МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Практические занятия

Учебно-методическое пособие

направление подготовки: **09.03.01 — Информатика и вычислительная техника**
направленность (профиль) программы: **Программное обеспечение средств вычислительной техники и автоматизированных систем**

Разработчик
доцент кафедры АСУ

В.Г. Резник

Томск
2021

Резник В.Г.

Распределенные вычислительные системы. Практические занятия по направлению подготовки бакалавриата 09.03.01. Учебно-методическое пособие. – Томск, ТУСУР, 2021. – 95 с.

Учебно-методическое пособие предназначено для проведения практических занятий по дисциплинам «Вычислительные системы и сети» и «Распределенные вычислительные системы» для студентов направления подготовки бакалавра: 09.03.01 «Информатика и вычислительная техника» направленности (профиля) программы - «Программное обеспечение средств вычислительной техники и автоматизированных систем».

Оглавление

Введение....4
1 Инструментальные средства языка Java....5
1.1 Практическое занятие №1 — лабораторная №2. Базовый синтаксис языка Java....6
1.1.1 Стандартный вывод результатов работы программ....6
1.1.2 Использование даты и времени....9
1.1.3 Особенности использования метода main(...)....10
1.1.4 Преобразования простых типов данных....11
1.1.5 Использование массивов....13
1.1.6 Работа со строками....14
1.1.7 Управляющие операторы....16
1.2 Практическое занятие №2 — лабораторная №3. Организация ввода-вывода в объектной модели Java....17
1.2.1 Файлы и каталоги....17
1.2.2 Сериализация объектов....20
1.2.3 Символьные потоки ввода-вывода....23
1.3 Практическое занятие №3 — лабораторная №4. Сети и многопоточное программирование....27
1.3.1 Использование интерфейса Runnable....27
1.3.2 Синхронизация многопоточного приложения....30
1.4 Практическое занятие №4 — лабораторная №4. Сокеты языка Java....35
1.5 Практическое занятие №5 — лабораторная №5. SQL-запросы к базам данных....43
1.5.1 Числовые типы данных....43
1.5.2 Строковые типы данных....44
1.5.3 Типы даты и времени....45
1.5.4 Специальные типы данных....47
1.5.5 Функции....51
2 Практическое занятие №6 — лабораторная №6. Объектные распределенные системы....53
2.1 Инструментальные средства языка Java для технологии RMI....53
2.1.1 Утилита rmic....54
2.1.2 Преобразование интерфейсов RMI в описание IDL CORBA....58
3 Web-технологии распределенных систем....60
3.1 Практическое занятие №7 — лабораторная №7. Классы технологии Java-сервлетов....60
3.1.1 Общая обработка запроса....61
3.1.2 Обработка контекста запроса....67
3.2 Практическое занятие №8 — лабораторная №8. HTML и технология JSP-страниц....71
3.2.1 Установка кодировки символов объекта response....72
3.2.2 Передача атрибутов JSP-страницам....76
3.3 Практическое задание №9 — лабораторная №9. Шаблон проектирования MVC....81
3.3.1 Проектирование элементов шаблона MVC....81
3.3.2 Реализация проекта webpad....90
Список использованных источников....95

Введение

Данное пособие содержит учебно-методический материал для проведения практических занятий по дисциплинам «Распределенные вычислительные системы» и «Вычислительные системы и сети», предполагающий уровень подготовки бакалавриата. Базовым учебником для этих дисциплин является источник [1], содержащий весь необходимый теоретический материал, а также примеры на языке Java, который рассматривается как основной язык программирования в пределах изучаемых дисциплин. При этом предполагается, что студент ранее не изучал данный язык, но прошёл успешное обучение по изучению языков C/C++.

Цель данного учебно-методического пособия — теоретическое освоение тех элементов языка Java, которые используются при выполнении лабораторных работ по данной дисциплине в объёме 36 академических часов (девять лабораторных работ).

Основная задача данного пособия — практическое закрепление элементов синтаксиса и семантики языка Java, которые описаны и демонстрируются в учебном пособии [1], но не изложены в нужной степени подробностей для самостоятельного выполнения лабораторных работ по причине сохранения целостности изложения предметного материала. В силу того, что сам объем практических занятий также ограничен (18 академических часов, - девять практических занятий), в тексте пособия используются ссылки на полноценный источник по языку Java [2].

В целом, содержание данного пособия привязано к трём разделам базового учебного пособия [1] :

- Инструментальные средства языка Java (раздел 2).
- Объектные распределенные системы (раздел 3).
- Web-технологии распределенных систем (раздел 4).

В указанных разделах выделены подразделы, каждый из которых соответствует отдельному практическому занятию с текстом, разделенным на пункты выполняемых упражнений. Сами тексты упражнений не привязаны к каким-либо инструментальным средствам поддержки языка Java, но с методической точки зрения такие средства необходимо использовать, чтобы обеспечить дополнительный автоматизированный контроль синтаксиса изучаемых языковых конструкций Java. Для этих целей, практические занятия проводятся в специализированных учебных классах кафедры АСУ, в среде программного комплекса ОС УПК АСУ [3].

Таким образом, данное учебно-методическое пособие является практическим дополнением учебного пособия [1] и привязано к его тематике, а, с другой стороны, является теоретическим материалом, обеспечивающим успешное выполнение лабораторных работ по заявленным дисциплинам.

1 Инструментальные средства языка Java

Данная глава посвящена краткому изучению базовых средств языка Java, входящих в стандартный пакет J2SE и охватывающих технологии применимые во всех сферах его применения.

В основном учебнике [1] этой тематике посвящена глава 2, которая специально введена в дисциплину для изучения основ языка Java для студентов, которые прошли обучение по языкам C/C++, знакомы с теорией и практикой объектно-ориентированного программирования, но не изучали этот язык непосредственно. В силу указанных обстоятельств, данная глава является своеобразным экспресс курсом по синтаксису и семантике языка Java, что в теоретическом плане охватывает следующие вопросы:

- Пакетная организация языка Java.
- Выражения, переменные и простые типы данных.
- Операторы, операнды и управляющие операторы.
- Потоки ввода-вывода.
- Управление сетевыми соединениями.
- Организация доступа к базам данных.

Для изучения перечисленных вопросов выделяется **5** практических занятий с суммарным временем их проведения — **10** академических часов. Столько же времени отводится и для самостоятельной подготовки к ним.

С методической точки зрения считается, что студент проводит практические занятия, следуя материалу учебника [1] и выполняя представленные в нем примеры. Дополнительно считается, что студент:

- прочитал подраздел 2.1 главы 2 и имеет представление о возможностях запуска программ в личной рабочей области обучающей среды ОС УПК АСУ;
- запустил среду разработки Eclipse EE, создал проект с именем ***proj1*** и выполнил первый пример по созданию и запуску на исполнение объекта класс ***Example1***.

С целью систематизации получаемых практических навыков, студенту следует придерживаться предложенной в учебнике [1] нумерации и содержанию рассматриваемых проектов, дополняя их выполнением упражнений данной главы. С другой стороны, в любом проекте студент может:

1. Использовать оператор ***package*** для выделения собственных примеров программ.
2. Создавать различные классы, содержащие статический метод ***main(...)*** и запускать их раздельно на выполнение.

Главное, что нужно учитывать при запуске отдельных программ в общем проекте, - *обязательно выделить вкладку с текстом запускаемой программы*!

1.1 Практическое занятие №1 — лабораторная №2. Базовый синтаксис языка Java

Данное практическое занятие посвящено базовому синтаксису языка Java. В целом, это охватывает достаточно объёмный материал. Например, в учебнике [2] этому вопросу посвящены главы 2 - 10, что в сумме составляет порядка 230 страниц текста. Этот материал используется в лабораторной работе №2.

В пределах двух часового объёма занятия невозможно освоить такой объем текста, поэтому основное внимание уделяется тем аспектам языка Java, которые не столько отличают его от языков C и C++, а часто используются в содержании данной дисциплины.

В целом, в пределах занятия студент должен освоить следующий список вопросов, дополнив его при необходимости сведениями из источника [2]:

а) стандартный вывод результатов работы программ;
б) особенности использования метода ***main(...)***;
в) преобразование простых типов данных;
г) использование массивов;
д) работа со строками;
е) управляющие операторы языка Java.

Дополнительно студент должен уметь пользоваться официальной документацией широко доступной в сети Интернет. Весь обсуждаемый здесь материал доступен по адресу: https://docs.oracle.com/javase/6/docs/api/java/lang/package-summary.html.

1.1.1 Стандартный вывод результатов работы программ

Все языки программирования имеют средства стандартного вывода, которые доступны сразу при запуске прикладных программ. Такой вывод необходим как для простейшего представления результатов вычислений, так и для отладки программного обеспечения. Естественно, что он используется и в учебных целях, что демонстрируется, например, проектом ***proj1*** (см. [1], листинг 2.3, стр. 45). Представленный ниже текст этого листинга демонстрирует прежде всего работоспособность инструментальных средств среды Eclipse EE и является традиционным началом любого процесса обучения программированию.

```
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());

      }
```

```
}
```

В общем случае, для вывода стандартных сообщений предусмотрено два канала, которые представлены двумя объектами:

- ***java.lang.System.out*** — для вывода нормальных сообщений, например, результатов расчётов программы;
- ***java.lang.System.err*** — для вывода ошибок.

> Каким из каналов и для каких целей будет пользоваться программист, — это решать ему самому. Главное, что оба уже созданных объекта (***out*** и ***err***) принадлежат одному классу ***java.io.PrintStream***, содержащему много методов.
>
> Описания этих методов можно найти в официальной документации по адресу: https://docs.oracle.com/javase/6/docs/api/java/io/PrintStream.html.

Обычно, для вывода сообщений используются методы ***ptintf(String str)*** и ***println(String str)***, последний из которых добавляет к строке символ перевода строки. Именно этот метод использован в показанном выше примере. Причём, обратите внимание, что этот метод делает неявное преобразование даты (оператор ***new Date()***) в строку.

Для форматированного вывода следует использовать метод (print format):

```
printf(String format, Object … args);
```

где:

- ***format*** — строка формата, как описано в синтаксисе (см. далее);
- ***args*** — аргументы, на которые ссылаются спецификаторы формата в строке формата; если аргументов больше, чем спецификаторов формата, дополнительные аргументы игнорируются; количество аргументов является переменным и может быть нулевым.

Синтаксис строки формата аналогичен синтаксису языков C/C++. Он содержит строку символов со вставками вида:

```
%[argument_index$][flags][width][.precision]conversion
```

где:

а) Необязательный элемент ***arguments_index*** - это десятичное целое число, обозначающее позицию аргумента в списке аргументов. На первый аргумент ссылаются «1$», на второй - «2$» и так далее.

б) Необязательный элемент ***flags*** — это набор символов, которые изменяют формат вывода. Набор допустимых флагов зависит от преобразования.

в) Необязательный элемент ***width*** — ширина представляет собой неотрицательное десятичное целое число, указывающее минимальное количество символов, которое будет записано в вывод.

г) Необязательная часть ***precision*** (точность) — неотрицательное десятичное целое число, обычно используемое для ограничения количества символов. Конкретное поведение зависит от преобразования.

д) Требуемое преобразование (***conversion***) — это символ, указывающий, как аргумент должен быть отформатирован. Набор допустимых преобразований для данного аргумента зависит от типа данных аргумента.

Символы преобразования — следующие:

а) ***b*** — если аргумент ***arg*** равен нулю, то результатом будет "false". Если ***arg*** - логическое значение, то результатом является строка, возвращаемая функцией ***String.valueOf()***; в противном случае результат будет «true»;
б) ***h*** — если аргумент ***arg*** равен нулю, то результат равен нулю; в противном случае результат получается путём вызова ***Integer.toHexString*** ();
в) ***s*** — если аргумент ***arg*** равен нулю, то результат равен нулю; если ***arg*** не равен пустой строке, результат получается вызовом ***arg.toString ()***;
г) ***c*** — результатом является символ Unicode;
д) ***d*** — результат форматируется как десятичное целое число;
е) ***o*** — результат форматируется как восьмеричное целое;
ж) ***x*** — результат форматируется как шестнадцатеричное целое;
з) ***e*** — результат форматируется как десятичное число в компьютеризированной научной нотации;
и) ***f*** — результат форматируется как десятичное число с плавающей точкой;
к) ***g*** — результат форматируется с использованием компьютеризированной научной нотации или десятичного формата, в зависимости от точности и значения после округления;
л) ***a*** — результат форматируется как шестнадцатеричное число с плавающей запятой со значениями и показателем степени;
м) ***t*** — префикс даты/времени для символов преобразования даты и времени; смотрите преобразование даты/времени;
н) ***%*** - результатом является литерал "%" ('\ u0025');
о) ***n*** — разделитель строк; результатом является специфичный для платформы разделитель строк.

Если в программе необходимо получить строку некоторого заданного формата, следует воспользоваться статическим методом ***format(...)*** объекта класса ***String***:

```
String str = String.format(String ss, Object … args);
```

Задание.

В проекте с именем ***proj1*** написать программу, обеспечивающую форматированный вывод сообщений на консоль терминала.

