуск класса **Example3** на исполнение приведёт к созданию объектов класса *Example2* и выполнению его различных методов. Результат работы проекта **proj2** представлен на рисунке 2.13.

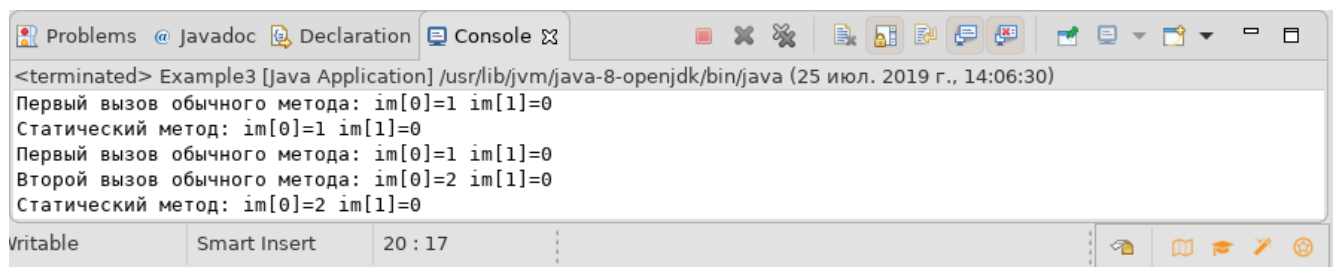


Рисунок 2.13 — Результат запуска проекта proj2 в среде Eclipse EE

2.3 Управляющие операторы языка

Управляющие операторы языка Java практически полностью совпадают с аналогичными конструкциями языков C/C++, поэтому их описание приведено в шаблонном виде и без комментариев.

Оператор if

```

if (логическое_условие)
{
    //Блок операторов, выполняемых при истинности
    //логического условия
}
else if (логическое_условие)

```

```

{
//Блок операторов, выполняемых при истинности
//логического условия
}
...
else
{
//Блок операторов, выполняемых при не истинности
//логического условия
}

```

Оператор switch

```

switch (Проверяемое_выражение)
{
case значение1:
    ...
    //Блок выполняемых операторов
    break;
case значение2:
    ...
    //Блок выполняемых операторов
    break;
default:
    ...
    //Блок выполняемых операторов
}

```

Проверяемое_выражение — это один из перечисляемых типов данных (переменных): *int*, *boolean*, *char* или *byte*.

Оператор цикла while

```

while (логическое_условие)
{
    ...
    //Блок выполняемых операторов
}

```

Оператор цикла do ... while

```
do
{
...
//Блок выполняемых операторов
}
while (логическое_условие)
```

Оператор цикла for

```
for (инициализация; условие; итератор)
{
...
//Блок выполняемых операторов
}
```

Операторы перехода

Операторы перехода используются для более гибкого управления работой циклических операторов. Всего имеется три таких операторов: **break**, **continue** и **return**:

- **break** - обеспечивает безусловный выход из цикла на один уровень;
- **continue** — позволяет игнорировать текущую итерацию цикла и перейти к следующей итерации цикла;
- **return** — используется для безусловного возврата из метода; для методов возвращающих значение, указывает возвращаемый объект или простой тип данных.

2.4 Потоки ввода-вывода

Излишне утверждать, что в разработке сетевых приложений ввод, вывод играет очень важную роль. В языке Java для организации ввода-вывода используются специальные классы, которые должны агрегироваться во вновь определяемые классы и создаваемые объекты. Технология программирования на языке Java использует два общих подхода:

- методы *стандартного* ввода-вывода;
- *обобщённые* методы, основанные на двух классах ***InputStream*** и ***OutputStream***.

Рассмотрим сначала первый подход.

2.4.1 Стандартный ввод-вывод

Все языки программирования содержат средства *стандартного* ввода-вывода. Язык Java не является исключением. Чтобы раскрыть семантику этого факта, рассмотрим рисунок 2.14, хорошо известный студентам по дисциплине «Операционные системы».

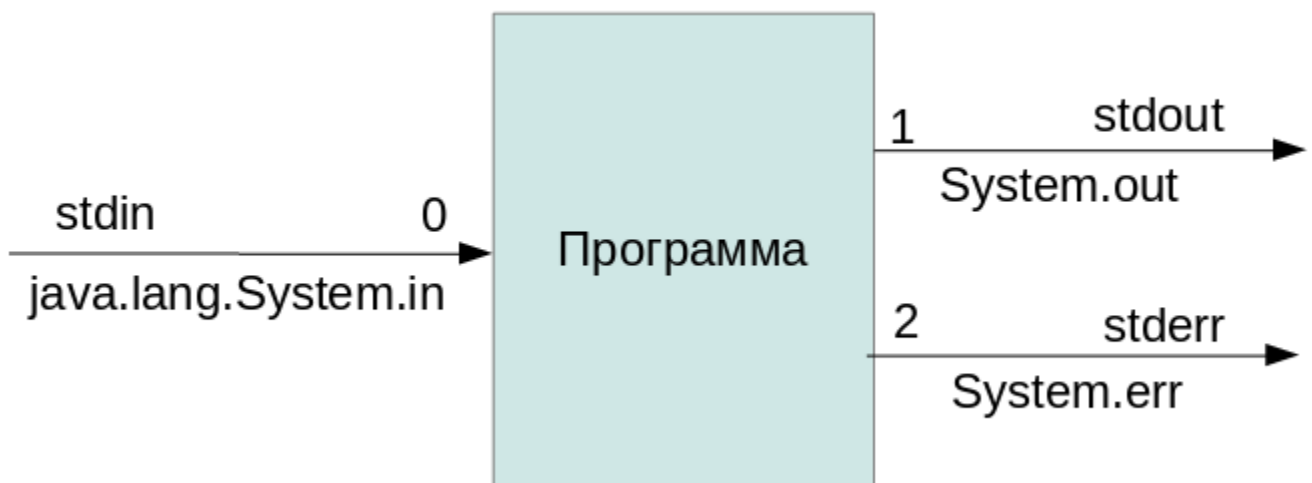


Рисунок 2.14 — Демонстрация семантики стандартного ввода-вывода

Здесь показано, что любая нормально запущенная программа ОС, сразу имеет:

- в командном интерпретаторе - доступные дескрипторы (устройства) 0, 1 и 2;
- в языке C — доступные потоки *stdin*, *stdout* и *stderr*;
- в языке Java — доступны три объекта *System.in*, *System.out* и *System.err*.

Документация по языку Java формально определяет объекты стандартного ввода-вывода в пакете *java.lang*:

```
public static final InputStream in;  
public static final PrintStream out;  
public static final PrintStream err;
```

Таким образом, объекты стандартного ввода-вывода являются *статическими*, что уже отмечалось в рассмотренных примерах, и *финальными*, т.е. — не имеющими возможность иметь дочерние классы.

Объект ввода *in* порожден из класса *InputStream* и имеет три основных метода:

<code>public abstract int read() throws IOException</code>	Читает по одному байту из потока ввода и возвращает целое число в пределах значений от 0 до 255.
<code>public int read(byte[] b) throws IOException</code>	Читает содержимое потока ввода и помещает его в массив байт <i>b</i> . Возвращает число прочитанных байт.
<code>public int read(byte[] b, int off, int len) throws IOException</code>	Читает <i>len</i> байт из потока ввода и помещает его в массив байт <i>b</i> со смещением <i>off</i> . Возвращает число прочитанных байт.

Объекты *out* и *err* порождены классом *PrintStream*, который сам порождён от класса *OutputStream*. Они имеют много методов, но мы представим только четыре:

<code>public void print(String s)</code>	Обеспечивает вывод строки, производя конкатенацию и преобразование типов. В конце строки <i>не добавляет</i> символ перевода строки.
<code>public void println(String s)</code>	Обеспечивает вывод строки, производя конкатенацию и преобразование типов. В конце строки <i>добавляет</i> символ перевода строки.
<code>public void write(byte[] b) throws IOException</code>	Выводит массив байт <i>b</i> .
<code>public void write(byte[] buf, int off, int len)</code>	Выводит <i>len</i> байт из массива <i>b</i> , начиная со смещения <i>off</i> .

Программируя ввод-вывод, необходимо всегда помнить, что многие методы порождают исключения *IOException*, поэтому, определяя собственные методы, необходимо проектировать их обработку или игнорирование. Рассмотрим это примером программы, представленной на листинге 2.6, которая в цикле читает символы с буфера клавиатуры и выводит их на стандартный вывод в виде пары символ-значение.

Листинг 2.6 — Исходный текст класса Example4 из среды Eclipse EE

```
package ru.tusur.asu;
import java.io.IOException; // Для обработки исключения

public class Example4 {

    public static void main(String[] args) {
        // Определение целочисленной рабочей переменной
        int myKey;

        System.out.println("Для выхода из программы, нажмите:\n" +
            "Ctrl-Z в DOS/Windows\n" +
```

```

        "Ctrl-D в UNIX/Linux");
    try
    {
        while ((myKey = System.in.read()) != -1)
        {
            System.out.println((char)myKey + ": " + myKey);
        }
    }
    catch(IOException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Другой пример, использующий динамически определяемый байтовый буфер, представлен на листинге 2.7.

Листинг 2.7 — Исходный текст класса Example5 из среды Eclipse EE

```

package ru.tusur.asu;
import java.io.IOException; // Для обработки исключения

public class Exemple5 {

    public static void main(String[] args) {

        byte[] b; // Объявление байтового массива
        int n;     // Переменная хранения размера массива

        System.out.println("Программа Example5 делает 5 циклов:\n");
        try
        {
            for (int i = 0; i < 5; i++)
            {
                // Ожидаем ввод и определяем число символов
                // в буфере терминала
                while ((n = System.in.available()) == 0) ;
                b = new byte[n]; // Создаем массив буфера b
                System.in.read(b); // Читаем символы в буфер b
                // Печатаем весь буфер
                System.out.println(new String(b));
            }
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

2.4.2 Классы потоков ввода

Для решения общих задач ввода-вывода язык Java использует специаль-

ный пакет *java.io*, обладающий набором классов и интерфейсов, позволяющих управлять процессами внутри различных потоков. Базовым классом для методов ввода является *InputStream*. Его диаграмма наследования представлена на рисунке 2.15, а общий синтаксис объявления имеет вид:

InputStream имя_объекта =
new Конструктор_одного_из_классов_ввода

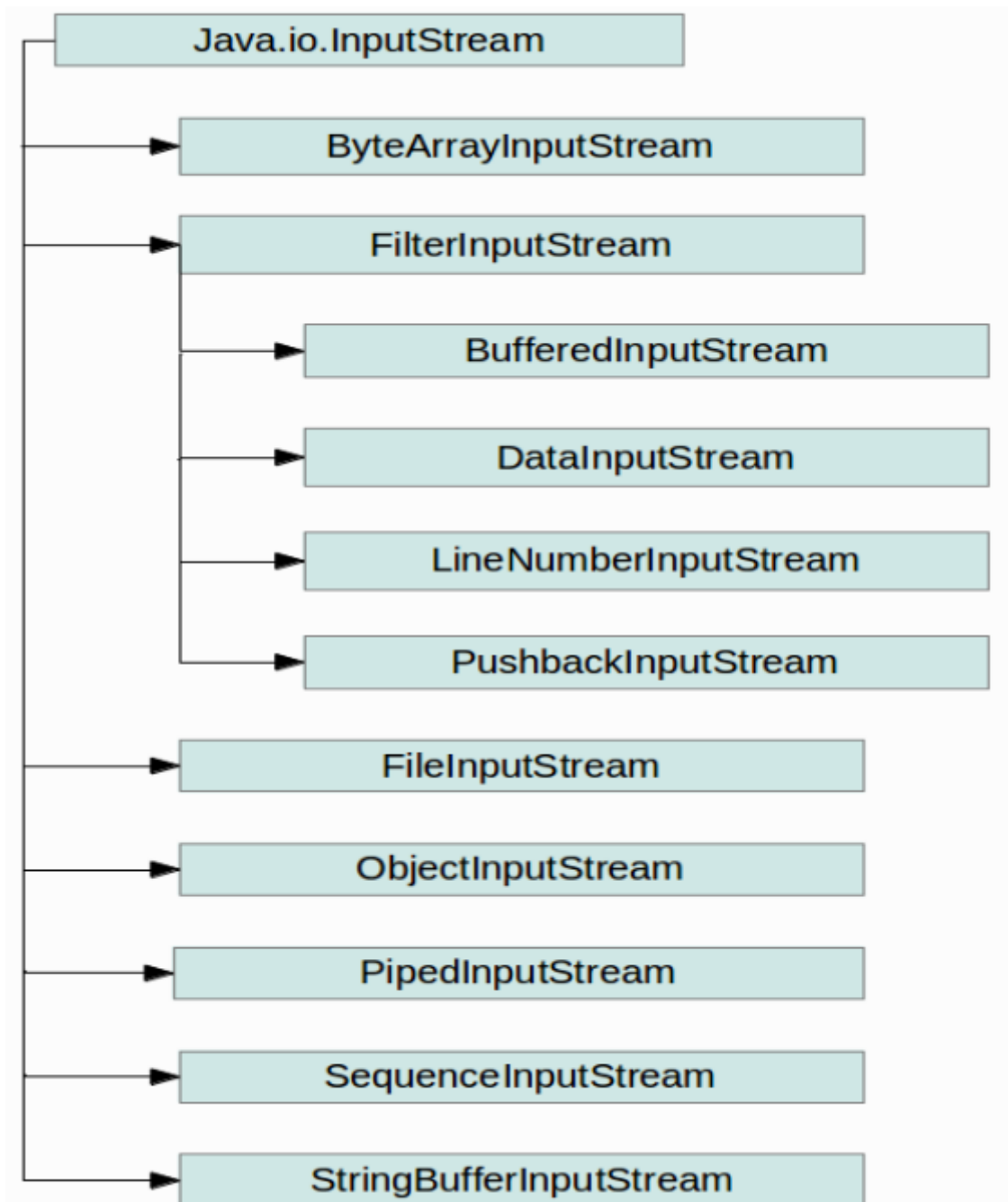


Рисунок 2.15 — Диаграмма наследования класса *InputStream*

Объектам класса ***InputStream*** доступны все методы, которые также доступны стандартному вводу. Обычно, на практике используются ещё методы:

<i>available()</i>	Возвращает число байт, доступных для чтения в потоке ввода.
<i>skip(long N)</i>	Пропускает во входном потоке N байт.
<i>close()</i>	Закрывает входной поток ввода.

Рассмотрим программу листинга 2.8, демонстрирующую использование входного потока ***FileInputStream*** на примере чтения текстового файла и отображения результата чтения на стандартный вывод.

Листинг 2.8 — Исходный текст класса Example6 из среды Eclipse EE

```
package ru.tusur.asu;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class Example6 {

    public static void main(String[] args) {

        // Объявление буфера и счетчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Демонстрация потока FileInputStream.\n"
            + "Чтение файла: /home/vgr/src/rvs/Example1.java\n"
            + "-----\n");

        try{
            InputStream in =
                new FileInputStream("/home/vgr/src/rvs/Example1.java");

            while ((n = in.available()) > 0) // Читаем в цикле
            {
                b = new byte[n];
                in.read(b);
                System.out.print(new String(b));
            }

            in.close();          // Закрываем поток ввода

        }catch(IOException e)    // Обрабатываем исключение
        {
            System.out.println(e.getMessage());
        }

    }
}
```

Следует обратить внимание, что чтение файла осуществляется в цикле, потому что входной буфер всегда ограничен в размере и реальное чтение, в

общем случае, происходит блоками переменной длины.

2.4.3 Классы потоков вывода

Для реализации общих задач вывода информации в языке Java используется базовый класс ***OutputStream***, также принадлежащий пакету ***java.io***. Диаграмма наследования его классов-потомков приведена на рисунке 2.16, а общий синтаксис объявления имеет вид:

OutputStream имя_объекта =
new Конструктор_одного_из_классов_вывода

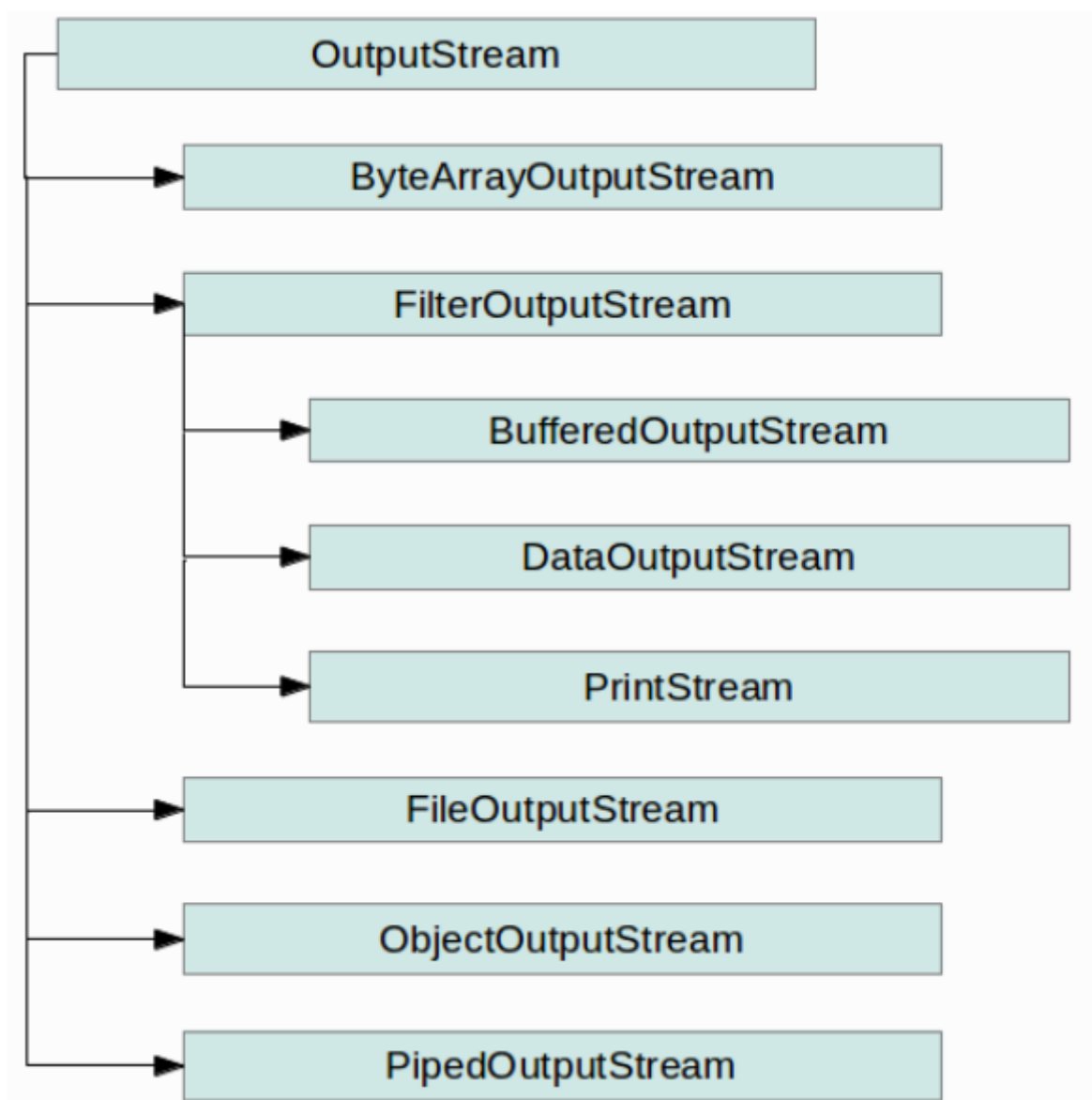


Рисунок 2.16 — Диаграмма наследования класса OutputStream

Обратим внимание, что на диаграмме присутствует класс стандартного вывода ***PrintStream***, который является дочерним классом от класса ***FilterOutputStream***. Сам же класс ***OutputStream*** имеет много методов, подобных стандартному выводу, а наиболее значимые из них:

<i>write(int b)</i>	Запись одного байта <i>b</i> в поток вывода.
<i>write(byte[] b)</i>	Запись массива байт <i>b</i> в поток вывода.
<i>write(byte[] b, int off, int len)</i>	Запись в поток вывода части массива <i>b</i> длиной <i>len</i> , начиная с позиции <i>off</i> .
<i>flush()</i>	Принудительный сброс данных из промежуточного буфера в поток вывода.
<i>close()</i>	Закрывает выходной поток вывода.

Рассмотрим программу листинга 2.9, демонстрирующую использование выходного потока ***FileOutputStream*** на примере создания текстового файла с помощью объекта класса ***OutputStream***, последующего чтения этого файла и отображения результата чтения на стандартный вывод.