Результат программы оформить приблизительно так, как показано в шаблоне программы:

```
package rvs.pr1;

import java.io.PrintStream;

public class RvsOut {

	public static void main(String[] args) {
		/**
		 * Определение и задание начальных значений.
		 */
		int    ix = 10;
		float  fy = 111;
		double dy = 45.78;

		/**
		 * Вывод на печать.
		 */
		PrintStream msg = System.out;

		msg.printf("Вывод целого числа ix = %1$d; %1$o \n", ix);
		msg.printf("Вывод float  числа fy = %1$f; %1$g \n", fy);
		msg.printf("Вывод double числа dy = %1$f; %1$e \n", dy);
	}

}
```

1.1.2 Использование даты и времени

В приложениях часто приходится работать с датой и временем. В учебном пособии [1] используется только класс ***java.util.Date***, который был введён в первой версии языка Java и в настоящее время считается устаревшим. Тем не менее, он вполне приемлем, для относительного измерения времени в миллисекундах для чего используется метод ***getTime()***. Например:

```
PrintStream msg = System.out;

/**
 * Класс Date().
 */
Date dt = new Date();
long time = dt.getTime();
msg.printf("Число миллисекунд от 01.01.1970 = %d \n", time);
```

Для более продвинутого использования даты и времени следует обратиться к абстрактному классу ***java.util.Calendar*** или ***java.util.GregorianCalendar***. Для этого необходимо внимательно изучить главу 16 источника [2, стр. 482 - 493].

В простейших случаях можно воспользоваться целочисленными константами класса ***Calendar***: AM, AM_PM, APPRIL, AUGUST, DATE, DAY_OF_MONTH, DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_YEAR, DESEMBER, DST_OFFSET, ERA, FEBRUARY, FIELD_COUNT, FRIDAY, HOUR, HOUR_OF_DAY, JANUARY, JULY, JUNE, MARCH, MAY, MILLISECOND, MINUTE, MONDAY, MONTH, NOVEMBER, OCTOBER, PM, SATURDAY, SECOND, SECOND, SEPTEMBER, SUNDAY, THURSDAY, TUESDAY, UNDECIMBER, WENDESDAY, WEEK_OF_MONTH, WEEK_OF_YAR, YAR, ZONE_OFFSET.

Пример использования констант класса ***Calendar***:

```
/**
 * Константы класса Calendar.
 *
 * Создание календаря с текущей датой,
 * языковой зоной и часовом поясе.
 */
Calendar cal = Calendar.getInstance();
msg.println("Текущее состояние календаря:");
msg.printf("\t Год    : %d \n", cal.get(Calendar.YEAR));
msg.printf("\t Месяц  : %d \n", cal.get(Calendar.MONTH));
msg.printf("\t День   : %d \n", cal.get(Calendar.DAY_OF_MONTH));
msg.printf("\t Час    : %d \n", cal.get(Calendar.HOUR_OF_DAY));
msg.printf("\t Минута : %d \n", cal.get(Calendar.MINUTE));
msg.printf("\t Секунда: %d \n", cal.get(Calendar.SECOND));
msg.printf("\t Мл.сек : %d \n", cal.get(Calendar.MILLISECOND));
```

Задание.

В проекте с именем ***proj1*** написать программу, обеспечивающую форматированный вывод даты и времени на консоль терминала.

1.1.3 Особенности использования метода main(...)

Для изучения последующих вопросов данного практического занятия необходимо предварительно прочитать [1, подраздел 2.2, стр. 51-58].

Каждая исполняемая программа на языке Java должна иметь метод ***main(...)***, который имеет следующее общее описание:

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

В отличие от языка ***C***, метод ***main(...)*** языка Java не возвращает обязательного целочисленного значения, необходимого ОС для нормального завершения процесса. Естественно, что этим вопросом занимается виртуальная машина Java - JVM. Если же необходимо, чтобы метод ***main(...)*** или любой другой метод любого класса возвращал нужное значение кода возврата программы, то следует воспользоваться методом ***exit(...)*** класса ***System***. Это обеспечивается выражением:

```
System.exit( целочисленный код возврата );
```

Хотя метод ***main(...)*** имеет особый статус, — как главный метод запуска программ, на него распространяются все правила определения и использования методов классов языка Java:

- все объектные переменные, объявленные внутри метода являются локальными в пределах его границ;

- если в классе объявлены объектные переменные или другие методы отличные от ***main(...)***, то для их использования необходимо явное создание объекта класса.

Например:

```
package rvs.pr1;

public class Main2 {
	/**
	 * Глобальные переменные.
	 * @param args
	 */
	private	String version = "1.0";
	public	static String title	= "Пример программы";

	public String  getVersion()
	{
		return version;
	}

	public static void main(String[] args) {
		/**
		 * Создание локального объекта.
		 */
		Main2 m2 = new Main2();
		System.out.println(title + ": " + m2.getVersion());

	}
}
```

1.1.4 Преобразования простых типов данных

Основные обозначения ключевых слов, операторов и простых типов данных приведены в [1, пункт 2.2.1]. В целом, они не требуют пояснений для студентов знакомых с основами ООП.

Поскольку Java оперирует с объектами, то в [1, пункт 2.2.2, стр. 53] приведена следующая таблица преобразования типов:

Таблица 2.4 - Простые типы данных и классы-обёртки языка Java [1]

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

Общепринятые обозначения Java предполагают, что:

а) *имена классов* начинаются с прописной буквы;

б) *имена методов* начинаются со строчной буквы.

Эти условия не являются обязательными, но их все придерживаются.

В процессе программирования, постоянно приходится проводить преобразование простейших типов данных. Многие методы делают это неявно, например, методы класса ***PrintStream***, рассмотренные выше. Для других достаточно явного преобразования типа, указанного в круглых скобках. Но возникают случаи, когда

имеются сомнения в правильности проведённых преобразований или среда разработки указывает на несоответствие типов и блокирует выполнение приложения. Тогда следует:

а) *преобразовать* простейший тип в соответствующий ему объект, согласно обозначению типов в таблице 2.4; для этого имеются методы ***valueOf(...)***.

б) *использовать* методы созданного объекта для преобразования его в нужный простой тип; для этого используются различные методы, как показано ниже в примере.

```
package rvs.pr2;

/**
 * Примеры преобразования типов.
 * @author vgr
 *
 */
public class Types {

	public static void main(String[] args) {

		/**
		 * Преобразование целого числа в объект и другие типы.
		 */
		int      i = 5;
		Integer ii = Integer.valueOf(i);
		float   f  = ii.floatValue();
		double  d  = ii.doubleValue();
		long    l  = ii.longValue();
		/**
		 * Преобразование целого числа в строку.
		 */
		String ss = String.valueOf(i);
		/**
		 * Преобразование строки во float и другие типы.
		 */
		Float ff = Float.valueOf(ss);
		f = ff.floatValue();
		d = ff.doubleValue();
		l = ff.longValue();

	}
}
```

Задание.

В проекте с именем ***proj2*** написать программу, обеспечивающую преобразование одних простых типов в другие.

1.1.5 Использование массивов

В Java массивы являются объектами, которые могут создаваться и передаваться в другие методы. Перед использованием массивов для них сначала нужно выделить требуемый объем памяти. В целом процесс создания и использования массивов Java можно разделить на три основных этапа:

а) объявление массива.
б) выделение памяти для элементов массива.
в) инициализация элементов массива.

Практическое объявление и использование массивов достаточно многообразно, поэтому объясним это набором примеров, реализованных в проекте ***proj2*** в виде класса ***Array2***:

```
package rvs.pr3;
/**
 * Примеры определения массивов.
 * @author vgr
 *
 */
public class Array2 {

	public static void main(String[] args) {
		int max = 20;
		/**
		 * Объявление массивов:
		 */
		double[] d1;
		int[]    i1;
		long[][] l1;
		String[] s1;
		/**
		 * Выделение памяти для элементов массива:
		 */
		d1 = new double[40];
		i1 = new int[max];
		l1 = new long[10][max];
		s1 = new String[10];
		/**
		 * Инициализация элементов массива:
		 */
		for(int i=0; i<2*max; i++)
			d1[i] = 2.5 * i;

		for(int i=0; i<10; i++)
		{
			i1[i] = 30 *i;
			for(int j=0; j<max; j++)
				l1[i][j] = i + j;
		}

		s1[0] = "Текст0";
		s1[1] = "Текст1";

		double[] d2 = {2.5, 4.7, 5.6};       // Три элемента
		long[][] l2 = {{7, 3, 2},
```

```
                                {6, 5, 20}};    // Массив 2x3
            String[] s2 = {"Текст0",
                           "Текст1",
                           "Текст2"};           // Три элемента
      }
}
```

В качестве данных объявляемых массивов могут выступать классы. В следующем примере класс ***Type2*** используется как массив в классе ***Array3***:

```
package rvs.pr3;

/**
 * Класс, выступающий как элемент массива.
 * @author vgr
 *
 */
class Type3
{
      int    i;
      double d;
}

/**
 * Класс использующий другой класс как массив.
 * @author vgr
 *
 */
public class Array3 {

      public static void main(String[] args) {
            /**
             * Объявляем и инициализируем массив объектов.
             */
            int N = 10;
            Type3[] t3 = new Type3[N]; // Выделяем ссылки.
            for(int i=0; i<N; i++)
            {
                  t3[i]   = new Type3(); // Создаём объект.
                  t3[i].i = i;
                  t3[i].d = 1.41 * i;
            }
      }
}
```

1.1.6 Работа со строками

В программировании постоянно приходится использовать строковые объекты, которые представляют собой массивы символов. Язык Java предоставляет два типа строковых объектов:

1. ***java.lang.String*** — *неизменяемый* строковый объект; все действия совершаемые по изменению такого объекта приводят к созданию новой строки.
2. ***java.lang.StringBuffer*** — *изменяемый* строковый объект, допускающий доступ к его элементам и изменению самого объекта.

Достаточно полному описанию свойств этих объектов посвящена глава 13

учебника [2, глава 13, стр. 331-357]. В данном пункте, мы ограничимся описанием наиболее часто употребляемых свойств, которые продемонстрируем конкретными примерами. Начнём с класса ***String***. Обратите внимание, что в каждом операторе создаётся новая строка:

```
package rvs.pr3;
/**
 * Примеры использования класса String.
 * @author vgr
 *
 */
public class String2 {

	public static void main(String[] args) {
		// Типичное создание строки
		String str1  = "  Первый пример создания строки  ";

		// Создание строки на основе массива символов
		char[] char1 = {'П','е','р','в','ы','й'};
		String str2  = new String(char1);

		// Создание пустого строкового объекта
		String str3 = new String();

		// Копирование строки str1 в строку str3
		str3 = str1;

		// Извлечение 5-го символа из строки str1
		char char2 = str1.charAt(5);

		// Удаление всех пробелов в начале и конце строки
		str1 = str1.trim();

		// Выделение подстроки
		str3 = str1.substring(0,6);

		// Сравнение строк
		if(str2.equals(str3))
			System.out.println("Строки равны");

		// Замена символа 'е' на символ 'Е' в строке str1
		str1 = str1.replace('ы', 'Ы');

		// Разделение строки на слова и вычисление длины слов
		String[] str4 = str1.split(" ");
		for(int i=0; i<str4.length; i++)
			System.out.println(str4[i] + ": " + str4[i].length()
					+ " -> " + str4[i].getBytes().length);
	}
}
```

Теперь рассмотрим класс ***StringBuffer***. Главное его отличие — возможность изменения объекта без создания нового экземпляра. Это значительно увеличивает скорость вычислений, если со строками производится множество преобразований.

Класс ***StringBuffer*** содержит три основных конструктора:

а) создание пустой строки ***StringBiffer()***;
б) создание строки с начальным значением ***StringBiffer(String str)***.
в) создание пустой строки с выделением памяти ***StringBiffer(int length)***.

В любом случае, память буферу выделяется автоматически, поэтому, если строка длиннее в выделенного буфера, то ему будет добавлена память. Приведём несколько примеров:

```
package rvs.pr3;
/**
 * Примеры использования класса StringBuffer.
 * @author vgr
 *
 */
public class BString {

	public static void main(String[] args) {
		// Пустой буфер
		StringBuffer sb1 = new StringBuffer();

		// Пустой буфер размера 10 символов
		StringBuffer sb2 = new StringBuffer(10);

		// Буфер инициализирован строкой
		StringBuffer sb3 =
				new StringBuffer("  Первый пример создания строки  ");

		// Добавление в начало строки
		sb3.insert(0, "<Начало строки>");

		// Добавление в конец строки
		sb3.append("<Конец строки>");
		System.out.println(sb3);

		// Реверс строки
		sb3.reverse();
		System.out.println(sb3);

	}
}
```

1.1.7 Управляющие операторы

Описание управляющих операторов перечисленно в [1, подраздел 2.3, стр. 59-60]. Обычно их использование не вызывает затруднений у студентов изучивших языки C и C++. Тем более, что они постоянно используются во всех приведённых примерах. Если все же возникают какие-либо вопросы по их применению управляющих операторов, то следует воспользоваться учебником [2, глава 5, стр. 106-132].

Задание по практическому занятию №1. Следует внимательно изучить примеры приведённые в данном подразделе и добавить в них свои комментарии и вывод на печать. Этот материал используется в лабораторной работе №2.

1.2 Практическое занятие №2 — лабораторная №3. Организация ввода-вывода в объектной модели Java

Второе практическое занятие дисциплины посвящено организации ввода-вывода в объектной модели языка Java. Учебный материал этого подраздела используется в лабораторной работе №3.

Минимальные теоретические представления, которыми должен обладать студент приступающий к данному занятию, изложены в учебнике [1, подраздел 2.4, стр. 61-68]. Изложенных там примеров достаточно для понимания последующего материала дисциплины. Тем не менее, студенту следует более широко ориентироваться в инструментах ввода-вывода, которые сосредоточены в отдельном пакете ***java.io***. Достаточно полное описание этих средств изложено в учебнике [2, глава 17, стр. 501-547].