Листинг 2.9 — Исходный текст класса Example7 из среды Eclipse EE

```
package ru.tusur.asu;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Example7 {

    public static void main(String[] args) {

        // Объявление буфера и счетчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Демонстрация потока FileOutputStream.\n"
            + "Создание и чтение файла: /home/vgr/demo7.txt \n"
            + "-----\n");

        try{

            OutputStream out =
                new FileOutputStream("/home/vgr/demo7.txt");

            // Пишем строки в файл
            out.write("Привет всем друзьям!\n".getBytes());
            out.write("Мы написали программу записи в файл.\n".getBytes());
            out.write("Посмотрим, что из этого получится...\n".getBytes());
            out.flush();           // Сбрасываем поток вывода
            out.close();           // Закрываем поток вывода
```

```

        InputStream in =
            new FileInputStream("/home/vgr/demo7.txt");

        while ((n = in.available()) > 0) // Читаем в цикле
        {
            b = new byte[n];
            in.read(b);
            System.out.print(new String(b));
        }

        in.close(); // Закрываем поток ввода

    } catch (IOException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Следует особо обратить внимание, что при записи в файл:

- используется метод ***write(byte[] b)***, оперирующий с заранее созданным массивом байт;
- строки символов, заключённые в двойные кавычки, являются объектами и к ним применяется *статический* метод ***getBytes()*** из класса ***java.lang.String***, динамически преобразующий объект строки в массив байт;
- перед закрытием любого выходного потока методом ***close()*** всегда нужно сбрасывать промежуточный буфер вывода методом ***flush()***.

Завершая изучение общих вопросов программирования на языке Java, рассмотрим пример использования класса ***File***, представленный на листинге 2.10 и демонстрирующий ряд характеристик уже созданного файла ***/home/vgr/demo7.txt***.

Листинг 2.10 — Исходный текст класса Example8 из среды Eclipse EE

```

package ru.tusur.asu;

import java.io.File;
import java.util.Date;

public class Example8 {

    public static void main(String[] args) {
        // Определяем и создаем объект класса File
        File myf = new File("/home/vgr/demo7.txt");

        System.out.println("Это - файл: " + myf.isFile());
        System.out.println("Это - директория: " + myf.isDirectory());

        System.out.println("Можно писать в файл: " + myf.canWrite());
        System.out.println("Можно читать файл: " + myf.canRead());
    }
}

```

```

System.out.println("Можно запускать файл: " + myf.canExecute());

System.out.println("Имеет родителя: " + myf.getParent());
System.out.println("Имеет путь: " + myf.getPath());
System.out.println("Имеет имя: " + myf.getName());

System.out.println("Длина файла: " + myf.length());
System.out.println("Последняя модификация: "
    + new Date(myf.lastModified()));
}
}

```

2.5 Управление сетевыми соединениями

Управление сетевыми соединениями является важной темой для нашей дисциплины. В языке Java имеется специальный пакет *java.net*, содержащий базовые классы и методы для работы со стеком протоколов TCP/IP и предполагающий, что компьютер студента подключён к сети и может пользоваться службой доменных имён (DNS). Мы исходим из того, что студентом уже изучен курс «*Сети и телекоммуникации*», поэтому изложение материала не содержит терминов и определений, излагаемых в этой дисциплине. Следуя содержанию ПО пакета *java.net*, в данном подразделе выделяются следующие части:

- адресация в Internet, основанная на классе *InetAddress*;
- адресация в Internet на основе классов *URL* и *URLConnection*;
- сокет протокола *TCP*;
- сокет протокола *UDP*;
- пример технологии клиент-сервер.

2.5.1 Адресация на базе класса *InetAddress*

Работая с Internet в стеке протоколов *TCP/IP версии 4*, мы привыкли к цифровым адресам в виде четырёх чисел, разделённых точками, и доменным адресам, представляющим слова, разделённые точками. Например:

- **192.168.0.20** — цифровой адрес компьютера в локальной сети класса С;
- **asu.tusur.ru** — доменное имя web-сервера кафедры АСУ.

В языке Java в качестве адресов используются объекты класса *InetAddress*, а привычные нам адреса Internet — в качестве строковых аргументов в трех *статических* методах этого класса:

<i>getLocalHost()</i>	Создаёт объект адреса для локального компьютера.
<i>getByName(String host)</i>	Создаёт объект адреса для <i>host</i> : строкового значения цифрового или доменного адреса Internet. Если вместо <i>host</i> указать <i>null</i> , то

	объект соответствует цифровому имени: 127.0.0.1
<i>getAllByName(String host)</i>	Создаёт массив объектов адресов для <i>hosts</i> : строкового значения цифрового или доменного адреса Internet. Если вместо <i>host</i> указать <i>null</i> , то объект соответствует цифровому имени: 127.0.0.1

Созданные объекты класса ***InetAddress*** используются в методах других классов, например в классах сокетов, но также имеют два собственных метода:

<i>getHostAddress()</i>	Возвращает адрес для объекта <i>InetAddress</i> .
<i>getHostName()</i>	Возвращает имя для объекта <i>InetAddress</i> .

Для примера рассмотрим программу листинга 2.11, в которой создаются объекты класса ***InetAddress*** для локального компьютера, доменного имени кафедры АСУ и доменного имени ***www.yandex.ru*** (см. также рисунок 2.17).

Листинг 2.11 — Исходный текст класса Example9 из среды Eclipse EE

```
package ru.tusur.asu;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class Example9 {

    public static void main(String[] args) {

        System.out.println("Использование класса InetAddress.\n"
                           + "-----");

        try {
            // Создаем объекты адресов
            InetAddress addr1 =
                InetAddress.getLocalHost();
            InetAddress addr2 =
                InetAddress.getByName("asu.tusur.ru");
            InetAddress[] addr3 =
                InetAddress.getAllByName("www.yandex.ru");

            // Выводим на печать содержимое объектов
            System.out.println("Локальная ЭВМ: "
                               + addr1.getHostAddress() + " | " + addr1.getHostName());

            System.out.println("Кафедра АСУ : "
                               + addr2.getHostAddress() + " | " + addr2.getHostName());

            for(int i=0; i < addr3.length; i++)
                System.out.println("Яндекс      : "
                                   + addr3[i].getHostAddress() + " | "
                                   + addr3[i].getHostName());

        } catch (UnknownHostException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```

    }
}

<terminated> Example9 [Java Application] /usr/lib/jvm/java-8-openjdk/bin/java (26 июл. 2019 г., 21:31:51)
Использование класса InetAddress.
-----
Локальная ЭВМ: 78.140.20.13 | upkasu
Кафедра АСУ : 88.204.72.158 | asu.tusur.ru
Яндекс : 5.255.255.80 | www.yandex.ru
Яндекс : 5.255.255.88 | www.yandex.ru
Яндекс : 77.88.55.77 | www.yandex.ru
Яндекс : 77.88.55.80 | www.yandex.ru
Яндекс : 2a02:6b8:a:0:0:0:0:a | www.yandex.ru

```

Рисунок 2.17 — Результат работы программы листинга 2.11

2.5.2 Адресация на базе *URL* и *URLConnection*

Для работы с web-протоколами такими как, *http*, *ftp* и другими, в пакете *java.net* имеются классы:

- **URL** — класс для непосредственной работы с *URL*-адресами;
- **URLConnection** — класс, методы которого обеспечивают процессы загрузки ресурсов Internet и их информационную поддержку с использованием объектов класса *URL*.

Поддерживается следующая схема *URL*-адресов **protocol://host:port/file**, которая обеспечивается тремя конструкторами класса *URL*:

```

URL(String saddr);
URL(String protocol, String host, String file);
URL(String protocol, String host, int port, String file);

```

Объекты класса *URL* имеют ряд важных методов:

- **getProtocol()**, **getHost()**, **getPort()**, **getFile()** - возвращают значения компонент соответствующего *URL*-адреса;
- **openConnection()** - возвращает объект класса *URLConnection*;
- **openStream()** - возвращает объект входного потока класса *InputStream*.

На листинге 2.12 представлен исходный текст программы, использующей *URL*-адрес «<http://asu.tusur.ru/>» для чтения доступного по нему ресурса и записи его в файл */home/vgr/demo10.html*.

Листинг 2.12 — Исходный текст класса *Example10* из среды *Eclipse EE*

```
package ru.tusur.asu;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.MalformedURLException;
import java.net.URL;

public class Example10 {

    public static void main(String[] args) {

        // Объявление буфера и счетчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Использование класса URL\n"
            + "Сохраняем в файле: /home/vgr/demo10.html\n"
            + "-----");

        try
        {
            URL url = // Объявление и создание объекта
                new URL("http://asu.tusur.ru/");

            // Печатаем компоненты URL-адреса
            System.out.println("Протокол: " + url.getProtocol() + "\n"
                + "Хост : " + url.getHost() + "\n"
                + "Порт : " + url.getPort() + "\n"
                + "Файл : " + url.getFile() + "\n");

            InputStream in = // Открываем входной поток
                url.openStream();
            OutputStream out = // Открываем выходной поток
                new FileOutputStream("/home/vgr/demo10.html");

            while ((n = in.available()) > 0) // Читаем и пишем в цикле
            {
                b = new byte[n];
                in.read(b);
                //System.out.print(new String(b));
                out.write(b);
            }

            in.close(); // Закрываем потоки
            out.close();

        }
        catch (MalformedURLException e1)
        {
            System.out.println(e1.getMessage());
        }
        catch (IOException e2)
        {
            System.out.println(e2.getMessage());
        }
    }
}
```



```

    }
}

```

Полученный программой файл можно посмотреть в любом браузере, но следует помнить, что объекты класса **URL** предназначены для работы с адресами и не содержат средств анализа адресуемого ресурса.

Для манипулирования адресуемыми ресурсами используется объект класса **URLConnection**, который можно получить командой:

```
URLConnection ucon = url.openConnection();
```

где *url* — объект класса **URL**, имеющий множество методов. Например:

- **connect()** - устанавливает соединение с ресурсом, если оно ещё не было установлено, или вызывает исключение, если соединение — невозможно;
- **getContentLength()** - возвращает размер ресурса;
- **getContentType()** - возвращает тип содержимого адресуемого ресурса;
- **getDate()** - возвращает дату загрузки;
- **getExpiration()** - возвращает срок хранения;
- **getPermission()** - определяет параметры доступа к ресурсу;
- **getInputStream()** - возвращает объект потока ввода класса **InputStream**.

2.5.3 Сокеты протокола TCP

Для организации взаимодействия между двумя программами по протоколу TCP пакет **java.net** предоставляет два класса:

- **Socket** — класс для создания объектов клиентского сокета;
- **ServerSocket** — класс создающий объекты, используемые серверными программами для организации соединения с программами клиентов.

Чтобы программа-клиент могла соединиться с сервером, она должна знать адрес и порт сервера, а затем создать объект клиентского сокета командой:

```
Socket client =
    new Socket(InetAddress address, int port);
```

Чтобы программа-сервер могла принимать соединения от клиентов, она должна создать объект типа **ServerSocket** командой:

```
ServerSocket server =
    new ServerSocket(int port);
```

Получив запрос на соединение, программа-сервер должна согласиться на него и создать объект клиентского сокета командой:

```
Socket client =  
    server.accept();
```

Для возможности непосредственного обмена данными обе программы, и клиент и сервер, должны создать входные и выходные потоки, используя классы пакета *java.io*: и методы объектов класса *Socket*:

```
InputStream in =  
    client.getInputStream();
```

```
OutputStream out =  
    client.getOutputStream();
```

Далее, обмен данными между программами выполняется через объекты потоков ввода-вывода с помощью соответствующих методов *read(...)* и *write(...)*.

Описанная выше технология обмена данными между двумя программами называется — *технология клиент-сервер*, а метод взаимодействия — *синхронным*. Алгоритм, по которому осуществляется обмен данными, - *протоколом*. Соответствующий пример программы приводится в пункте 2.5.5.

2.5.4 Сокеты протокола UDP

Для организации асинхронного взаимодействия между программами пакет *java.net* предоставляет классы *DatagramSocket* и *DatagramPacket*, реализованные на транспортном уровне протокола UDP.

Сокет протокола UDP создаётся объектом класса *DatagramSocket*, который имеет три конструктора:

```
DatagramSocket();  
DatagramSocket(int port);  
DatagramSocket(int port, InetAddress addr);
```

Каждый объект класса *DatagramSocket* имеет следующие основные методы:

- *getInetAddress()* - возвращает адрес, к которому осуществляется подключение;
- *getPort()* - возвращает порт, к которому осуществляется подключение;

- ***getLocalAddress()*** - возвращает локальный адрес компьютера, с которого осуществляется подключение;
- ***getLocalPort()*** - возвращает локальный порт, через который осуществляется подключение;
- ***send(DatagramPacket pack)*** — передача пакета;
- ***receive(DatagramPacket pack)*** - приём пакета.

Пакеты протокола UDP создаются объектами класса ***DatagramPacket***, который имеет два конструктора:

- ***DatagramPacket(byte[] buf, int length)*** — создать пакет данных размера ***length*** на основе массива байт - ***buf***;
- ***DatagramPacket(byte[] buf, int length, InetAddress addr, int port)*** - создать полный пакет данных размера ***length*** на основе массива байт — ***buf***, предназначенный для передачи по адресу ***addr*** и порту ***port***.

Каждый объект класса ***DatagramPacket*** имеет следующие основные методы:

- ***getAddress()*** и ***setAddress()*** - возвращает или устанавливает адрес подключения;
- ***getPort()*** и ***setPort()*** - возвращает или устанавливает порт подключения;
- ***getLength()*** и ***setLength()*** - возвращает или устанавливает размер дейтаграммы;
- ***getData()*** - возвращает массив байт содержимого дейтаграммы;
- ***getData(byte[] buf)*** — позволяет записать данные в пакет.

Естественно, что двусторонний обмен между программами с помощью дейтаграмм требует организовать с каждой стороны как коды клиента, так и коды сервера. Обычно, это требует мультинитевой организации взаимодействующих программ. Примеры таких программ — это тема лабораторных работ.

2.5.5 Простейшая задача технологии клиент-сервер

Решение задач технологии *клиент-сервер* требует от программиста трёх навыков:

- определение общего состава прикладных функций решаемой задачи;
- распределение прикладных функций между программой клиента и программой сервера;
- описание протокола обмена данными между программой клиента и программой сервера;

Поскольку познавательная цель данного пункта - демонстрация исполь-

зования классов и методов пакета *java.net* языка Java, то минимизируем прикладную часть задачи до простейшего уровня:

- программа-**клиент** передаёт на **сервер** конечную последовательность текстовых сообщений, ожидая последовательное подтверждение на приём каждого из них;
- программа-**сервер** принимает отдельные сообщения и подтверждает каждое из них передачей **клиенту** собственного текстового сообщения.

Решение задачи проведём на базе протокола TCP, который предполагает организацию соединения между программами до передачи и приема сообщениями, а также разрыв соединения после обмена данными. Для конкретизации технологии *клиент-сервер* предполагаем, что:

- программа-**сервер** запускается и ждёт запрос на соединение от программы-**клиента**; после установки соединения, она производит обмен сообщениями, а после разрыва соединения по инициативе **клиента** завершает свою работу;
- программа-**клиент** инициирует соединение с сервером, производит обмен сообщениями, разрывает соединение и завершает свою работу.

Реализация протокола поставленной задачи требует определения *средств синхронизации* процессов обмена данными между взаимодействующими программами. Такие средства являются обязательными для задач, реализуемых на базе протоколов транспортного уровня. Они включают:

- определение аварийных ситуаций, фиксируемых программными средствами языка Java;
- определение состояний блокировки обмена данными между взаимодействующими программами;
- организацию перехода программ в фиксированные исходные состояния для продолжения процессов взаимодействия (сам процесс синхронизации).

В рамках нашей задачи, средства синхронизации фиксируются следующим набором правил:

- любая аварийная ситуация, фиксируемая программными средствами языка Java, приводит к завершению программ;
- монитором и инициатором синхронизации является программа-**клиент**;
- событием конца отдельного сообщения является приём любой стороной диалога символа перевода строки «\n»;
- состояние блокировки обмена данными определяет программа-**клиент** посредством нарушения границ фиксированного тайм-аута при ожидании подтверждения на приём сообщения от программы-**сервера**;
- программа-**клиент** инициирует синхронизацию посылкой серверу сим-

- программа-*сервер* должна вернуть символ «\r» в пределах границы тайм-аута, иначе программа-*клиент* разрывает соединение и завершает работу.

В пределах изложенной постановки задачи, программа-*сервер* реализована в виде класса *TCPServer* и представлена на листинге 2.13.

Листинг 2.13 — Исходный текст класса TCPServer из среды Eclipse EE

```
package asu.server;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Date;

/*
 * Сервер стартует и прослушивает порт.
 * Сервер акцептирует запрос на соединение, создает потоки ввода/вывода
 * и в цикле выполняет:
 * 1) читает входной поток посимвольно и выводит их на экран;
 * 2) получив символ "\n" == 10, посылает клиенту строку "OK\n";
 * 3) получив символ "\r" == 13, посылает его назад клиенту;
 * 4) завершает работу по любому исключению или закрытию входного потока.
 */

public class TCPServer {

    public static void main(String[] args)
    {
        int port;                //Номер порта сервера
        String smes= "OK\n";      //Сообщение посылаемое клиенту

        ServerSocket svrsocket; //Серверный сокет
        Socket clientsocket;     //Клиентский сокет

        //Время получения первого пакета
        long statime = new Date().getTime();

        //Текущее время относительно акцепта соединения
        long curtime = 0;

        InputStream in;          //Входной поток сервера
        OutputStream out;        //Выходной поток сервера

        int ch = (int)'\n';      //Символ конца строки
        System.out.println("\n\n = " + ch);
        int cr = (int)'\r';      //Символ синхронизации
        System.out.println("\r = " + cr + "\n");

        try{
```

```

port = // Читаем номер порта как аргумент программы
      new Integer(args[0]).intValue();

InetAddress localhost = //Адресс сервера
      InetAddress.getLocalHost();
System.out.println("Server Address: "
      + localhost.getHostAddress());
System.out.println("Port Address: "
      + port); //args[0]);

//Объявляем порт прослушивания
svrsocket = new ServerSocket(port);
System.out.println("TCPserver start: " + new Date());

//Прослушиваем порт
clientsocket = svrsocket.accept(); //Ожидаем соединения
statime = new Date().getTime();
System.out.print(curtime +
      ": TCP-server - accept conection...\n\n" + "0: ");

//Открываем потоки ввода/вывода
in = clientsocket.getInputStream();
out = clientsocket.getOutputStream();

//Цикл диалога с клиентом
// Читаем по байту, пока не закроется входной поток сокета
while((ch=in.read()) != -1){

    System.out.print((char)ch);
    //Перевод строки - значит строка закончилась
    if(ch == 10){
        curtime = new Date().getTime();
        System.out.print((curtime - statime) + ": ");

        //Отвечаем клиенту "OK\n"
        out.write(smes.getBytes());
    }
    //Возврат каретки - значит синхронизация с клиентом
    if(ch == 13) out.write("\r".getBytes());

}

out.flush(); // Освобождаем поток вывода
out.close(); // Закрываем потоки
in.close(); //
clientsocket.close(); // Закрываем сокеты
svrsocket.close(); //

}catch(UnknownHostException ue){
    System.out.println("UnknownHostException: " + ue.getMessage());
}catch(IOException e){
    System.out.println("IOException: " + e.getMessage());
}catch(Exception ee){
    System.out.println("Exception: " + ee.getMessage());

    // Подсказка для запуска программы
    System.out.println("\nrun: java asu.server.TCPServer port\n");
}

```

```

        curtime = new Date().getTime(); // Подводим итог
        System.out.println("\n" + (curtime - statime)
            + ": TCP-server - stop");

        System.exit(0); // Системное завершение программы
    }
}

```

Для нормального запуска программы-*сервера*, необходимо указать в качестве ее аргумента номер используемого порта, например, - число 8888. А поскольку мы реализуем программы в среде Eclipse EE, то необходимо зайти в каталог *bin* ее проекта и выполнить команду:

```
$ java asu.server.TCPServer 8888
```

Результат такого запуска сервера показан на рисунке 2.18.

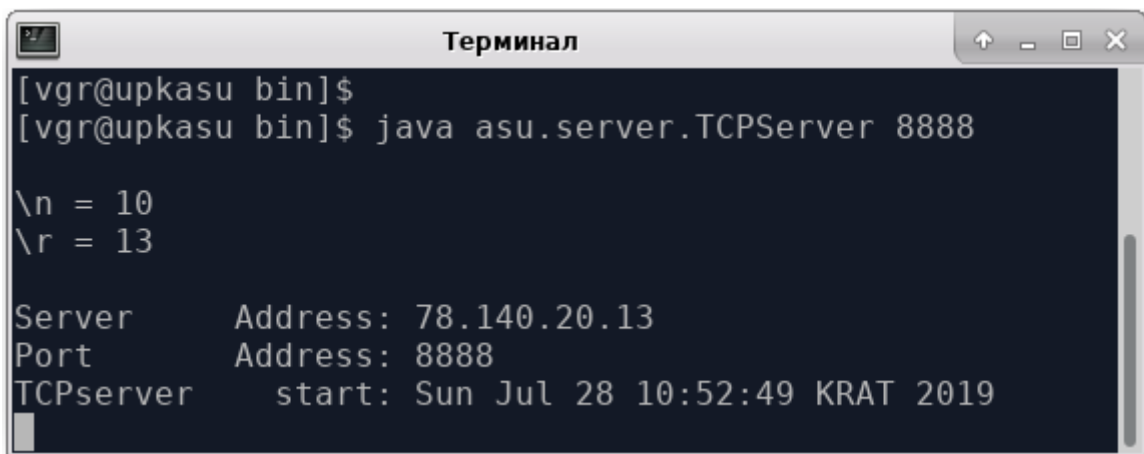


Рисунок 2.18 — Результат работы программы-сервера

Соответствующая программа-*клиент* реализована в виде класса *TCPClient* и представлена на листинге 2.14.

Листинг 2.14 — Исходный текст класса TCPClient из среды Eclipse EE

```

package asu.client;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Date;

/*
 * Клиент читает параметры командной строки:
 * адрес сервера, порт сервера, количество посылаемых сообщений, текст сооб-

```

щения.