Учебный материал данного занятия ограничен только тремя аспектами из общего состава предлагаемых инструментов, изложенных отдельными пунктами:

1. Работа с файлами и каталогами ОС.
2. Ввод/вывод объектов.
3. Символьный ввод/вывод.

1.2.1 Файлы и каталоги

Программисту постоянно приходится обращаться к файлам и каталогам за доступом к информации. Полученную в процессе работы программ информацию необходимо сохранять в файлах. Кроме того, программисту следует учитывать ограничения доступа к файлам и каталогам, которые наложены ограничениями ОС и распространяются на инструментальные средства языка Java.

Для решения большинства указанных вопросов Java предлагает средства класса ***java.io.File***, демонстрационный пример которых представлен в [1, листинг 2.10, стр. 68].

Листинг 2.10 — Исходный текст класса Example8 из среды Eclipse EE [1]

```
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()));
      }
}
```

Этот пример показывает, что про любой файл можно узнать много интересного, но необходимо учитывать в какой ОС вы работаете. Сам класс ***File*** содержит много других методов. Официальная документация на него доступна, например, по ссылке: https://docs.oracle.com/javase/6/docs/api/java/io/File.html.

Для создания объекта типа ***File*** следует воспользоваться услугами одного из четырех конструкторов:

1. ***File (File parent, String child)*** — создаёт новый экземпляр типа ***File*** из родительского абстрактного пути и строки дочернего пути.
2. ***File (String pathname)*** — создаёт новый экземпляр файла путём преобразования заданной строки пути в абстрактный путь.
3. ***File (String parent, String child)*** — создаёт новый экземпляр ***File*** из строки родительского пути и строки дочернего пути.
4. ***File (URI uri)*** - создаёт новый экземпляр ***File*** путём преобразования указанного URI-файла в абстрактный путь.

Дополнительно, можно воспользоваться четырьмя статическими полями, учитывающими используемую ОС:

1. ***static String pathSeparator*** — системнозависимый символ-разделитель пути, представленный для удобства в виде строки.
2. ***static char pathSeparatorChar*** — системнозависимый символ-разделитель пути.
3. ***static String separator*** — системнозависимый символ разделителя имён по умолчанию, представленный для удобства в виде строки.
4. ***static char separatorChar*** — системнозависимый символ разделителя имён по умолчанию.

Кроме информационных методов, часть из которых представлена выше, объекты типа ***File*** имеют методы, существенно влияющие на содержимое файловых систем. К ним можно отнести:

- ***static File createTempFile (String prefix, String suffix)*** — создаёт пустой файл в каталоге временных файлов по умолчанию, используя заданный префикс и суффикс для генерации его имени.
- ***static File createTempFile (String prefix, String suffix, File directory)*** — создаёт новый пустой файл в указанном каталоге, используя заданный префикс и строки суффикса для генерации его имени.

- ***boolean delete()*** — удаляет файл или каталог, обозначенный этим абстрактным путем.
- ***void deleteOnExit ()*** — требует, чтобы файл или каталог, обозначенный этим абстрактным путём, были удалены при завершении работы виртуальной машины.
- ***String[] list()*** — возвращает массив строк с именами файлов и каталогов в каталоге, обозначенном этим абстрактным путём.
- ***String[] list (FilenameFilter filter)*** — возвращает массив строк с именами файлов и каталогов в каталоге, обозначенном этим абстрактным путём, которые удовлетворяют указанному фильтру.
- ***File[] listFiles()*** — возвращает массив абстрактных путей, обозначающих файлы в каталоге, обозначенном этим абстрактным путём.
- ***File[] listFiles (FileFilter filter)*** — возвращает массив абстрактных путей, обозначающих файлы и каталоги в каталоге, обозначенном этим абстрактным путём, которые удовлетворяют указанному фильтру.
- ***File[] listFiles (FilenameFilter filter)*** — возвращает массив абстрактных путей, обозначающих файлы и каталоги в каталоге, обозначенном этим абстрактным путем, которые удовлетворяют указанному фильтру.
- ***boolean mkdir()*** — создает каталог, названный этим абстрактным путём.
- ***boolean mkdirs()*** — создает каталог, названный этим абстрактным путём, включая любые необходимые, но несуществующие родительские каталоги.
- ***boolean renameTo (File dest)*** — переименовывает файл, обозначенный этим абстрактным путём.
- ***boolean setExecutable (boolean executable)*** — удобный метод для установки разрешения владельца на выполнение этого абстрактного пути.
- ***URI toURI ()*** — формирует URI, который представляет это абстрактное имя пути.

Для демонстрации новых возможностей класса ***File***, в проекте ***proj3*** реализуем следующий пример:

```
package rvs.pr1;

import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

/**
 * Дополнительный пример по использованию класса File.
 * @author vgr
 *
 */
public class File2 {

	public static void main(String[] args) throws IOException {
		PrintStream msg =
				System.out;
		// Получаем домашнюю директорию пользователя
		String home =
				System.getenv("HOME");
		msg.println("Домашняя директория: " + home);
```

```
            msg.println("Разделитель пути: " + File.pathSeparator);
            msg.println("Разделитель имен: " + File.separator);

            File     files = new File(home);

            msg.println("URI: " + files.toURI().toString());
            msg.print("Для продолжения - нажми Enter ...");
            System.in.read();

            // Получаем список всех файлов.
            String[] fs    = files.list();

            msg.println("Список каталогов:");
            for(int i=0; i<fs.length; i++)
            {
                  File ff = new File(home, fs[i]);
                  if(ff.isDirectory())
                        msg.println("\t" + fs[i]);
            }
            msg.print("Для продолжения - нажми Enter ...");
            System.in.read();

            msg.println("Список файлов:");
            for(int i=0; i<fs.length; i++)
            {
                  File ff = new File(home, fs[i]);
                  if(ff.isFile())
                        msg.println("\t" + fs[i]);
            }
            msg.print("Для продолжения - нажми Enter ...");
            System.in.read();

            // Создаем каталог $HOME/src2
            File dir = new File(home, "src2");
            msg.print("Директория: " + dir.getPath());
            if(dir.mkdir())
                  msg.println(" - создана!");
            else
                  msg.println(" - не создана!");

            // Удаляем созданный каталог
            msg.print("Проверь наличие каталога и нажми Enter ...");
            System.in.read();

            if(dir.delete())
                  msg.println("\n\nКаталог - удален!");
            else
                  msg.println("\n\nКаталог - не удален!");
      }
}
```

1.2.2 Сериализация объектов

В общем случае, классы создаваемые пользователем реализуют объекты, которые не являются линейными. Другими словами, их нельзя представить в виде последовательности байт и передавать по сети или записывать в файлы.

Чтобы классы реализовывали линейные объекты, они должны:

1. Поддерживать интерфейс ***Serializable***.
2. Включать в себя только объекты, классы которых поддерживают интерфейс ***Serializable***.

Рассмотрим пример простейшего линейного класса, реализованного в проекте ***proj3***, который:

- создает объект, сохраняющий в себе сообщение в момент своего создания;
- записывающий себя в файл ***$HOME/src/Serial1.dat*** с помощью потока класса ***ObjectOutputStream***.

```
package rvs.pr2;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.io.Serializable;
import java.util.Date;

/**
 * Класс, записывающий себя в файл.
 * @author vgr
 *
 */
public class Serial1 implements Serializable
{
	/**
	 * Стандартная константа версии.
	 */
	private static final long serialVersionUID = 1L;

	// Сохраняемое сообщение:
	String file;
	String msg;

	// Конструктор
	Serial1()
	{
		file = System.getenv("HOME")
				+ File.separator + "src"
				+ File.separator + "Serial1.dat";
		msg  = "Сообщение записано " + new Date()
				+ "\nв файл: " + file;
	}

	// Вывод сохраненного сообщения
	public void printMsg()
	{
		System.out.print(msg);
	}

	// Метод main()
	public static void main(String[] args) {
		// Создаем объект
		Serial1 serial =
				new Serial1();
```

```
            try
            {
                  // Записываем объект в файл.
                  OutputStream out =
                              new FileOutputStream(serial.file);

                  ObjectOutputStream oos =
                              new ObjectOutputStream(out);
                  oos.writeObject(serial);
                  oos.flush();
                  oos.close();
                  out.flush();
                  out.close();

                  // Печатам сообщение.
                  serial.printMsg();
            }
            catch(FileNotFoundException e)
            {
                  System.out.print(e.getMessage());
            }
            catch (IOException e2)
            {
                  e2.printStackTrace();
            }
      }
}
```

Теперь, любая программа, которой доступно описание класса ***Serial1*** и которая знает, где записан созданный объект, может создать этот объект, воспользовавшись объектным чтением из файла ***ObjectInputStream***.

Такой пример также реализован в проекте ***proj3***:

```
package rvs.pr2;

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

/**
 * Класс, читающий объект serial из файла.
 * @author vgr
 *
 */
public class Serial2 {

      public static void main(String[] args) {
            try
            {
                  // Создаем объектный поток ввода.
                  InputStream in =
                              new FileInputStream(System.getenv("HOME")
                                          + File.separator + "src"
                                          + File.separator + "Serial1.dat");

                  ObjectInputStream ois =
                              new ObjectInputStream(in);
```

```
                // Читаем объект из файла.
                Serial1 serial =
                            (Serial1)ois.readObject();
                ois.close();
                in.close();

                // Печатаем сохраненное сообщение объекта.
                serial.printMsg();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            } catch (ClassNotFoundException e2)
            {
                e2.printStackTrace();
            }
        }
}
```

Таким же образом можно передавать объект через сеть, но у противоположной стороны обязательно должно быть описание класса ***Serial1***.

1.2.3 Символьные потоки ввода-вывода

В учебнике [1, подраздел 2.4] были рассмотрены *байтовые потоки* ввода-вывода, основанные на классах ***InputStream*** и ***OutputStream***. Они обеспечивают решение всех задач, связанных с программированием чтения и записи произвольных сериализованных типов данных (объектов), но вызывают затруднения, когда необходимо обрабатывать символьные данные, предполагающие использование национальных языков и различных кодировок символов.

Для оптимизации работы с символьными данными пакет ***java.io*** предлагает абстрактные классы ***java.io.Reader*** и ***java.io.Writer***. На их основе созданы дополнительные подклассы, ориентированные на специальные направления.

За работу с символьными потоками ввода отвечают следующие подклассы ***Reader***:

- ***BufferedReader*** — определяет возможности использования буферизованных потоков ввода;
- ***LineNumberReader*** — определяет дополнительные возможности считывания из потоков ввода строк с учётом их нумерации;
- ***CharArrayReader*** - определяет возможности использования символьного потока на основе массивов данных;
- ***FilterReader*** — определяет набор действий обработки фильтруемых потоков ввода;
- ***PushBackReader*** — определяет возможность возвращения обратно в поток ввода считанного символа;
- ***InputStreamReader*** — определяет возможности трансляции (перевода) байтовых потоков ввода в символьные;
- ***FileReader*** — определяет возможности использования в качестве потоков ввода файлов на жёстком диске компьютера;

- ***PipedReader*** — определяет возможности использования канальных потоков ввода символов;
- ***StringReader*** — определяет возможности использования строк в потоках ввода.

Общий доступ к документации, описывающей эти классы, возможен по адресу: https://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html.

Мы остановимся на классе ***InputStreamReader***, который реализует своеобразный мост от байтовых потоков к символьным потокам: он считывает байты и декодирует их в символы, используя указанную кодировку. Используемая им кодировка может быть указана по имени или задана явно, или может быть принят кодовый набор по умолчанию для платформы. Это обеспечивается конструкторами:

- ***InputStreamReader (InputStream in)*** — создаёт поток InputStreamReader, который использует набор символов по умолчанию.
- ***InputStreamReader (InputStream in, Charset cs)*** — создаёт новый поток InputStreamReader, который использует данный набор символов.
- ***InputStreamReader (InputStream in, CharsetDecoder dec)*** — создаёт поток InputStreamReader, который использует данный декодер ***charset***.
- ***InputStreamReader (InputStream in, String charsetName)*** — создаёт поток InputStreamReader, который использует именованную кодировку.

Каждый вызов одного из методов ***read()*** класса ***InputStreamReader*** может привести к тому, что один или несколько байтов будут считаны из базового потока байтового ввода. Чтобы обеспечить эффективное преобразование байтов в символы, из базового потока может быть прочитано больше байтов, чем необходимо для выполнения текущей операции чтения. Обслуживание потока ввода осуществляется методами:

- ***void close ()*** — закрывает поток и освобождает любые системные ресурсы, связанные с ним.
- ***String getEncoding ()*** — возвращает имя кодировки символов, используемой этим потоком.
- ***int read ()*** - читает один символ; символ читается, или -1, если достигнут конец потока;
- ***int read (char [] cbuf, int offset, int length)*** — читает символы в часть массива.
- ***boolean ready()*** - сообщает, готов ли этот поток для чтения.