```
* Клиент устанавливает соединение, создает потоки ввода/вывода
* и в цикле выполняет:
* 1) выводит на экран то, что посылает: "<номер сообщения>" + "<сообщение>";
* 2) выводит на экран символ входной поток символов;
* 3) получив символ "\n", переходит к п.1;
* 4) если входной поток пуст более dTime, то серверу посылается символ '\r';
* 5) получив от сервера символ '\r', то посылает пакет повторно;
* 6) завершает работу по любому исключению; после передачи заданного
* количества сообщений или когда входной канал - пуст; тайм-аут
* ожидания подтверждения больше dTime после reset (посылки серверу
* символа '\r').
*/
```

```
public class TCPClient {

    public static void main(String[] args)
    {
        InetAddress remotehost; //Адрес сервера
        int port = 0;           //Номер порта сервера
        int nn = 0;             //Количество сообщений
        String mes = " ";       //Сообщение клиента

        Socket clientsocket;    //Объявление клиентского сокета

        long curtime = 0;
        long dTime = 100;
        long rTime = 0;
        boolean flagRepeat = false;

        InputStream in;
        OutputStream out;

        int ch = (int)'\n';
        System.out.println("\n = " + ch);
        int cr = (int)'\r';
        System.out.println("\r = " + cr + "\n");

        try{
            System.out.println("\nTCP-client - start: " + new Date());

            //Чтение и инициализация аргументов программы
            remotehost = InetAddress.getByName(args[0]);
            System.out.println("Server Address: "
                               + remotehost.getHostAddress()); //args[0]);

            port = new Integer(args[1]).intValue();
            System.out.println("ServerPort Address: "
                               + port); //args[1]);

            dTime = new Integer(args[2]).longValue();
            System.out.println(" Time_out (msec): "
                               + args[2]);

            nn = new Integer(args[3]).intValue();
            System.out.println("Count Message: "
                               + args[3]);

            mes = args[4];
```



```

System.out.println("Client      Message: "
                  + args[4] + "\n");

//Устанавливаем соединение с сервером
clientsocket =
    new Socket(remotehost, port);
System.out.println("TCPclient connect: " + new Date());
long statime = new Date().getTime();

//Создание потоков ввода/вывода
in = clientsocket.getInputStream();
out = clientsocket.getOutputStream();

//Цикл диалога с сервером
for(int i=1; i<=nn; i++){
    //Посылаем серверу сообщение
    mes = String.valueOf(i) + " " + args[4] + "\n";
    out.write(mes.getBytes());

    //Фиксируем границу тайм-аута
    rTime = new Date().getTime() + dTime;
    curtime = new Date().getTime();
    System.out.println((curtime - statime) + ": " + mes);

    //ожидаем ответ сервера
    boolean flag = true;
    while(flag){
        if(in.available() > 0){

            //Блокирующая операция чтения одного байта
            ch = in.read();
            System.out.print((char)ch);
            if(ch == 10){ //Если пришло подтверждение
                curtime = new Date().getTime();
                System.out.println((curtime - statime) + ": ");
                flag = false;
                flagRepeat = false;
            }
            if(ch == 13){ //Если пришла синхронизация
                //Заново отправляем пакет
                out.write(mes.getBytes());
                rTime = new Date().getTime() + dTime;
            }
        }else{ //Проверка тайм-аута
            System.out.print(".");
            if(new Date().getTime() > rTime){
                if(flagRepeat) flag = false;
                else{ //Инициация синхронизации
                    out.write("\r".getBytes());
                    rTime = new Date().getTime() + dTime;
                    flagRepeat = true;
                }
            }
        }
    }
    if(flagRepeat) break;
}
out.flush();
out.close();

```

```

        in.close();
        clientsocket.close();

    }catch(UnknownHostException ue)
    {
        System.out.println("\nUnknownHostException: "
            + ue.getMessage());
    }catch(IOException e)
    {
        System.out.println("\nIOException: " + e.getMessage());
    }catch(Exception ee)
    {
        System.out.println("\nException: " + ee.getMessage());

        //Посказка для запуска программы
        System.out.println("\nrun: java asu.client.TCPClient address "
            + "port time_out count_message message\n");
    }

    System.out.println("\nTCPClient stop: " + new Date());
    System.exit(0);
}
}

```

Общий формат команды запуска программы-*клиента* из командной строки имеет вид:

```
$ java asu.client.TCPClient address port time_out count_message message
```

При исполнении команды запуска клиента следует учитывать адрес и номер порта запущенного сервера. При учёте данных, отображённых на рисунке 2.18, аргументы могут быть следующие:

- **address** = 78.140.20.13
- **port** = 8888
- **time_out** = 1000
- **count_message** = 100
- **message** = "Text message from client program"

При таких аргументах, команда запуска программы-*клиента* будет иметь вид:

```
$ java asu.client.TCPClient 78.140.20.13 8888 1000 \
    "Text message from client program"
```

Обе программы должны быть запущены в разных терминалах, а результат должен быть следующий:

- программа-*клиента* — последовательно отправит серверу 100 одинаковых сообщений и распечатает их на своём терминале;

- программа-**сервера** примет эти сообщения и распечатает их на своём терминале;
- по завершении работы, программа-**клиента** выведет на терминал общее время передачи всех сообщений.

2.6 Организация доступа к базам данных

Классическим примером распределенных систем является совокупность приложений, обрабатывающих данные в некоторой вычислительной сети. Принципиальным здесь является тот факт, что система разделена как минимум на две части, которые как программное обеспечение функционируют автономно, но связаны некоторым набором общих данных, которые они вместе обрабатывают. Сама обработка данных осуществляется по технологии «клиент-сервер», где:

- программа-**клиент** — приложение, выполняющее запросы к серверу;
- программа-**сервер** — СУБД, обслуживающее запросы клиента.

Познавательная цель данного подраздела — описание классов и методов языка Java, сосредоточенных в пакете *java.sql* и организующих общий доступ приложений к СУБД. Технология организации такого доступа основана на выполнении следующих четырёх этапов:

1. Загрузка драйвера необходимой СУБД.
2. Установка соединения между Java-программой и СУБД.
3. Передача в базу данных команд языка SQL.
4. Получение и обработка результатов таких команд (SQL-запросов).

Организация учебного материала проведена с учётом того, что студент уже освоил теорию и практику работы с реляционными СУБД в дисциплине «Базы данных» и способен формулировать правильные запросы на языке SQL. Это даёт основание сосредоточить основное внимание на инструментальных средствах пакета *java.sql* и детализировать изложение материала на примере конкретной СУБД — *Apache Derby* [31]. Такой подход закладывает познавательный фундамент для изучения материала последующих глав. Само изложение текущего учебного материала разделено на три пункта:

- описание инструментальных средств СУБД Apache Derby;
- описание классов и методов для формирования SQL-запросов к СУБД и подключения необходимых драйверов, обеспечивающих подключение Java-программ к конкретным СУБД;
- демонстрация Java-программы, реализующей простейший типовой пример работы с СУБД.

2.6.1 Инструментальные средства СУБД Apache Derby

Проект *Apache Derby*, появившийся в 1997 году, является примером СУБД полностью написанным на языке Java. Как отмечает Википедия [32]: «**Apache Derby** — реляционная СУБД, написанная на Java, предназначенная для встраивания в Java-приложения или обработки транзакций в реальном времени. Занимает 2 МБ на диске. Распространяется на условиях лицензии Apache 2.0. Ранее известна как **IBM Cloudscape**. Oracle распространяет те же бинарные файлы под именем **Java DB**», но в действительности, эта СУБД может работать как нормальный сетевой сервер, а размер ее зависит как от версии разработки, так и варианта установленного дистрибутива. Например, минимальный вариант, используемый в данной дисциплине, имеет размер 5 МБ. Официальный сайт проекта [33] предоставляет множество версий СУБД, которые обновляются регулярно раз в год и позволяют выбрать нужную конфигурацию, рабочая часть которой пригодна для установки как в среде Linux, так и в среде MS Windows. Сама процедура установки — достаточно проста и предполагает:

- выбор корневого каталога для установки СУБД Derby и копирования в него содержимого архива дистрибутива;
- настройку переменных среды ОС, учитывающих местоположение как самой СУБД, так и среды исполнения Java (JRE).

Для студентов, изучающих данную дисциплину, дистрибутив СУБД Derby установлен в архиве его личной рабочей области, которая подключается к среде учебного программного комплекса (УПК) [6] во время проведения им лабораторных работ. В проекции такого варианта и продолжим дальнейшее изложение учебного материала.

СУБД Derby инсталлировано в среду УПК следующим образом:

- **\$HOME/derby** — корневой каталог установленной СУБД;
- **\$HOME/derby/bin** — каталог размещения баз данных, утилит и сценариев запуска различных компонент СУБД;
- **\$HOME/derby/lib** — каталог размещения библиотек СУБД.

Для правильной настройки среды ОС необходимо правильно настроить четыре переменных среды в файле **\$HOME/.bashrc**, как показано на рисунке 2.19.

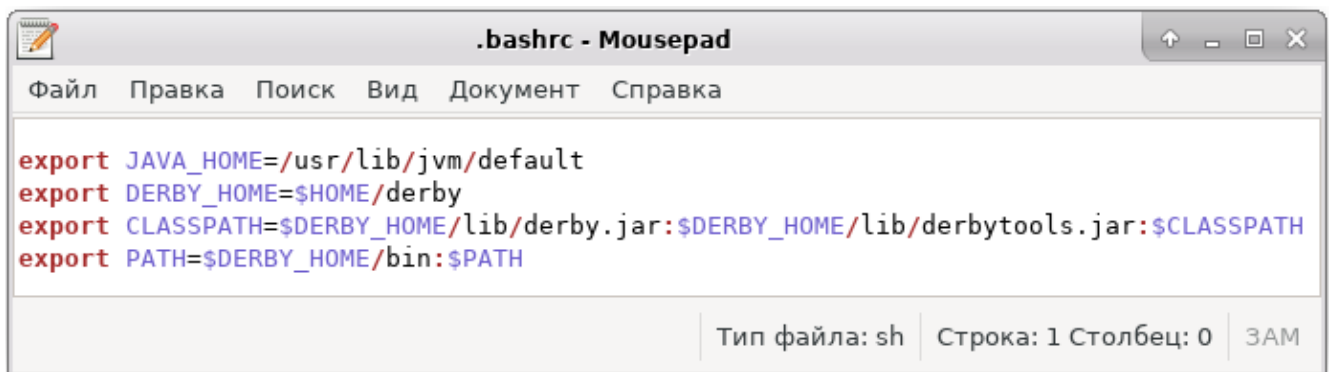


Рисунок 2.19 — Правильная настройка переменных среды ОС

Для проверки произведённых настроек, необходимо запустить новый терминал и выполнить команду:

```
$ java org.apache.derby.tools.sysinfo
```

Если настройки выполнены правильно, то на терминал будут выведены сообщения об используемых версиях дистрибутивов Java и Derby, показанные на рисунке 2.20.

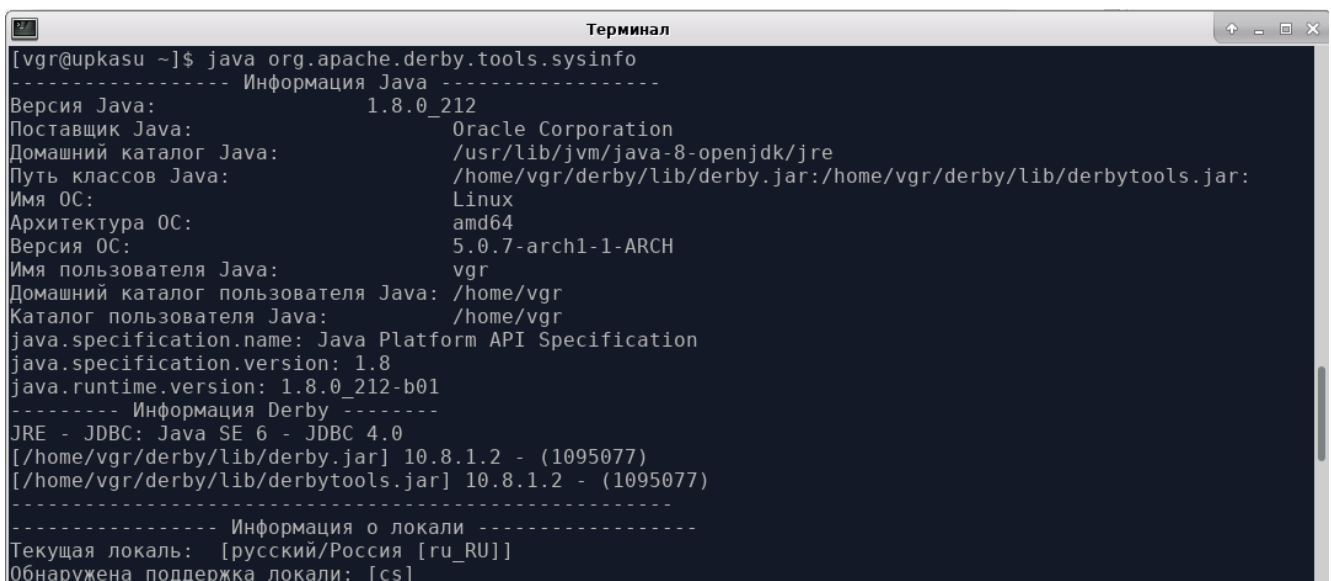


Рисунок 2.20 — Сообщение ПО Derby при правильной настройке среды ОС

Если СУБД Derby используется в серверном варианте, необходимо определиться с адресом и портом, по которым СУБД будет принимать соединения. Общий формат использования сценария, стартующего сервер, имеет вид:

```
$ $DERBY_HOME/bin/startNetworkServer -h адрес -p порт &
```

По умолчанию (без аргументов), указанный сценарий будет запускать сервер по адресу *localhost* и порту *1527*. Чтобы служба безопасности разрешила такой запуск для нашего дистрибутива Java, необходимо добавить следующий оператор в файл */etc/java-8-openjdk/security/java.policy*:

```
grant {  
  permission java.net.SocketPermission "localhost:1527", "listen";  
};
```

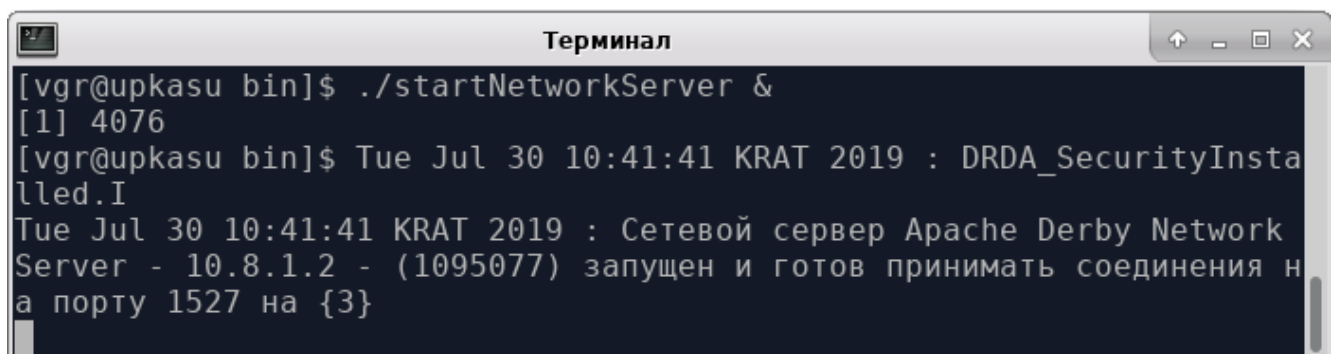
После внесённых изменений, команда:

```
$ $DERBY_HOME/bin/startNetworkServer &
```

запустит сервер, как это показано на рисунке 2.21, а запуск сценария:

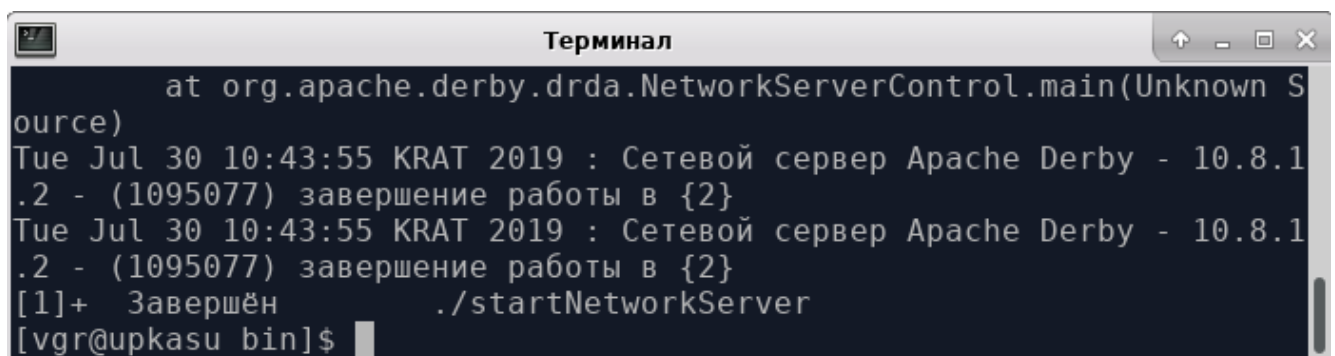
```
$ $DERBY_HOME/bin/stopNetworkServer
```

остановит сервер, показано на рисунке 2.22.



```
Терминал  
[vgr@upkasu bin]$ ./startNetworkServer &  
[1] 4076  
[vgr@upkasu bin]$ Tue Jul 30 10:41:41 KRAT 2019 : DRDA_SecurityInsta  
lled.I  
Tue Jul 30 10:41:41 KRAT 2019 : Сетевой сервер Apache Derby Network  
Server - 10.8.1.2 - (1095077) запущен и готов принимать соединения н  
а порту 1527 на {3}
```

Рисунок 2.21 — Локальный старт сервера Apache Derby

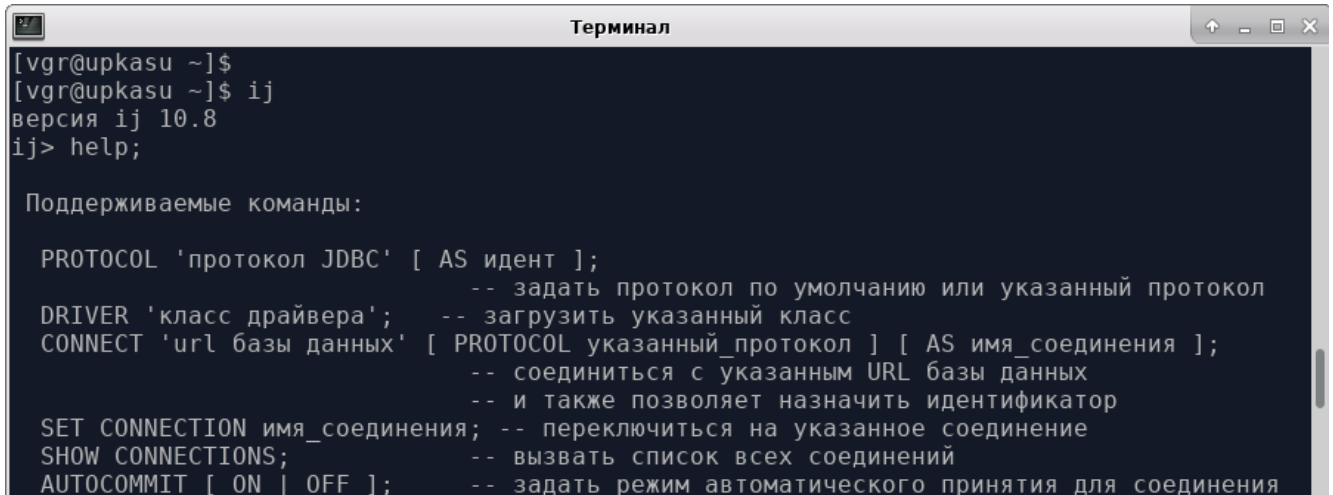


```
Терминал  
at org.apache.derby.drda.NetworkServerControl.main(Unknown S  
ource)  
Tue Jul 30 10:43:55 KRAT 2019 : Сетевой сервер Apache Derby - 10.8.1  
.2 - (1095077) завершение работы в {2}  
Tue Jul 30 10:43:55 KRAT 2019 : Сетевой сервер Apache Derby - 10.8.1  
.2 - (1095077) завершение работы в {2}  
[1]+  Завершён ./startNetworkServer  
[vgr@upkasu bin]$
```

Рисунок 2.22 — Завершение работы сервера Apache Derby

Проведённых настроек уже вполне достаточно для полноценной рабо-
ты с СУБД Derby, которая обеспечивается интерактивной утилитой *ij*. Она

позволяет создавать и удалять базы данных, делать к ним различные SQL-запросы и выводить результаты на стандартный ввод-вывод. На рисунке 2.23 показан пример запуска *ij* в интерактивном режиме и выполнение в ней команды *help*, которая выводит список доступных инструментальных средств этой утилиты.