Продемонстрируем использование этого классом примером, разместив его в проекте ***proj1***:

```
package rvs.pr3;

import java.io.IOException;
import java.io.InputStreamReader;

/**
 * Демонстрация работы потока InputStreamReader
```

```
 * @author vgr
 *
 */
public class ReadChars
{

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

		System.out.println("Для выхода из программы, нажмите:\n" +
		"Ctrl-Z в DOS/Windows\n" +
		"Ctrl-D в UNIX/Linux");

		InputStreamReader isr =
				new InputStreamReader(System.in);
		try
		{
			while ((mKey = isr.read()) != -1)
			{
				if( mKey == 13 )
				{
					System.out.println("\nВозврат каретки");
					continue;
				}
				if( mKey == 10 )
				{
					System.out.println("\nПеревод строки");
					continue;
				}
				System.out.print(mKey + " - "
					+ (char)mKey + ", ");
			}
			isr.close(); // Закрываем поток
		}
		catch(IOException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Следует сравнить результаты работы этого примера с результатом работы класса ***Example4*** из этого же проекта.

За работу с символьными потоками вывода языка Java отвечают следующие потомки абстрактного класса ***java.io.Writer***:

- ***BufferedWriter*** — класс определяет возможности использования буферизованных потоков вывода;
- ***CharArrayWriter*** — класс определяет возможности использование потока для записи данных в символьный массив;
- ***FilterWriter*** — класс определяет набор действий обработки фильтруемых потоков вывода;
- ***OutputStreamWriter*** — определяет возможности перевода символьных потоков вывода в байтовые;
- ***FileWriter*** — определяет возможности вывода символьных потоков в файлы;

- ***PipedWriter*** — класс определяет возможности вывода символьных потоков в каналы;
- ***PrintWriter*** — определяет возможности использования методов ***println(...)*** и ***print(...)*** для вывода информации;
- ***StringWriter*** — класс определяет возможности записи символьных строк в потоки вывода.

Более подробно рассмотрим поток ***OutputStreamWriter***, (см. сайт по адресу: https://docs.oracle.com/javase/6/docs/api/java/io/OutputStreamWriter.html).

Он имеет аналогичные классу ***InputStreamReader*** методы и возможности, которые продемонстрируем примером:

```
package rvs.pr3;

import java.io.IOException;
import java.io.OutputStreamWriter;

/**
 * Демонстрация работы потока OutputStreamWriter
 * @author vgr
 *
 */
public class WriteChars {

	public static void main(String[] args) {
		// Строка на русском языке.
		String rstr = "Пример использования русских символов\n";

		for(int i=0; i<rstr.length(); i++)
			System.out.write(rstr.charAt(i));

		// Использование OutputStreamWriter
		OutputStreamWriter osw =
				new OutputStreamWriter(System.out);
		try
		{
			for(int i=0; i<rstr.length(); i++)
				osw.write(rstr.charAt(i));
			osw.close();
		}
		catch(IOException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Задание.
Для закрепления учебного материала, студенту следует написать две программы, использующие классы ***FileReader*** и ***FileWriter***.

1.3 Практическое занятие №3 — лабораторная №4. Сети и многопоточное программирование

Третье практическое занятие дисциплины посвящено вопросам использования инструментальных средств языка Java для программирования сетевых приложений. Учебный материал этого подраздела является *первой частью* теоретического материала, используемого в лабораторной работе №4.

Минимальные теоретические представления, которые должны присутствовать у студента приступающего к данному занятию, изложены в учебнике [1, подраздел 2.5, стр. 69-81]. Этот теоретический материал необходим и для выполнения следующего практического занятия.

Учебный материал данного практического занятия посвящён вопросам многопоточного программирования языка Java, реализованного средствами базового пакета ***java.lang***. Эти вопросы не рассмотрены непосредственно в учебном пособии [1], поскольку они относятся к тематике дисциплины «*Операционные системы*». Тем не менее, когда Java используется для низкоуровневого программирования задач сетевого взаимодействия, они становятся актуальными, поэтому и рассматриваются на данном практическом занятии.

Достаточно полное описание средств изложено в учебнике [2, глава 11, стр. 263-300]. Мы ограничим наше занятие только двумя аспектами этой тематики:

1. Использование интерфейса ***Runnable***.
2. Синхронизация многопоточного приложения.

1.3.1 Использование интерфейса Runnable

Терминология ОС манипулирует двумя понятиями: *мультипрограммирование* и *мультизадачность*.

Мультипрограммирование — это способность ОС запускать множество программ (процессов) и управлять ими.

Мультизадачность (в терминах ОС) — это способность процесса параллельно выполнять отдельные части программы.

С точки зрения прикладного программиста, *мультизадачность* может быть реализована тремя способами: распараллеливанием приложения на несколько процессов, использование нескольких нитей (потоков, тредов - ***threads***) или — того и другого вместе.

Язык Java обеспечивает *мультизадачность (**multitasking**)* средствами *мультипоточного (многопоточного) программирования*.

Многопоточная система Java построена на классе ***java.lang.Thread***, описание которого можно найти в официальной документации, расположенной на сайте: https://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html. Он имеет четыре основных конструктора:

- Thread();
- Thread(Runnable target);
- Thread(Runnable target, String name);
- Thread(String name);

где:

- ***target*** — объект, реализующий интерфейс ***Runnable***;
- ***name*** — имя создаваемого интерфейса.

Этот класс реализует много методов, наиболее важными из которых являются следующие:

- ***static Thread currentThread()*** - возвращает ссылку на текущий запущенный поток;
- ***String getName()*** - получить имя потока;
- ***void setName(String name)*** - установить имя потока;
- ***int getPriority()*** - получить приоритет потока;
- ***void setPriority(int newPriority)*** — установить приоритет потока;
- ***boolean isAlive()*** - определить, выполняется ли ещё поток;
- ***void join([long millis[, int nanos]])*** — ожидать завершение потока;
- ***void run()*** - указать точку входа в поток;
- ***static void sleep(long millis[, int nanos])*** — приостановить поток на определённый период времени;
- ***void start()*** - запустить поток с помощью вызова его метода ***run()***.

Любая запущенная программа (процесс) реализует некоторый ***главный поток***. Он имеет два важных свойства:

- из него можно порождать «*дочерние*» потоки;
- он должен быть последним запущенным потоком, когда процесс завершает свою работу.

Ссылку на объект потока можно получить в процессе его выполнения с помощью статического метода ***currentThread()***. Приостановить поток на некоторое время можно с помощью метода ***sleep(...)***, а ожидать с помощью общего для всех объектов метода ***wait(...)***. При этом, следует помнить, что методы ***sleep(...)*** и ***wait(...)*** могут генерировать исключение ***InterruptExcaption***.

Использовать потоки применительно к некоторому классу можно двумя способами: расширяя класс ***Thread*** или подключая интерфейс ***Runnable***. В любом из этих случаев, программист должен реализовать метод ***run()***, который и является точной входа любого «*дочернего*» потока.

Первый способ использования потоков, основанный на расширении класса ***Thread***, необходимо применять, когда программист хочет расширить какие-то возможности этого класса. В большинстве случаев такой необходимости не наблюдается, поэтому достаточно в создаваемом классе только объявить интерфейс ***Runnable*** и реализовать метод ***run()***.

Для демонстрации использования интерфейса ***Runnable*** рассмотрим класс ***Thread2***, реализованный в проекте ***proj4*** и который запускает три дочерних потока, ждёт завершения их работы, после чего завершается сам, причём каждый дочерний поток моделирует свою работу методом ***sleep(...)***, длящимся одну секунду.

```
package rvs.pr1;
/**
 * Демонстрация запуска дочерних потоков.
 * Интерфейс Runnable.
 * @author vgr
 *
 */
public class Thread2 implements Runnable
{
	// Констаавнты класса
	String name;
	Thread th;

	// Конструктор
	Thread2(String name)
	{
		this.name = name;
		th        = new Thread(this, name);
		System.out.println("Создан поток: " + name);

		// Запуск потока
		th.start();
	}

	/**
	 * Метод run().
	 * Точка входа потока.
	 */
	public void run()
	{
		try
		{
			int i = 5;
			while(i > 0)
			{
				i--;
				System.out.println(name + " - работаю...");
				Thread.sleep(1000);
			}
		}
		catch(InterruptedException e)
		{
			System.out.println(e.getMessage());
		}
		System.out.println(name + " завершил работу...");
	}

	/**
	 * Управление потоками
	 * @param args
	 */
	public static void main(String[] args)
	{
		System.out.println("Главный поток: "
				+ Thread.currentThread());
		/**
```

```
            * Запускаю три дочерних потока...
            */
           Thread2 obj1 = new Thread2("Первый поток");
           Thread2 obj2 = new Thread2("Второй поток");
           Thread2 obj3 = new Thread2("Третий поток");

           System.out.println("\nЖдем завершения работы потоков:");
           try
           {
                 obj1.th.join();
                 obj2.th.join();
                 obj3.th.join();
           }
           catch(InterruptedException e)
           {
                 System.out.println(e.getMessage());
           }
           /**
            * Проверяем работу потоков:
            */
           System.out.println("Первый поток работает: "
                       + obj1.th.isAlive());
           System.out.println("Второй поток работает: "
                       + obj2.th.isAlive());
           System.out.println("Третий поток работает: "
                       + obj3.th.isAlive());

           System.out.println("\nЗавершаем главный поток!");
     }
}
```

1.3.2 Синхронизация многопоточного приложения

В предыдущем примере каждый из трех потоков выполнял свою работу независимо от других потоков и завершался. Как он выполнял свою работу — зависит от алгоритма реализации метода ***run()***. В этом случае, алгоритм синхронизации потоков — достаточно прост и сводится к ожиданию главным потоком состояния завершения всех дочерних потоков, используя метод ***join()***. В тех случаях, когда потоки обращаются к ресурсу, который должен использоваться эксклюзивно, необходимы дополнительные средства синхронизации.

Вопросы синхронизации многопоточных и много процессных приложений подробно рассматривались в дисциплине «*Операционные системы*» и студент с ними уже знаком. В общем случае выделяются две проблемы синхронизации:

1. Доступ к общему ресурсу или «*критической области*».
2. Взаимоблокировка процессов, когда процессы захватили только часть нужных им ресурсов и находятся в бесконечном ожидании недостающих.

Общее решение указанных проблем связано с правильным построением алгоритма работы процессов, использующих инструмент семафоров. Соответственно, общее правило избежания взаимных блокировок: *все необходимые ресурсы процесс должен захватывать одновременно*.

Когда вопросы взаимных блокировок — алгоритмически разрешены, вопро-

сы синхронизации процессов сводятся к ситуациям доступа к «*критическим областям*».

В языке Java, для доступа к критическим областям используется очень удобный инструмент, называемый ***монитором***.

Монитор — это объект, использующий взаимоисключающие блокировки (*mutually exclusive lock*) или ***mutex***, предполагающий, что только один поток может иметь собственный монитор в заданный момент, а все другие потоки, пытающиеся получить монитор будут заблокированы и будут освобождаться, когда захвативший монитор процесс освободит его.

Чтобы правильно построить алгоритм синхронизации потоков, необходимо помнить следующее:

- переключение потоков управляется внутренними механизмами ОС, обеспечивающими управление процессами и потоками, поэтому программист не может точно знать на каких операторах метода это происходит;
- переключение потоков происходит при обращении к ядру ОС;
- переключение потоков происходит при вызове ***sleep(...)***, ***wait(...)*** и других подобных методов.

Перечисленные выше свойства диспетчеризации ОС приводят потоки в *состояние состязаний* (гонок) по отношению к разделяемым ресурсам (*критическим областям*).

Наиболее распространенным способом языка Java, обеспечивающим вхождение потока в монитор, является использование оператора ***synchronized*** применительно к объекту, обслуживающему критическую область программы.

Рассмотрим пример в виде класса ***Thread3***, реализованный в проекте ***proj4***, который синхронизирует каждый отдельный цикл метода ***run()***, объявленный как критическая область вывода сообщений на консоль терминала. Для наглядности, вход в критическую область и выход из нее ограничим соответствующими сообщениями, а объектом синхронизации будет: ***System.out***.

```
package rvs.pr1;

import java.util.Date;

/**
 * Демонстрация синхронизации дочерних потоков.
 * Интерфейс Runnable.
 * @author vgr
 *
 */
public class Thread3 implements Runnable
{
	// Констaвнты класса
	String name;
	Thread th;

	// Конструктор
	Thread3(String name)
	{
		this.name = name;
```

```
            th        = new Thread(this, name);
            System.out.println("Создан поток: " + name);

            // Запуск потока
            th.start();
      }

      /**
       * Метод run().
       * Точка входа потока.
       */
      public void run()
      {
            System.out.println(name + " начал работу:");
            try
            {
                  long dt1 = new Date().getTime();
                  long dt2 = new Date().getTime();
                  int i = 3;

                  while(i > 0)
                  {
                        i--;
                        /**
                         * Оператор синхронизации
                         */
//                      synchronized(System.out)
                        {
                              System.out.println(name
                                          + " - вошел в  критическую область:");


                              while((dt2 - dt1) < 500)
                                    dt2 = new Date().getTime();

                              dt1 = dt2;

                              System.out.println(name
                                          + " - вышел из критической области.");
                        }
                        Thread.sleep(10);
                  }
            }
            catch(InterruptedException e)
            {
                  System.out.println(e.getMessage());
            }

            System.out.println(name + " завершил работу!");
      }

      public static void main(String[] args)
      {
            System.out.println("Главный поток: "
                        + Thread.currentThread());
            /**
             * Запускаю три дочерних потока...
             */
            Thread3 obj1 = new Thread3("Первый поток");
            Thread3 obj2 = new Thread3("Второй поток");
            Thread3 obj3 = new Thread3("Третий поток");
```

```
            System.out.println("\nЖдем завершения работы потоков:");
            try
            {
                  obj1.th.join();
                  obj2.th.join();
                  obj3.th.join();
            }
            catch(InterruptedException e)
            {
                  System.out.println(e.getMessage());
            }
            /**
             * Проверяем работу потоков:
             */
            System.out.println("Первый поток работает: "
                        + obj1.th.isAlive());
            System.out.println("Второй поток работает: "
                        + obj2.th.isAlive());
            System.out.println("Третий поток работает: "
                        + obj3.th.isAlive());

            System.out.println("\nЗавершаем главный поток!");
      }
}
```

В приведённом примере, оператор синхронизации — закомментирован, поэтому вывод программы имеет вид:

```
Главный поток: Thread[main,5,main]
Создан поток: Первый поток
Создан поток: Второй поток
Первый поток начал работу:
Второй поток начал работу:
Создан поток: Третий поток

Ждем завершения работы потоков:
Третий поток начал работу:
Первый поток - вошел в  критическую область:
Третий поток - вошел в  критическую область:
Второй поток - вошел в  критическую область:
Первый поток - вышел из критической области.
Второй поток - вышел из критической области.
Третий поток - вышел из критической области.
Первый поток - вошел в  критическую область:
Третий поток - вошел в  критическую область:
Второй поток - вошел в  критическую область:
Третий поток - вышел из критической области.
Первый поток - вышел из критической области.
Второй поток - вышел из критической области.
Третий поток - вошел в  критическую область:
Первый поток - вошел в  критическую область:
Второй поток - вошел в  критическую область:
Первый поток - вышел из критической области.
Третий поток - вышел из критической области.
Второй поток - вышел из критической области.
Первый поток завершил работу!
Третий поток завершил работу!
Второй поток завершил работу!
Первый поток работает: false
Второй поток работает: false
Третий поток работает: false
```

```
Завершаем главный поток!
```

Хорошо видно, что сообщения входа и выхода из критической области цикла происходят в произвольном порядке.