```
[vgr@upkasu ~]$  
[vgr@upkasu ~]$ ij  
версия ij 10.8  
ij> help;  
  
Поддерживаемые команды:  
  
PROTOCOL 'протокол JDBC' [ AS идент ];  
-- задать протокол по умолчанию или указанный протокол  
DRIVER 'класс драйвера'; -- загрузить указанный класс  
CONNECT 'url базы данных' [ PROTOCOL указанный_протокол ] [ AS имя_соединения ];  
-- соединиться с указанным URL базы данных  
-- и также позволяет назначить идентификатор  
SET CONNECTION имя_соединения; -- переключиться на указанное соединение  
SHOW CONNECTIONS; -- вызвать список всех соединений  
AUTOCOMMIT [ ON | OFF ]; -- задать режим автоматического принятия для соединения
```

Рисунок 2.23 — Выполнение команды *help* в интерактивном режиме утилиты *ij*

Самой утилитой мы воспользуемся в пункте 2.6.3, а сейчас перейдём к изучению классов и методов пакета *java.sql*.

2.6.2 SQL-запросы и драйверы баз данных

Для взаимодействия любой СУБД в языке Java предусмотрены специальные средства, называемые **JDBC**. Википедия так характеризует эти средства [34]: «**JDBC** (англ. Java DataBase Connectivity — *соединение с базами данных на Java*) — платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета *java.sql*, входящего в состав Java SE. JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает».

Непосредственно в пакете *java.sql* реализована только группа интерфейсов под общим названием **Driver**, а само ПО драйверов поставляется в дистрибутивах соответствующих СУБД или ищутся на сайтах вендоров, например, перечисленных в таблице 2.5.

Таблица 2.5 — Примеры официальных сайтов вендоров СУБД

Название СУБД	Адрес сайта
Derby DB (clients) Derby DB (embedded)	http://db.apache.org/derby/derby_downloads.html
MySQL	http://www.mysql.com/downloads/connector/j/
PostgreSQL	http://jdbc.postgresql.org/download.html
Microsoft SQLServer	http://go.microsoft.com/fwlink/?LinkId=245496
Oracle Database	http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html
HSQLDB	http://sourceforge.net/projects/hsqldb/files/hsqldb/
SQLite	http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC
Sybase Adaptive Server Enterprise	http://sourceforge.net/projects/jtds/files/
Sybase Adaptive Server IQ	http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect
JDBC/ODBC	Интегрирован в JDK

Что касается СУБД Derby, то нужные драйвера находятся в библиотеках:

- ***derby.jar*** — содержит драйвер для встроенного подключения к СУБД;
- ***derbyclient.jar*** — содержит драйвер для удалённого (сетевое) подключения к серверу СУБД.

Список имён классов, реализующих интерфейс ***Driver***, для разных СУБД представлен в таблице 2.6.

Таблица 2.6 - Имена классов драйверов для некоторых СУБД

Название СУБД	Класс драйвера
Derby DB (for remote clients) Derby DB (embedded)	<i>org.apache.derby.jdbc.ClientDriver</i> <i>org.apache.derby.jdbc.EmbeddedDriver</i>
MySQL	<i>com.mysql.jdbc.Driver</i>
PostgreSQL	<i>org.postgresql.Driver</i>
Microsoft SQLServer	<i>com.microsoft.jdbc.sqlserver.SQLServerDriver</i>
Oracle Database	<i>oracle.jdbc.driver.OracleDriver</i>
HSQLDB	<i>org.hsqldb.jdbc.JDBCDriver</i>
SQLite	<i>org.sqlite.JDBC</i>

Sybase Adaptive Server Enterprise	<i>net.sourceforge.jtds.jdbc.Driver</i>
Sybase Adaptive Server IQ	<i>com.sybase.jdbc3.jdbc.SybDriver</i>
JDBC/ODBC	<i>sun.jdbc.odbc.JdbcOdbcDriver</i>

Непосредственное подключение драйвера к коду Java-программы выполняется в формате шаблона:

```
try {
    Class.forName("your_driver_class_name");
} catch (ClassNotFoundException ex) {
    System.out.println(ex.getMessage());
}
```

Для манипулирования драйверами и подключения к СУБД, пакет *java.sql* содержит класс ***DriverManager*** со статическими методами:

<i>registerDriver(Driver driver)</i>	Регистрирует драйвер.
<i>deregisterDriver(Driver driver)</i>	Удаляет драйвер.
<i>getDrivers(void)</i>	Возвращает перечисление Enumeration<Driver>.
<i>getConnection(String jurl)</i>	Возвращает объект класса Connection.
<i>getConnection(String jurl, String user, String password)</i>	Возвращает объект класса Connection.

где ***jurl*** — строка, определяющая абсолютный адрес для JDBC, построенный по следующему общему принципу:

jdbc:название_набора_драйверов:имя_и_путь_к_базе_данных

Для каждой СУБД, строка ***jurl*** имеет свой уникальный формат, определённый вендором драйвера. В частности, она может включать аргументы ***user*** и ***password***, которые соответствуют имени пользователя и его пароля, объявленные при создании базы данных. Формат этой строки для СУБД Derby будет рассмотрен в следующем пункте.

Остальные классы и методы пакета *java.sql* не зависят от используемого драйвера или адреса JDBC. Так, объекты класса ***Connection*** имеют два основных метода связанных с формированием SQL-запросов:

- ***createStatement()*** - возвращает объект класса ***Statement***, методы которого отправляют полностью сформированный SQL-запрос;
- ***prepareStatement(String sql)*** - создаёт объект класса ***PreparedStatement***, содержащий не полностью сформированный SQL-запрос.

Объекты класса *Statement* предлагают два метода:

- *executeQuery(String sql)* — отправляет СУБД запрос, полностью сформированной, на основе оператора **SELECT**, строкой *sql*; возвращает объект класса *ResultSet*;
- *executeUpdate(String sql)* — отправляет СУБД запрос, полностью сформированной, на основе оператора **DELETE** или **INSERT** или **UPDATE**, строкой *sql*; возвращает целое число изменённых строк.

Объекты класса *PreparedStatement* сформированы на не полностью определённой строке SQL-запроса, где неизвестные параметры помечаются символом «?». Их методы разделены на две группы:

- первая группа — методы *setInt(int n, int i)*, *setString(int n, String s)* и другие, где первый аргумент задаёт номер изменяемого параметра, а второй — само значение параметра, согласно его типу;
- вторая группа — методы *executeQuery()* и *executeUpdate()*, аналогичные классу *Statement*.

Объекты класса *ResultSet* содержит упакованный по строкам результат возврата SQL-запроса **SELECT**. Имеют три группы методов:

- первая группа — метод *getMetaData()* возвращает только объект класса *ResultSetMetaData*;
- вторая группа — *first()*, *last()*, *previous()* и *next()* обеспечивают перемещение по строкам результата запроса;
- третья группа — *getBytes(...)*, *getInt(...)*, *getString(...)* и другие, обеспечивают извлечение из объекта запроса типизированных данных; в качестве аргумента этих методов указывается номер столбца или его имя.

Объект класса *ResultSetMetaData* содержит метаданные результата запроса, которые извлекаются методами не требующими пояснений: *getColumnCount()*, *getColumnName()*, *getColumnType()* и *isReadOnly(int column)*.

2.6.3 Типовой пример выборки данных

Завершив в предыдущем пункте общее описание интерфейсов, классов и методов пакета *java.sql*, перейдём к рассмотрению конкретного примера, который наглядно покажет как использовать эти инструментальные средства пакета. Для этого возьмём задачу ведения личных записей в таблице с именем *notepad*, хранящейся в базе данных с именем *exampleDB*. Чтобы не путаться в альтернативных вариантах, конкретизируем условия задачи следующими ограничениями:

- будем использовать *встроенный вариант* использования базы данных, который нам потребуется и в других примерах, предполагающий размещение ее в домашней директории пользователя в каталоге: ***\$HOME/databases***;
- таблица ***notepad*** имеет два поля, первое из которых является целочисленным уникальным ключом с именем ***notekey***, а второе — поле переменной длины с именем ***text***, содержащее отдельную запись пользователя;
- доступ прикладных программ к базе данных ***exampleDB*** осуществляется от имени пользователя ***upk*** с паролем ***upkasu***.

Приступая к реализации приложений работы с базой данных ***notepad***, конкретизируем содержание строки ***jurl***, определяющей абсолютный адрес JDBC для доступа к ней. В условиях поставленной задачи, полная строка для *встроенного варианта* имеет вид:

```
jdbc:derby:/home/vgr/databases/exampleDB;create=true;user=upk;password=upkasu;
```

где опции ***create***, ***user*** и ***password*** могут отсутствовать в зависимости от вариантов их использования. Например, опция ***create=true*** означает, что база данных будет создана, если она — отсутствует.

Соответственно, *альтернативный вариант* для серверного использования базы данных, когда она будет создана в каталоге ***\$DERBY_HOME/derby/bin***, имеет вид:

```
jdbc:derby://localhost:1527/exampleDB;create=true;user=upk;password=upkasu;
```

Разобравшись с ***jurl*** JDBC, создадим базу данных ***exampleDB***, которая обычно должна существовать перед запуском и отладкой прикладных программ, работающих с ней. А для этого, воспользуемся служебным инструментом СУБД представленным в виде утилиты ***ij***.

Общая технология разработки баз данных предполагает создание порождающего сценария, реализующего некоторый первоначальный ее вариант. Для нашего примера задачи такой сценарий представлен на рисунке 2.24, отображающего содержимое файла ***createDB.sql***, размещённого в каталоге ***\$HOME/src/rvs/*** и запускаемого из него командой:

```
$ ij createDB.sql
```



```
-----
-- Сценарий создания базы данных: exampleDB.
-- Создается таблица notepad — личные записи студента.
-----
-- Вариант сервера:
-- CONNECT 'jdbc:derby://localhost:1527/exampleDB;create=true;'
--     USER 'upk' PASSWORD 'upkasu';
-----
-- Вариант встроенной БД:
CONNECT 'jdbc:derby:/home/vgr/databases/exampleDB;create=true;'
    USER 'upk' PASSWORD 'upkasu';
-----
-- Сначала удаляем все таблицы, если они были созданы.
drop table notepad;
-----
-- Создаем таблицу notepad

create table notepad (
    notekey int not null,    -- ключ записи - ключ таблицы
    text varchar(50),        -- текст записи
    primary key (notekey)
);
-----
-- Вносим в таблицу некоторое количество записей:
insert into notepad values(321, 'Запись 1');
insert into notepad values(743, 'Запись 2');
-----
select * from notepad;
-----
-- Завершение работы сценария:
commit;
disconnect;
exit;
-----
| Тип файла: SQL | Строка: 34 Столбец: 58 | ЗАМ
```

Рисунок 2.24 — Сценарий создания первоначального варианта базы данных ExampleDB

Результат запуска этого сценария, создающего *встроенный вариант* базы данных, представлен на рисунке 2.25. Если запустить сетевой вариант СУБД и изменить в сценарии строку соединения, то такая же база данных будет создана в директории *\$HOME/derby/bin/* и обсуживать сетевые приложения.

Что касается приложений, работающих с базой данных *exampleDB*, то они с успехом могут быть реализованы в виде набора *SQL*-сценариев и функций любого языка *shell*, использующих нужные запуски утилиты *ij*.

```
Терминал
[vgr@upkasu rvs]$ ij createDB.sql
версия ij 10.8
ij> -----
-- Сценарий создания базы данных: exampleDB.
-- Создается таблица notepad – личные записи студента.
-----
-- Вариант сервера:
-- CONNECT 'jdbc:derby://localhost:1527/exampleDB;create=true;'
-- USER 'upk' PASSWORD 'upkasu';
-----
-- Вариант встроенной БД:
CONNECT 'jdbc:derby:/home/vgr/databases/exampleDB;create=true;'
USER 'upk' PASSWORD 'upkasu';
ij> -----
-- Сначала удаляем все таблицы, если они были созданы.
drop table notepad;
ОШИБКА 42Y07: Схема 'UPK' не существует
ij> -----
-- Создаем таблицу notepad

create table notepad (
    notekey int not null,    -- ключ записи - ключ таблицы
    text varchar(50),        -- текст записи
    primary key (notekey)
);
Вставлено/обновлено/удалено строк: 0
ij> -----
-- Вносим в таблицу некоторое количество записей:
insert into notepad values(321, 'Запись 1');
Вставлена/обновлена/удалена 1 строка
ij> insert into notepad values(743, 'Запись 2');
Вставлена/обновлена/удалена 1 строка
ij> -----
select * from notepad;
NOTEKEY    |TEXT
-----
321         |Запись 1
743         |Запись 2

Выбрано строк: 2
ij> -----
-- Завершение работы сценария:
commit;
ij> disconnect;
ij> exit;
[vgr@upkasu rvs]$
```

Рисунок 2.25 — Результат запуска сценария createDB.sql

Для начального изучения практики применения классов и методов пакета *java.sql*, рассмотрим пример интерактивной программы, которая уста-

навливает соединение со встроенным вариантом БД *exampleDB* и в цикле выполняет следующие действия;

- читает все записи таблицы *notepad* и печатает их на стандартный вывод;
- последовательно запрашивает у пользователя содержимое полей *notekey* и *text*, формируя к базе данных SQL-запрос на вставку новой записи;
- в случае любого исключения или, если пользователь ввёл в поле *notekey*, строку нулевой длины, то программа завершает свою работу.

Описанная программа реализована в виде класса *Example11*, а ее исходный текст представлен на листинге 2.15. Следует обратить внимание на стиль ее написания, предполагающий выделение отдельных специализированных методов и обработку в каждом из них возникающих исключений:

- метод конструктора *Example11(...)* обеспечивает подключение драйвера и создание соединения с БД;
- метод *getResultSet()* читает содержимое таблицы *notepad*;
- метод *setInsert()* записывает в таблицу *notepad* новую строку;
- специализированные методы *getKey()* и *getString()* ориентированы на чтение со стандартного ввода;
- метод *setClose()* закрывает объекты созданные конструктором;
- статический метод *main(...)* создаёт объект класса *Example11* и, с помощью других методов этого объекта, организует интерактивный интерфейс с пользователем.

Листинг 2.15 — Исходный текст класса *Example11* из среды *Eclipse EE*

```
package ru.tusur.asu;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Example11 {

    // Объекты класса
    boolean flag = true;
    Connection conn;
    ResultSet rs;

    // Конструктор
    Example11(String dburl, String dbuser, String dbpassword)
    {
```

```

    try {
        //Подключаем необходимый драйвер
        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

        //Устанавливаем соединение с БД
        conn = DriverManager.getConnection(dburl,
            dbuser, dbpassword);

    } catch (ClassNotFoundException e1){
        System.out.println(e1.getMessage());
        flag = false;
    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        flag = false;
    }
}

// Метод, реализующий SELECT
public int getResultSet()
{
    String sql1 = "SELECT * FROM notepad ORDER BY notekey";

    try
    {
        Statement st =
            conn.createStatement();
        rs = st.executeQuery(sql1);
        return 1;

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        return 0;
    }
}

// Метод, реализующий INSERT
public int setInsert(int key, String str)
{
    String sql2 = "INSERT INTO notepad values( ? , ? )";

    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql2);

        // Установка первого параметра
        pst.setInt(1, key);
        // Установка второго параметра
        pst.setString(2, str);

        return pst.executeUpdate();

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        return -1;
    }
}

```

```

// Метод чтения целого числа со стандартного ввода
public int getKey()
{
    int ch1 = '0';
    int ch2 = '9';
    int ch;
    String s = "";

    try
    {
        while(System.in.available() == 0) ;

        while(System.in.available() > 0)
        {
            ch = System.in.read();
            if (ch == 13 || ch < ch1 || ch > ch2)
                continue;
            if (ch == 10)
                break;

            s += (char)ch;
        };
        if (s.length() <= 0)
            return -1;
        ch = new Integer(s).intValue();
        return ch;
    } catch (IOException e1){
        System.out.println(e1.getMessage());
        return -1;
    }
}

// Метод чтения строки текста со стандартного ввода
public String getString()
{
    String s = "\r\n";
    String text = "";
    int n;
    char ch;
    byte b[];

    try
    {
        //Ожидаем поток ввода
        while(System.in.available() == 0) ;
        s = "";
        while((n = System.in.available()) > 0)
        {
            b = new byte[n];
            System.in.read(b);
            s += new String(b);
        };
        // Удаляем последние символы '\n' и '\r'
        n = s.length();
        while (n > 0)
        {
            ch = s.charAt(n-1);

```



```

        if (ch == '\n' || ch == '\r')
            n--;
        else
            break;
    }
    // Выделяем подстроку
    if (n > 0)
        text = s.substring(0, n);
    else
        text = "";
    return text;

} catch (IOException e1){
    System.out.println(e1.getMessage());
    return "Ошибка...";
}
}
// Закрытие соединения
public void setClose()
{
    try
    {
        rs.close();
        conn.commit();
        conn.close();

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
    }
}

public static void main(String[] args)
{
    System.out.println("Программа ведения записей в БД exampleDB.\n"
        + "Используются методы класса Example11\n"
        + "-----");

    // Исходные данные
    String url = "jdbc:derby:/home/vgr/databases/exampleDB";
    String user = "upk";
    String password = "upkasu";

    // Создаем объект класса
    Example11 obj =
        new Example11(url, user, password);
    if (!obj.flag)
    {
        System.out.println(
            "Не могу создать объект класса Example11...");
        System.exit(1);
    }

    int ns;           // Число прочитанных строк
    int nb;           // Число прочитанных байт
    String text, s;   // Строка введенного текста

    // Цикл обработки запросов
    while(obj.flag)

```

```

{
    //Печатаем заголовок ответа
    System.out.println("        Ключ    Текст\n"
        + "-----");
    obj.getResultSet();

    if (obj.rs == null)
    {
        System.out.println("Отсутствует результат SELECT...");
        break;
    }
    ns = 0;
    try
    {
        //Выводим (построчно) результат запроса к БД
        while(obj.rs.next()){
            System.out.format("%10d", obj.rs.getInt(1));
            System.out.println("    " + obj.rs.getString(2));
            ns++;
        }
        // Выводим итог запроса
        System.out.println("-----\n"
            + "Прочитано " + ns + " строк\n"
            + "-----\n"
            + "Формируем новый запрос!");

        System.out.print("\nВведи ключ или Enter: ");
        nb = obj.getKey();

        if (nb == -1)
            break;          // Завершаем работу программы

        System.out.print("Строка текста или Enter: ");
        s = obj.getString();
        text = s;

        while (s.length() > 0)
        {
            System.out.print("Строка текста или Enter: ");
            text += ("\n" + s);
            s = obj.getString();
        }

        if (text.length() <= 0)
            text = "Нет данных...";

        ns = obj.setInsert(nb, text);
        if (ns == -1)
            System.out.println("\nОшибка добавления строки !!!");
        else
            System.out.println("\nДобавлено " + ns + " строк...");

    } catch (SQLException e1){
        System.out.println(e1.getMessage());
        break;
    }
}

```

```
        //Закрываем все объекты и разрываем соединение
        //obj.setClose();
        System.out.println("Программа завершила работу...");
    }
}
```

Демонстрация приведённой программы — это занятие для лабораторной работы, но отметим, что ее идейная основа ложится в учебный материал последующих глав.

На этом завершается учебный материал главы 2, посвящённый инструментальным средствам языка Java.

Вопросы для самопроверки

1. Назовите две стандартные поставки инструментальных средств Java.
2. В чем состоит основное отличие языка Java от языков C/C++?
3. На какие четыре платформы подразделяются дистрибутивы языка Java?
4. Какие переменные среды ОС необходимо настроить для правильной работы инструментальных средств Java?
5. Что такое - пакетная организация ПО Java?
6. Сколько простых типов данных имеет язык Java и чем они отличаются от объектов?
7. Чем интерфейсы отличаются от классов?
8. Перечислите объекты стандартного ввода-вывода языка Java.
9. Перечислите базовые классы языка Java, обеспечивающие его ввод-вывод.
10. Какие классы сетевой адресации использует Java и какому пакету они принадлежат?
11. Для чего предназначены классы *InputStream* и *OutputStream* языка Java?
12. Что такое — сокет языка Java и какими классами они поддерживаются?
13. Что такое — синхронное и асинхронное взаимодействие и какие классы пакета *java.net* их поддерживают?
14. В чем состоит отличие между классами *Socket* и *ServerSocket* пакета *java.net*?
15. В каком пакете языка Java сосредоточены базовые средства доступа к базам данных?
16. Какая СУБД полностью написана на языке Java?
17. Какие переменные среды ОС необходимо настроить для правильной работы с СУБД Apache Derby?
18. Что такое адресация JDBC и для чего она используется?
19. Что такое драйверы баз данных и где их взять?
20. В чем состоят особенности встроенного использования СУБД Derby?