Если убрать комментарий с оператора синхронизации, то получим:

```
Главный поток: Thread[main,5,main]
Создан поток: Первый поток
Создан поток: Второй поток
Первый поток начал работу:
Первый поток - вошел в  критическую область:
Первый поток - вышел из критической области.
Второй поток начал работу:
Второй поток - вошел в  критическую область:
Второй поток - вышел из критической области.
Создан поток: Третий поток
Первый поток - вошел в  критическую область:
Первый поток - вышел из критической области.

Ждем завершения работы потоков:
Третий поток начал работу:
Третий поток - вошел в  критическую область:
Третий поток - вышел из критической области.
Первый поток - вошел в  критическую область:
Первый поток - вышел из критической области.
Второй поток - вошел в  критическую область:
Второй поток - вышел из критической области.
Третий поток - вошел в  критическую область:
Третий поток - вышел из критической области.
Второй поток - вошел в  критическую область:
Второй поток - вышел из критической области.
Первый поток завершил работу!
Третий поток - вошел в  критическую область:
Третий поток - вышел из критической области.
Второй поток завершил работу!
Третий поток завершил работу!
Первый поток работает: false
Второй поток работает: false
Третий поток работает: false

Завершаем главный поток!
```

Завершая тему третьего практического занятия, уделим несколько слов связи многопоточного программирования с задачами программирования сетевых задач. Фактически такая связь лежит на поверхности, поскольку любая сетевая задача предполагает взаимодействие минимум двух программ: программы клиента и программы сервера. Само такое взаимодействие предполагает операции записи и чтения из каналов связи, которые в общем случае осуществляются параллельно, образуя двусторонний канал передачи информации.

Учитывая, что количество вариантов сетевого взаимодействия достаточно многообразно, мы ограничимся лишь одним примером, которому посвящено следующее практическое занятие.

1.4 Практическое занятие №4 — лабораторная №4. Сокеты языка Java

Четвёртое практическое занятие дисциплины посвящено вопросам использования сокетов языка Java для программирования сетевых приложений. Учебный материал этого подраздела является *второй частью* теоретического материала, используемого в лабораторной работе №4.

Считается, что студент достаточно хорошо освоил материал, изложенный в учебном пособии [1, подраздел 2.5, стр. 69-81], а также разобрался с примерами программ сервера и клиента, реализованных в виде классов ***TCPServer*** (проект ***proj5***) и ***TCPClient*** (проект ***proj6***).

Учитывая, что данное практическое занятие привязано к одной лабораторной работе, как и предыдущее, мы ограничимся только одним вопросом: использование интерфейса ***Runnable*** применительно к классу ***TCPClient***, реализованный в [1, пункт 2.5.5, листинг 2.14]. Листинг этой программы представлен ниже, исключительно для удобства изложения этого материала.

Листинг 2.14 — Исходный текст класса TCPClient (заимствовано из [1])

```
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 сообщения.
 * 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 посылаем 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.print((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.TCPClients address "
                        + "port time_out count_message message\n");
        }

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

Напомню, что программа клиента читает аргументы, перечисленные в ком-

ментарии, а затем посылает заданное число одинаковых сообщений серверу. При этом, на каждое посланное сообщение клинт ожидает подтверждение от сервера и, только после этого, посылает новое. Одновременно программа клиента отвечает за синхронизацию работы клиента и сервера, отслеживая заданные тайм-ауты.

К недостаткам представленного алгоритма работы программы клиента можно отнести следующие претензии:

1. Цикл ожидания сообщения от сервера использует метод проверки буфера методом ***available()***, что приводит к бесполезной трате ресурсов ЭВМ.
2. В процессе анализа принятого сообщения, программа клиента одновременно отслеживает тайм-ауты, что значительно усложняет сам алгоритм обработки принятого сообщения.

Конечно для учебной программы такой сложности решение является вполне приемлемым, но если усложнять алгоритм обработки полезной информации и усложнять алгоритм взаимодействия клиента и сервера, то сложность программы будет сильно влиять на ее качество.

Повысить качество программы клиента можно следующим образом:

- разделить обработку принимаемой информации и отслеживание тайм-аута на два объекта, выполняющихся в разных потоках;
- вместо метода ***available()*** использовать метод ***read()***, что исключит непроизводительные расходы ресурсов ЭВМ.

Реализация таких изменений покажем листингом исходного текста класса ***TCPClient2***, к которому добавлен специальный класс ***Sync2()***, отвечающий за процедуру синхронизации:

```
package rvs.pr1;

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 сообщения.
 * 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 посылаем 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').
 */
```

```
/**
 * Определяем класс, обеспечивающий синхронизацию клиента и сервера,
 * а также аварийный выход из программы.
 *
 * Основная программа будет читать сообщение методом read()
 * с блокировкой и реагировать на синхронизацию.
 */
class Sync2 implements Runnable
{
    /**
      * Требуемые параметры, которые должны восстанавливаться
      * главной программой.
      */
    boolean flag       = true;
    boolean flagRepeat = true;
    long dTime   = 100;
    long rTime   = 0;

    InputStream  in;
    OutputStream out;
    Thread th;

    // Конструктор класса синхронизации.
    Sync2(long time){
        dTime = time;
        rTime = new Date().getTime() + dTime;
        th = new Thread(this, "Поток синхронизации");
    }

    /**
     * Метод потока, отслеживающего тайм-аут.
     */
    public void run()
    {
      System.out.println("Поток запустился");
      try
      {
            while(this.flag)  //Проверка тайм-аута
            {
                if(new Date().getTime() > this.rTime)
                {
                    if(!this.flagRepeat)    // Завершаем работу.
                    {
                        this.flag = false;  // Сообщаем головной программе.
                        this.in.close();    // Закрываем входной поток.
                        System.out.println("run() - вышел по flagRepeat");
                        return;
                    }
                    else      //Инициация синхронизации
                    {
                        System.out.println("run() - посылаю символ синхронизации");
                        this.flagRepeat = false;   // Отмечаем событие флагом.
                        this.out.write("\r".getBytes());
                        this.rTime = new Date().getTime()
                                    + this.dTime;
                    }
                }
                Thread.sleep(this.dTime/2); // Засыпаем на половину тайм-аута.
            }
            System.out.println("Поток остановился");
            System.out.println("run() - вышел по flag");
      }
```

```
      catch (IOException e1)
      {
                  this.flag = false;
                  System.out.println("run1:" + e1.getMessage());
      }
      catch (InterruptedException e2)
      {
                  this.flag = false;
                  System.out.println("run2:" + e2.getMessage());
      }
    }
}

/**
 * Головная программа использует только метод main():
 */
public class TCPClient2
{
      /**
       * Главный поток программы.
       * @param args
       */
      public static void main(String[] args)
      {
        int ch = (int)'\n';
        System.out.println("\\n = " + ch);
        int cr = (int)'\r';
        System.out.println("\\r = " + cr + "\n");

        InetAddress remotehost; //Адрес сервера
        int port = 0;           //Номер порта сервера
        int nn   = 0;           //Количество сообщений
        String mes = " ";       //Сообщение клиента
        long dTime   = 100;
        long curtime = 0;

        Socket clientsocket;  //Объявление клиентского сокета

        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();

            /**
             * Создаем экземпляр объекта синхронизации.
             */
            Sync2 obj =
                        new Sync2(dTime);

            /**
             * Создаем потоки ввода/вывода в объекте синхронизации.
             */
            obj.in  = clientsocket.getInputStream();
            obj.out = clientsocket.getOutputStream();

            /**
             * Запускаем поток синхронизации.
             */
            obj.th.start();

//System.in.read();

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

                /**
                 * Фиксируем границу тайм-аута в объекте синхронизации.
                 */
                curtime = new Date().getTime();
                obj.rTime = curtime + obj.dTime;

                /**
                 * Печатаем отосланное сообщение.
                 */
                System.out.println((curtime - statime) + ": " + mes);

                /**
                 * Цикл ожидания ответа сервера:
                 */
                while(obj.flag)
                {
                    //Блокирующая операция чтения по одному байту
                    ch = obj.in.read();

                    if(ch == -1) // Завершаем работу
                    {
                        obj.flag = false;
                        break;
                    }

                    System.out.print((char)ch); //Печатаем символ.
```

```
                    if(ch == 10)
                    {   /**
                         * Если пришло подтверждение, то выходим
                         * из цикла ожидания.
                         */
                        break;
                    }
                    if(ch == 13)
                    {   /**
                         * Если пришла синхронизация, то
                         * заново отправляем пакет и восстанавливаем
                         * объект синхронизации.
                         */
                        obj.out.write(mes.getBytes());
                        obj.rTime = new Date().getTime() + obj.dTime;
                        obj.flagRepeat = true;
                    }
                }
                  if(!obj.flag) // Выход по флагу.
                        break;
            }

            /**
             * Ожидаем завершение потока синхронизации.
             */
            obj.flag = false;
            obj.th.join();

            /**
             * Завершаем работу программы.
             */
            obj.out.flush();
            obj.out.close();
            obj.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 rvs.pr1.TCPClient2 address "
                        + "port time_out count_message message\n");
        }

        System.out.println("\nTCPclient2  stop: " + new Date());
        System.exit(0);
    }
}
```

1.5 Практическое занятие №5 — лабораторная №5. SQL-запросы к базам данных

Пятое практическое занятие дисциплины посвящено вопросам использования пакета ***java.sql*** языка Java для программирования SQL-запросов к базам данных на примерах работы с СУБД Apache Derby. Учебный материал этого подраздела является дополнением к учебному материалу, используемому в лабораторной работе №5.

Считается, что студент достаточно хорошо освоил материал, изложенный в учебном пособии [1, подраздел 2.6, стр. 82-96], а также разобрался с примерами программ, описанных в [1, пункте 2.6.3] и реализованных в проекте ***proj7*** классом ***Example11***.

Цель данного практического занятия — более подробное изучение типов данных, используемых СУБД Derby, а также — функций, применяемых в типовых запросах к СУБД на стандартном языке SQL. Соответственно, учебный материал этого занятия изложен в наборе изложенных ниже пунктов.

Общее описание всех типов данных СУБД Apache Derby можно найти на английском языке, в официальной документации [4]. Учитывая особую значимость некоторых типов данных, они кратко описаны в данном подразделе методического пособия. Более того, выбран конспективный вид описания, учитывающий, что студент имеет достаточный навык их понимания, полученный при изучении курса «*Базы данных*».

1.5.1 Числовые типы данных

Числовыми являются типы, которые имеют строго определённый размер:

- *Целочисленные типы*
  - `SMALLINT(2 bytes)`
  - `INTEGER     (4 bytes)`
  - `BIGINT      (8 bytes)`
- *Числа с плавающей запятой*
  - `REAL              (4 bytes)`
  - `DOUBLE PRECISION (8 bytes)`
  - `FLOAT (алиес для DOUBLE PRECISION или REAL)`
- *Exact numeric (точный числовой тип)*
  - `DECIMAL (устанавливаемая базовая точность)`
  - `NUMERIC (алиес DECIMAL)`

Тип SMALLINT

```
-32768 (java.lang.Short.MIN_VALUE)
```

```
 32767 (java.lang.Short.MAX_VALUE)
```

Тип INTEGER

```
-2147483648 (java.lang.Integer.MIN_VALUE)
 2147483647 (java.lang.Integer.MAX_VALUE)
```

Тип DECIMAL

```
{ DECIMAL | DEC } [(precision [, scale ])]
```

Приведём ряд примеров:

```
-- здесь только уменьшается точность
values cast (1.798765 AS decimal(5,2));
1
--------
1.79

-- здесь - ошибка
values cast (1798765 AS decimal(5,2));
1
--------
ERROR 22003: The resulting value is outside the range
for the data type DECIMAL/NUMERIC(5,2).
```

1.5.2 Строковые типы данных

Тип CHAR

CHAR - строка символов фиксированной длины не более 254 байта.

CHAR[ACTER] [(length)]

length — целое число в байтах (по умолчанию = 1).
Соответствует:

- Java type: java.lang.String
- JDBC metadata type (java.sql.Types): CHAR

Пример:

```
VALUES 'hello this is Joe''s string';

-- создание таблицы с полем CHAR
```

```
CREATE TABLE STATUS (
STATUSCODE CHAR(2) NOT NULL
    CONSTRAINT PK_STATUS PRIMARY KEY,
    STATUSDESC VARCHAR(40) NOT NULL
);
```

Тип VARCHAR

VARCHAR — строка переменной длины, максимум 32,672 символа.

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

Соответствует:
- Java type: java.lang.String
- JDBC metadata type (java.sql.Types): VARCHAR

Необходимо учесть следующие особенности применения этого типа данных:

- Derby ничем не набивает значение VARCHAR, когда длина строки — меньше заданной.
- Derby удаляет конечные пробелы строки, когда ее длина больше, чем специфицировано VARCHAR.
- Другие символы, отличные от пробелов, не удаляются и генерируется исключение.
- Когда CHARs и VARCHARs совместно сравниваются, более короткое значение добивается пробелами до длины большего значения.

Замечание

Тип строковой константы - CHAR, а не VARCHAR.

1.5.3 Типы даты и времени

Тип DATE

DATE — строгий тип формата *year-month-day* доступный в пакете *java.sql.Date*.

DATE доступен:
- при компиляции Java type: ***java.sql.Date***
- метаданные JDBC (java.sql.Types): ***DATE***

Даты, время и временные штампы не могут быть перемешаны в каких либо сравнениях. Любое значение, которое распознаётся посредством методом типа ***java.sql.Date*, -** доступно в столбце, ссылающемся на *SQL date/time* тип данных.

Derby доступны следующие форматы для DATE:

```
yyyy-mm-dd
mm/dd/yyyy
dd.mm.yyyy
```

Год должен задаваться *четырьмя цифрами*.
Месяц и **день** могут иметь *одну или две цифры*.

Derby также принимает строки *в локализованном* специфическом формате даты, используя *локализации сервера* СУБД.

Примеры:

```
VALUES DATE('1994-02-23')

VALUES '1993-09-01'
```

Тип TIME

TIME определяет строгий формат времени дня.

TIME соотносится:
- Java type: ***java.sql.Time***
- JDBC метатип (java.sql.Types): ***TIME***

Даты, время и временные штампы не могут быть перемешаны в каких либо сравнениях. Любое значение, распознаваемое методами ***java.sql.Time***, доступно в столбце соотносящимся с *SQL date/time* типом данных. ***Derby*** доступны следующие форматы для TIME:

```
hh:mm[:ss]
hh.mm[.ss]
hh[:mm] {AM | PM}
```

Часы могут задаваться одной или двумя цифрами. **Минуты** и **секунды**, если присутствуют, должны иметь две цифры.

Derby также допускает *специализированный локализованный формат*, используя

локализации СУБД.

Примеры:

```
VALUES TIME('15:09:02')
VALUES '15:09:02'
```

Тип данных TIMESTAMP

TIMESTAMP обеспечивает комбинированную установку значений DATE и TIME. Он допускает дополнительное дробление секунд *до 6 цифр после запятой*.

TIMESTAMP соотносит:
- Java type: ***java.sql.Timestamp***
- JDBC мета тип (java.sql.Types): ***TIMESTAMP***

Даты, время и временные штампы не могут быть перемешаны в каких либо сравнениях. ***Derby*** обеспечивает следующие форматы для TIMESTAMP:

```
yyyy-mm-dd hh:mm:ss[.nnnnnn]
yyyy-mm-dd-hh.mm.ss[.nnnnnn]
```

Год всегда должен иметь 4 цифры.

Месяцы, дни, и часы могут иметь одну или две цифры.

Минуты и секунды должны иметь по две цифры.

Наносекунды, если присутствуют, могут иметь от одной до 6 цифр.

Derby также допускает специализированный *локализованный формат*, используя локализации СУБД.

Примеры:

```
VALUES '1960-01-01 23:03:20'
VALUES TIMESTAMP('1962-09-23 03:23:34.234')
VALUES TIMESTAMP('1960-01-01 23:03:20')
```

1.5.4 Специальные типы данных

Тип BLOB

BLOB (binary large object) — строка переменной длины, которая может быть длиной *до 2,147,483,647 символов*. Подобно другим бинарным типам, *строка BLOB не ассоциируется с code page*. Дополнительно, строка BLOB *не является строкой символьного типа*.

Замечание

Длина BLOB специфицируется в байтах. По умолчанию 2Гб.

Синтаксис типа BLOB:

```
{ BLOB | BINARY LARGE OBJECT } [ ( length [{K |M |G }] ) ]
```

- java тип: ***java.sql.Blob***
- java.sql.Types: ***BLOB***
- применительно к объектам типа ***java.sql.ResultSet*** применяется метод *getBlob*

Пример:

```
create table pictures(name varchar(32) not null primary key, pic blob(16M));

-- поиск всех логотипов изображений

select length(pic), name from pictures where name like '%logo%';

-- поиск всех дублируемых изображений (blob comparisons)

select a.name as double_one, b.name as double_two
      from pictures as a, pictures as b
      where a.name < b.name
      and a.pic = b.pic
      order by 1,2;
```

Данные типа CLOB

Значения данных типа ***CLOB*** (character large object) могут быть длиной до 2,147,483,647 символов.

CLOB использует кодировку *unicode* и позволяет хранить документы большой длины на любых языках. Длина поля для CLOB может использовать масштабирующие символы *K*, *M*, или *G*, предполагающие соответствующее умножение на 1024, 1024*1024, 1024*1024*1024.

Length для CLOB специализируется в символах (*unicode*).

```
{CLOB |CHARACTER LARGE OBJECT} [ ( length [{K |M |G}] ) ]
```

По умолчанию, ***CLOB*** без спецификатора ***length*** допускает *2Г байт символов* (2,147,483,647).

CLOB соотносится:
- Java type: ***java.sql.Clob***
- JDBC мета тип (java.sql.Types): ***CLOB***

Используйте *метод* ***getClob***, при извлечении данных из объектов типа ***java.sql.ResultSet*** .

Абстрактный пример:

```
import java.sql.*;

public class clob
{
  public static void main(String[] args) {
     try {
       String url = "jdbc:derby:clobberyclob;create=true";

       // Load the driver. This code is not needed if you are using
       // JDK 6, because in that environment the driver is loaded
       // automatically when the application requests a connection.

        Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

        Connection conn = DriverManager.getConnection(url);

        Statement s = conn.createStatement();

       s.executeUpdate(
           "CREATE TABLE documents (id INT, text CLOB(64 K))");

       conn.commit();

       // --- добавить файл

       java.io.File file = new java.io.File("asciifile.txt");

       int fileLength = (int) file.length();

       // - сначала, открываем входной поток

       java.io.InputStream fin = new java.io.FileInputStream(file);

       PreparedStatement ps = conn.prepareStatement(
          "INSERT INTO documents VALUES (?, ?)");

       ps.setInt(1, 1477);

       // - устанавливаем значение входного параметра для input stream
```

```
        ps.setAsciiStream(2, fin, fileLength);
        ps.execute();
        conn.commit();

        // --- reading the columns

        ResultSet rs = s.executeQuery(
            "SELECT text FROM documents WHERE id = 1477");

        while (rs.next()) {
            java.sql.Clob aclob = rs.getClob(1);
            java.io.InputStream ip = rs.getAsciiStream(1);
            int c = ip.read();

            while (c > 0) {
                System.out.print((char)c);
                c = ip.read();
            }

            System.out.print("\n");
            // ...
        }
    } catch (Exception e) {
        System.out.println("Error! "+e);
    }
  }
}
```

Тип данных XML

Тип данных ***XML*** используется для документов *Extensible Markup Language* (XML).

Тип данных ***XML*** использует:

- установку XML документов посредством SQL/XML определения хорошо форматированного значения ***XML(DOCUMENT(ANY))***.
- Временные значения для ***XML(SEQUENCE)***, которые могут не быть хорошо форматированными значениями ***XML(DOCUMENT(ANY))***.

Замечание

Для приложений с *retrieve, update, query* или другим доступом к данным значений XML, приложения должны иметь *классы JAXP parser* и для ***Xalan*** в переменной среды ***classpath***.

Derby сообщит об ошибке, если parser (интерпретатор) или Xalan — не найден. В подобных ситуациях, вам потребуется сделать некоторые шаги, чтобы поместить parser и Xalan в ваш ***classpath***, потому что сторона JDBC, обеспечивающая доступ SQL/XML в Derby, непосредственно не может связать значение XML или вернуть значение XML прямо из результата, установленного JDBC. Поэтому,

вы должны связать полученные данные XML со строкой Java или символьным потоком непосредственно применяя XML операторы XMLPARSE и XMLSERIALIZE, как часть вашего SQL запроса.

XML соотносит:

- Java type: ***None***
- тип Java для значений XML: ***java.sql.SQLXML***. Тем не менее, тип java.sql.SQLXML не доступен Derby;
- JDBC metadata type (java.sql.Types): ***None***
- тип метаданных для значений ***XML - SQLXML***. Тем не менее, тип SQLXML не доступен Derby.

Чтобы принять значения XML из СУБД Derby, используя JDBC, применяйте оператор XMLSERIALIZE в SQL-запросе.

Например:

```
SELECT XMLSERIALIZE (xcol as CLOB) FROM myXmlTable
```

Чтобы принять значение XML используйте метод ***getXXX()***, который соотносит целевой тип сериализации с типом данных CLOB. Чтобы установить значение XML в СУБД Derby, применяя JDBC, используйте оператор XMLPARSE в строке SQL-запроса.

Например:

```
INSERT INTO myXmlTable(xcol) VALUES XMLPARSE(
      DOCUMENT CAST (? AS CLOB) PRESERVE WHITESPACE)
```

Когда вы используете любые методы ***setXXX()***, совместимые с типом String, то применяйте один из следующих методов: *PreparedStatement.setString()* или *PreparedStatement.setCharacterStream()*.

1.5.5 Функции

Функция CURRENT_DATE

CURRENT_DATE возвращает текущую дату; значение не изменится, ее запустить несколько раз в одном запросе:

```
CURRENT_DATE
```

```
или

CURRENT DATE
```

Пример:

```
-- find available future flights:

SELECT * FROM Flightavailability where flight_date > CURRENT_DATE;
```

Функция CURRENT_TIME

CURRENT_TIME возвращает текущее время; возвращаемое значение не изменится, если ее запустить более одного раза в пределах *single statement*.

```
CURRENT_TIME
или
CURRENT TIME
```

Примеры:

```
VALUES CURRENT_TIME
-- или:
VALUES CURRENT TIME
```

Функция CURRENT_TIMESTAMP

CURRENT_TIMESTAMP возвращает текущий timestamp; возвращаемое значение не изменится, если ее запустить более одного раза в пределах *single statement*.

```
CURRENT_TIMESTAMP
или
CURRENT TIMESTAMP
```

Учебное задание

В процессе данного практического занятия, студенту необходимо выполнить следующее самостоятельное упражнение с использованием утилиты ***ij***:

- создать произвольную таблицу данных;
- организовать два SQL-запроса к созданной таблице.

2 Практическое занятие №6 — лабораторная №6. Объектные распределенные системы

Данная глава посвящена объектным распределенным системам. Предполагается, что на предыдущих пяти практических занятиях студент освоил основы базовых средств языка Java, входящих в стандартный пакет J2SE, и готов применить их для программирования конкретных систем.

Учебный материал данного (шестого) практического занятия охватывает теоретическую основу построения объектных распределенных систем, изложенную в учебном пособии [1, глава 3, стр. 98-151]. Студенту необходимо изучить все три подраздела указанной главы и уделить основное внимание подразделу 3.3, описывающему технологию RMI.

Учебная цель данного практического занятия — более полно показать связь технологии RMI с более общей моделью CORBA, предполагая, что отдельные части большой распределенной системы CORBA реализуются средствами языка Java.

Результаты данного практического занятия включаются как составная часть лабораторной работы №6: «*Реализация распределенной системы средствами технологии RMI*».

В целом, данная глава содержит всего лишь один подраздел, который в минимальной степени дополняет учебный материал главы 3 учебного пособия [1]. Такой подход обоснован стремлением уделить больше времени web-технологиям, которые в настоящее время составляют основной тренд развития РВ-сетей.

2.1 Инструментальные средства языка Java для технологии RMI

RMI является собственной технологией языка Java предназначенной для реализации распределенных сетей (РВ-сетей) на основе протоколов JRMP или IIOP. Как показано теорией и примерами [1, подраздел 3.3], новый протокол JRMP не требует генерации заглушек на стороне клиента и скелетонов на стороне сервера. Тем не менее, проектирование большой и сложной РВ-сети может потребовать использовать технологию CORBA, поскольку отдельные части системы могут быть реализованы на языках отличных от Java или содержать элементы, реализованные на протоколе IIOP. В такой ситуации необходимо будет использовать описание интерфейсов системы на языке IDL CORBA и применять утилиту ***rmic*** для генерации заглушек и скелетонов совместимые с протоколами системы CORBA.

В течении данного практического занятия мы рассмотрим:

- краткое описание утилиты rmic, являющееся переводом официальной документации, доступной из среды УПК АСУ командой: ***man rmic***;
- пример генерации интерфейса для технологии CORBA на основе описания интерфейса вызова удалённых методов, обеспеченных средствами описания

интерфейсов языка Java.

2.1.1 Утилита rmic

rmic - генерирует классы-заглушки, скелеты и связки для удалённых объектов, которые используют протокол удалённых методов Java (JRMP) или межорбитальный протокол Интернета (IIOP). Также генерирует язык определения интерфейса группы управления объектами (OMG) (IDL)

```
rmic [опции] имена_определённые_в_пакете
```

где ***опции*** — (см. далее).

Примечание об устаревании: Поддержка статической генерации заглушек и скелетов протокола удалённого метода Java (JRMP) устарела. Oracle рекомендует вместо этого использовать динамически генерируемые заглушки JRMP, исключая необходимость использовать этот инструмент для приложений на основе JRMP. См. Спецификацию на определение ***java.rmi.server.UnicastRemoteObject*** по адресу ***http://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html*** для получения дополнительной информации.

Компилятор ***rmic*** генерирует файлы-заглушки и классы скелета, используя протокол удалённого метода Java (JRMP) и файлы классов-заглушек и связующих (протокол IIOP) для удалённых объектов. Эти файлы классов генерируются из скомпилированных классов языка программирования Java, которые являются классами реализации удалённых объектов. Класс удалённой реализации - это класс, который реализует интерфейс ***java.rmi.Remote***. Имена классов в команде ***rmic*** должны относиться к классам, которые были успешно скомпилированы с помощью команды ***javac***, и должны соответствовать полному пакету. Например, при выполнении команды ***rmic*** для имени файла класса ***HelloImpl***, как показано здесь, создаётся файл ***HelloImpl_Stub.classfile*** в подкаталоге ***hello*** (названный для пакета класса):

```
rmic hello.HelloImpl
```

Скелет для удалённого объекта - это объект на стороне сервера протокола JRMP, у которого есть метод, который отправляет вызовы реализации удалённого объекта.

Связывание для удалённого объекта - это объект на стороне сервера, похожий на скелет, но взаимодействующий с клиентом по протоколу IIOP.

Заглушка — это прокси на стороне клиента для удалённого объекта, который отвечает за передачу вызовов методов для удалённых объектов на сервер, где находится фактическая реализация удалённого объекта. Следовательно, ссылка клиента на удалённый объект на самом деле является ссылкой на локальную заглушку.

По умолчанию команда ***rmic*** генерирует классы-заглушки, которые используют только версию протокола-заглушку 1.2 JRMP, как если бы была указана опция ***-v1.2***. Параметр ***-vcompat*** был установлен по умолчанию в выпусках до 5.0. Используйте опцию ***-iiop*** для генерации классов заглушки и связки для протокола IIOP. Смотрите параметры.

Заглушка реализует только удалённые интерфейсы, а не любые локальные интерфейсы, которые также реализует удалённый объект. Поскольку заглушка JRMP реализует тот же набор удалённых интерфейсов, что и удалённый объект, клиент может использовать встроенные операторы языка программирования Java для приведения и проверки типов. Для протокола IIOP должен использоваться метод ***PortableRemoteObject.narrow***.

ПАРАМЕТРЫ

-bootclasspath path

Переопределяет расположение файлов класса начальной загрузки.

-classpath path

Определяет путь, который команда rmic использует для поиска классов. Этот параметр переопределяет переменную по умолчанию или переменную среды CLASSPATH, когда она установлена. Каталоги разделены двоеточиями. Общий формат для пути: ***.: <Your_path>***, например: ***.: /usr/local/java/classes***.

-d directory

Определяет корневой каталог назначения для сгенерированной иерархии классов. Вы можете использовать эту опцию, чтобы указать каталог назначения для файлов заглушки, скелета и связей. Например, следующая команда помещает классы-заглушки и скелеты, полученные из ***MyClass***, в каталог ***/java/classes/exampleclass***.

```
rmic -d  /java/classes    exampleclass.MyClass
```

Если опция ***-d*** не указана, то поведение по умолчанию такое, как если бы ***-d*** был указан. Иерархия пакетов целевого класса создаётся в текущем каталоге, и в него помещаются файлы-заглушки ***/tie/skeleton***. В некоторых более ранних выпусках команды ***rmic***, если не была указана опция ***-d***, иерархия пакетов не создавалась, и все выходные файлы помещались непосредственно в текущий каталог.

-extdirs path

Переопределяет расположение установленных расширений.

-g

Позволяет генерировать всю информацию отладки, включая локальные переменные. По умолчанию генерируется только информация о номере строки.

-idl

Заставляет команду ***rmic*** генерировать OMG IDL для указанных классов и любых указанных классов. IDL предоставляет чисто декларативный, независимый от языка программирования способ указания API для объекта. IDL используется в качестве спецификации для методов и данных, которые можно записывать и вызывать с любого языка, который обеспечивает привязки CORBA. Это включает Java и C ++ среди других. См. Java IDL: сопоставление IDL с языком Java на ***http://docs.oracle.com/javase/8/docs/technotes/guides/idl/mapping/jidlMapping.html***

Когда используется опция ***-idl***, другие опции также включают:

- Опции ***-always*** или ***-alwaysgenerate*** форсируют регенерацию, даже если существующие ***stubs/ties/IDL*** новее, чем входной класс.

- Опция ***-factory*** использует ключевое слово ***factory*** в сгенерированном IDL.

-IdlModule из ***JavaPackage[.class]toIDLModule*** указывает отображение пакета ***IDLEntity***, например: ***-idlModulemy.module my::real::idlmod***.

-IdlFilefromJavaPackage[.class]toIDLFile указывает отображение файла IDLEntity, например: ***-idlFile test.pkg.X TEST16.idl.***

-iiop

Заставляет команду ***rmic*** генерировать классы-заглушки и связки IIOP, а не классы-заглушки и скелеты JRMP. Класс-заглушка является локальным прокси для удалённого объекта и используется клиентами для отправки вызовов на сервер. Каждый удалённый интерфейс требует класс-заглушку, который реализует этот удалённый интерфейс. Ссылка клиента на удалённый объект является ссылкой на заглушку. Связывающие классы используются на стороне сервера для обработки входящих вызовов и отправки вызовов соответствующему классу реализации. Каждый класс реализации требует связующего класса.

Если вы вызываете команду ***rmic*** с ***-iiop***, то она генерирует заглушки и связи, соответствующие этому соглашению об именах:

_<ImplementationName>_stub.class
_<interfaceName>_tie.class

Когда вы используете опцию ***-iiop***, другие опции также включают:

- Опции ***-always*** или ***-alwaysgenerate*** форсируют регенерацию, даже если существующие ***stubs/ties/IDL*** новее, чем входной класс.

- Параметр ***-nolocalstubs*** означает, что не следует создавать заглушки, оптимизированные для однопроцессных клиентов и серверов.

- Параметр ***-noValueMethods*** должен использоваться вместе с параметром ***-idl***. Опция ***-noValueMethods*** предотвращает добавление методов и инициализаторов значения типа в выдаваемый IDL. Эти методы и инициализаторы являются необязательными для значений типов и генерируются, если в параметре ***-idl*** не указан параметр ***-noValueMethods***.

- Параметр ***-poa*** изменяет наследование с ***org.omg.CORBA_2_3. portable. ObjectImpl*** на ***org.omg.PortableServer.Servant***. Модуль ***PortableServer*** для ***Portable Object Adapter*** (POA) определяет собственный тип ***Servant***. В языке программирования Java тип ***Servant*** сопоставляется с классом ***org.omg.PortableServer.Servant***. Он служит базовым классом для всех реализаций серванта POA и предоставляет ряд методов, которые может вызывать программист приложения, и методы, которые вызываются POA и которые могут быть переопределены пользователем для управления аспектами поведения серванта. На основе спецификации преобразования языка IDG в Java для CORBA V 2.3.1 ptc / 00-01-08.pdf..RE

-J

Используемая с любой командой Java, опция ***-J*** передаёт аргумент, следующий за ***-J*** (без пробелов между -J и аргументом), интерпретатору Java

-keep or -keepgenerated

Сохраняет сгенерированные исходные файлы .java для классов-заглушек, скелетов и связей и записывает их в тот же каталог, что и файлы .class.

-nowarn

Отключает предупреждения. Когда используются параметры ***-nowarn***. Компилятор не выводит никаких предупреждений.

-nowrite

Не записывает скомпилированные классы в файловую систему.

-vcompat (устарело)

Создаёт классы-заглушки и скелеты, совместимые с версиями протокола заглушки JRMP 1.1 и 1.2. Эта опция была по умолчанию в выпусках до 5.0. Сгенерированные классы-заглушки используют версию протокола-заглушки 1.1 при загрузке в виртуальную машину JDK 1.1 и используют версию протокола-заглушки 1.2 при загрузке в виртуальную машину 1.2 (или более позднюю). Сгенерированные классы скелета поддерживают версии протокола заглушки 1.1 и 1.2. Сгенерированные классы являются относительно большими, чтобы поддерживать оба режима работы. Примечание. Эта опция устарела. Смотри описание.

-verbose

Заставляет компилятор и компоновщик распечатывать сообщения о том, какие классы компилируются и какие файлы классов загружаются.

-v1.1 (устарело)

Генерирует классы заглушек и скелетов только для версии протокола заглушки 1.1 JRMP. Опция -v1.1 полезна только для генерации классов-заглушек, совместимых с сериализацией с существующими статически развёрнутыми классами-заглушками, которые были сгенерированы командой rmic из JDK 1.1 и которые не могут быть обновлены (и динамическая загрузка классов не используется) , Примечание. Эта опция устарела. Смотри описание.

-v1.2 (устарело)

(По умолчанию) Генерирует классы-заглушки только для версии протокола-заглушки 1.2 JRMP. Скелетные классы не генерируются, поскольку скелетные классы не используются с версией протокола-заглушки 1.2. Сгенерированные классы-заглушки не работают, когда они загружаются в виртуальную машину JDK 1.1. Примечание. Эта опция устарела.

ПЕРЕМЕННЫЕ ОКРУЖАЮЩЕЙ СРЕДЫ:

CLASSPATH

Используется для предоставления системе пути к пользовательским классам. Каталоги разделены двоеточиями, например: ***.:/usr/local/java/classes***.

2.1.2 Преобразование интерфейсов RMI в описание IDL CORBA

Рассмотрим интерфейс RMI-приложения, реализованный в проекте ***proj12*** учебного пособия [1, пункт 3.3.1, рисунок 3.18]. Его исходный текст имеет вид, представленный ниже:

```
package asu.rvs.rmi;

public interface RmiPad extends java.rmi.Remote
{
      // Объявление методов:

      // метод, проверяющий наличие соединения с БД;
      public boolean isConnect() throws
            java.rmi.RemoteException;

      // метод, получающий содержимое таблицы notepad БД
      // exampleDB в виде списка текстовых строк;
      public String[] getList() throws
            java.rmi.RemoteException;

      // метод, добавляющий текст к содержимому
      // таблицы notepad БД; также учитывается
      // уникальность ключа key;
      public int setInsert(int key, String str) throws
            java.rmi.RemoteException;

      // метод, удаляющий по заданному ключу key запись из
      // таблицы notepad БД;
      public int setDelete(int key) throws
```

```
        java.rmi.RemoteException;
}
```

Если необходимо реализовать приложение по технологии CORBA и с таким интерфейсом, то необходимо описать интерфейс типа ***orbpad.idl***, как это было сделано в проекте ***proj10*** (см. [1, пункт 3.2.4, рисунок 3.13]). В условиях, когда проект большой и сложный, описание интерфейса и его сопровождение тоже становится сложным, что приводит к множеству ошибок. В таких случаях, сопровождать интерфейсы языка Java — гораздо проще, а утилита ***rmic*** позволяет генерировать описание IDL CORBA на основе интерфейса Java, что улучшает качество разработки проектов.

Для примера, создадим каталог ***$HOME/src/proj12***, в который перенесём описание интерфейса ***RmiPad.java***, с учётом его оператора ***package***. Далее, в этом каталоге выполним команду:

```
rmic  -idl  asu.rvs.rmi.RmiPad
```

В результате указанного действия будет создан файл ***RmiPad.idl***, а также дополнительные каталоги и файл, необходимый для проекта технологии CORBA.

Учебное задание

Провести экспериментальные действия с файлом ***RmiPad.java***, указанные выше, и дать им подробное описание.

3 Web-технологии распределенных систем

Данная глава посвящена web-технологиям, которые широко используются в современных распределенных системах. Предполагается, что студент достаточно хорошо освоил основы базовых средств языка Java, входящих в стандартный пакет J2SE, и готов к изучению технологий пакета J2EE.

Учебный материал данной главы охватывает три последних практических занятия, теоретическая основа которых изложена в учебном пособии [1, глава 4, стр. 152-197]. Студенту необходимо изучить все три подраздела указанной главы и тогда приступать к материалу практических занятий.

В целом, каждый подраздел данной главы посвящён одному практическому занятию, учебный материал которого входит составной частью отдельной лабораторной работы, а их тематика посвящена следующим вопросам:

- классы технологии Java-сервлетов;
- HTML и технология JSP-страниц;
- технология шаблона MVC.

Содержательная тематика практических занятий приведена в материале каждого подраздела. Тем не менее, все приведённые примеры базируются на примерах, сосредоточенных в проекте ***proj14*** среды разработки Eclipse EE.

3.1 Практическое занятие №7 — лабораторная №7. Классы технологии Java-сервлетов

Общее описание технологии сервлетов изложено в учебнике [1, подраздел 4.3]. В пунктах 4.3.1-4.3.3 этого подраздела кратко рассмотрены:

- описание абстрактных классов ***Servlet*** и ***HttpServlet***;
- инструментальные средства контейнера сервлетов — ***Apache Tomcat;***
- базовые возможности класса ***RequestDispatcher***, обеспечивающего обработку запросов, поступающих от программ-клиентов (браузеров) и принятых методами ***doGet(...)*** или ***doPost(...)*** сервлетов сервера Apache Tomcat.

Перечисленный выше материал учебника [1] составляет основу лабораторной работы №7: «*Технология сервлетов на базе сервера Apache Tomcat*». Учебный же материал этого практического занятия конспективно показывает основную технологическую траекторию прохождения запроса к web-серверу и формирование ответа программе-клиента (браузеру).

Учебная цель данной (седьмой) практической работы — более подробное описание обработки сервером запросов, поступающих от программ клиентов, поскольку они могут содержать некоторый контекст, который может существенно повлиять на схему и алгоритм формирования ответа.

Исходя из поставленной цели, в пунктах 3.1.1 и 3.1.2 данного подраздела

рассмотрены два аспекта обработки сервером входящего запроса:

- общий аспект, связанный с передачей самого запроса к серверу;
- частный аспект, связанный с семантикой запроса.

3.1.1 Общая обработка запроса

Запросы к серверу Apache Tomcat формируются в приложении браузера, который использует средства языка HTML, обеспечивающие ввод необходимых для запроса данных. Базовым средством языка HTML, поддерживающим такой ввод является конструкция:

<FORM action="URL" method="get или post"
accept-charset="кодировка">
…
</FORM>

Внутри этой конструкции определяются поля ввода текста разных типов, задаваемых тегами: ***<INPUT …>***.

Чтобы конкретизировать наши рассуждения, рассмотрим отдельный пример в рамках проекта ***proj14***, включающий:

- HTML-страницу с именем ***post1a.html***, содержащую форму запроса;
- сервлет ***Example14a.java***, обслуживающий запрос на стороне сервера.

За основу страницы ***post1a.html*** возьмём код страницы ***post1.html***, дополнив его скрытым полем и формой, обращающейся к сервлету ***Example14a***. В результате, ее текст будет иметь вид, представленный на листинге 3.1.

Листинг 3.1 — Исходный текст файла post1a.html для сервлета Example14a

```
<html>
<head>
<meta charset="UTF-8" />
<title>Практика №7</title>
</head>
<body>
      <hr>
      <b>Запрос к таблице ведения записей</b>
      <hr>
      <form action="Example14a" method="post" accept-charset="UTF-8">
        <p> Введи ключ :
            <input type="text" size="10" name="key">
        </p>
        <p> Введи текст: <br>
            <textarea rows="5" cols="40" name="text"></textarea>
        </p>
        <p>
            <input type="hidden" name="test" value="Русский текст">
            <input type="submit">
        </p>
      </form>
```

```
    <hr>
</body>
</html>
```

В тексте этой страницы следует обратить внимание на используемую страницей кодировку символов, которая отражена в двух тегах: ***<META>*** и ***<FORM>***. Первая из них предназначена для поисковых систем, а вторая указывает на кодировку последующего ввода данных. Более того, станица использует скрытое поле с именем ***test***, содержащее русскоязычный текст, который также будет передаваться в качестве запроса на сервер.

Важное значение также имеет кодировка самого текста листинга 3.1, которая соответствует кодировке символов используемой редактором текста. Здесь необходимо напомнить, что редакторы ПО Linux использует кодировку ***UTF-8***, а редакторы MS Windows — ***Cp1251***, требующие обозначение в HTML-страницах как кодировка ***windows-1251***.

Поскольку файл ***post1a.html*** подготовлен в редакторе Eclipse EE ОС Linux, то любой правильно настроенный браузер в любой ОС — должен правильно отображать полученную страницу.

Давайте проверим — это. Результат показан на рисунке 3.1.

Рисунок 3.1 — Результат отображения post1a.html во встроенном браузере среды разработки Eclipse EE

Как видим, результат — негативный, но если мы отобразим страницу в браузере Firefox, то результат будет иной (см. рисунок 3.2).

Можно было бы заключить, что встроенный браузер среды Eclipse EE реализован не очень качественно и, возможно, - это так и есть, но давайте обратимся к тексту сервлета ***Example14a***, обслуживающего этот запрос. Его содержимое показано на листинге 3.2.

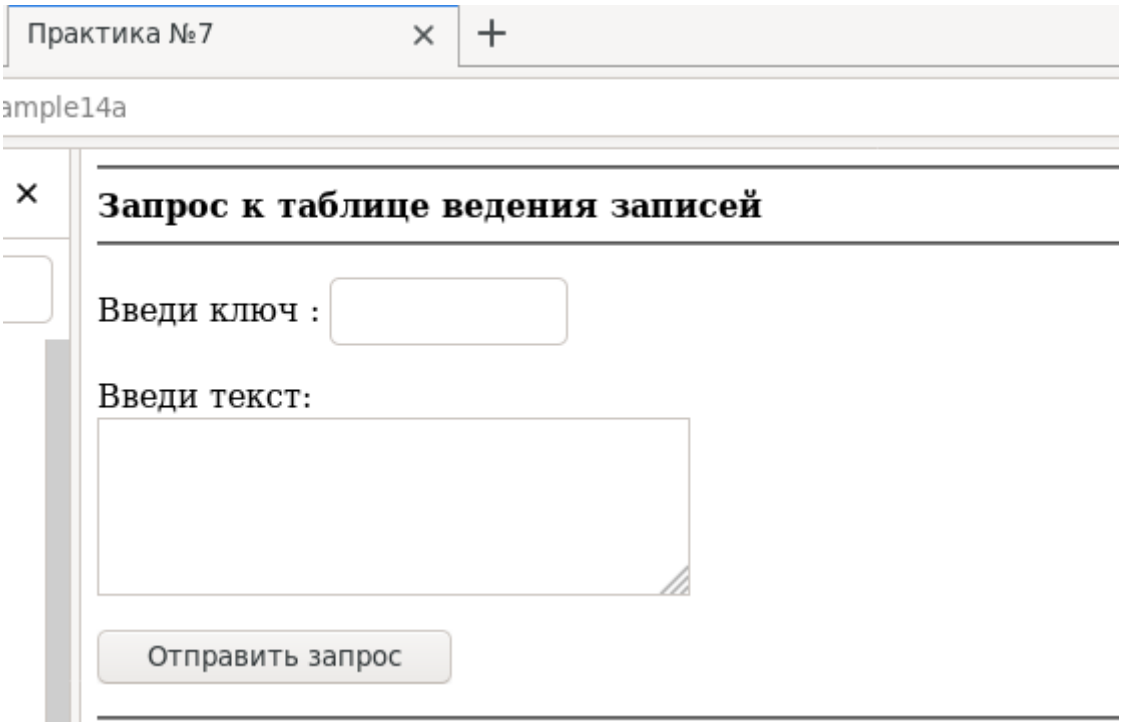

Рисунок 3.2 — Результат отображения post1a.html в браузере Firefox

Листинг 3.2 — Исходный текст файла сервлета Example14a.java

```
package rvs.servlets;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class Example14a
 */
@WebServlet("/Example14a")
public class Example14a extends HttpServlet {
    private static final long serialVersionUID = 1L;

    // Стандартные методы класса Example14a.

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Example14a() {
        super();
    }

      /**
       * @see HttpServlet#doGet(HttpServletRequest request,
       *          HttpServletResponse response)
       */
```

```
protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
{
      /**
       * Явная установка кодировок объектов запроса и ответа.
       * Стандартная установка контекста ответа.
       */
      //request.setCharacterEncoding("UTF-8");
      //response.setCharacterEncoding("UTF-8");
      response.setContentType("text/html");
      System.out.println("Метод doGet() - принял запрос...");
      System.out.println("request.getCharacterEncoding()  = "
                  + request.getCharacterEncoding());
      System.out.println("response.getCharacterEncoding()  = "
                  + response.getCharacterEncoding());

      /**
       * Стандартное подключение ресурса сервлета.
       */
      RequestDispatcher disp =
                  request.getRequestDispatcher("/WEB-INF/post1a.html");

      System.out.println("request.getCharacterEncoding()  = "
                  + request.getCharacterEncoding());
      System.out.println("response.getCharacterEncoding()  = "
                  + response.getCharacterEncoding());

      disp.forward(request, response);
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 *          HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
{
      /**
       * Явная установка кодировок объектов запроса и ответа.
       * Стандартная установка контекста ответа.
       */
      System.out.println("Метод doPost() - принял запрос...");
      System.out.println("request.getCharacterEncoding()  = "
                  + request.getCharacterEncoding());
      System.out.println("response.getCharacterEncoding()  = "
                  + response.getCharacterEncoding());

      //request.setCharacterEncoding("UTF-8");
      //response.setCharacterEncoding("UTF-8");
      response.setContentType("text/html");

      System.out.println("key  = "
                  + request.getParameter("key"));
      System.out.println("text = "
                  + request.getParameter("text"));
      System.out.println("test = "
                  + request.getParameter("test"));
      /**
       * Стандартное подключение ресурса сервлета.
       */
```

```
            RequestDispatcher disp =
                        request.getRequestDispatcher("/WEB-INF/post1a.html");

            System.out.println("request.getCharacterEncoding()  = "
                        + request.getCharacterEncoding());
            System.out.println("response.getCharacterEncoding()  = "
                        + response.getCharacterEncoding());

            disp.forward(request, response);
      }

}
```

Мы видим, что текст сервлета ***Example14a*** содержит только три стандартных метода: конструктор ***Example14a(...)***, метод ***doGet(...)*** и метод ***doPost(...)***. В каждом из обрабатывающих методов печатается установленная по умолчанию кодировка объектов ***request*** и ***response***, - до и после создания объекта типа ***RequestDispatcher***.

Теперь, если мы посмотрим, что отображает метод ***doGet(...)***, который обработал запросы от встроенного браузера и браузера Firefox, то увидим (см. рисунок 3.3), что:

- объект запроса ***request*** вообще не отображает кодировку символов;
- объект ответа ***response*** отображает кодировку ***ISO-8859-1***, не соответствующую ни тегу ***<META>***, ни тегу ***<FORM>***.

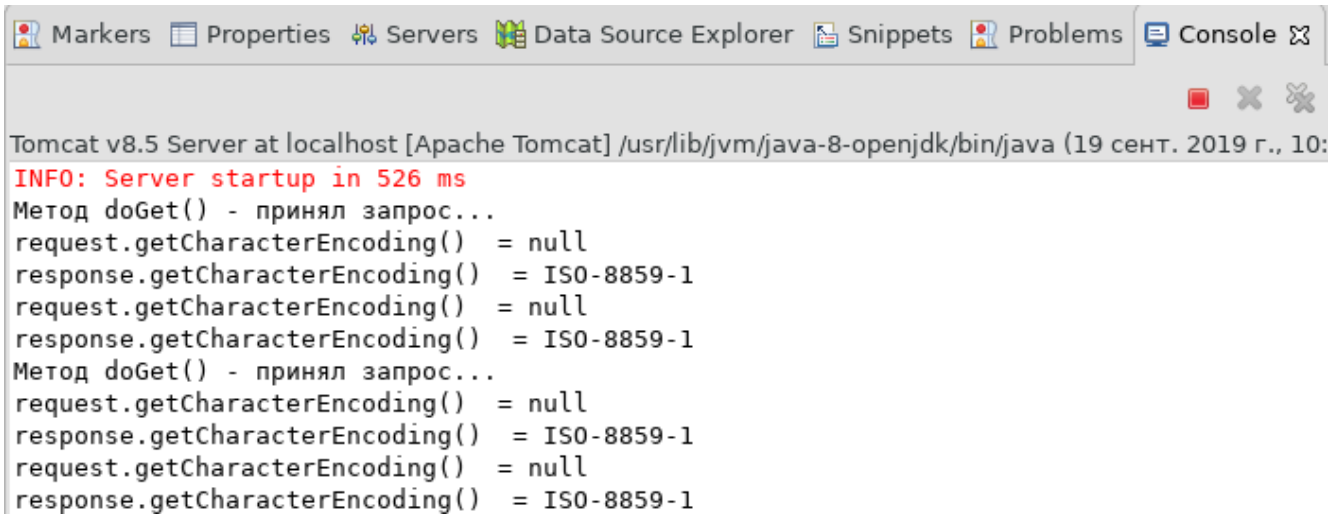

Рисунок 3.3 — Отображения кодировок объектов request и response методом doGet(…) сервлета Example14a

Теперь продолжим эксперимент и заполним сначала форму рисунка 3.1, вставив в оба поля ввода слово «*привет*» и отправим запрос серверу, а затем проделаем тоже самое с формой рисунка 3.2.

Результатом будет то, что изображения форм в браузерах не изменится, хотя запросы будет обрабатывать метод сервера — ***doPost(...)***. а вывод будет гораздо обширнее, что показано на рисунке 3.4:

- характеристики кодировок объектов ***request*** и ***response*** — те же, что и у метода ***doGet(...)***;

- дополнительно показаны принятые сервером значения параметров с именами: ***key***, ***text*** и ***test***.

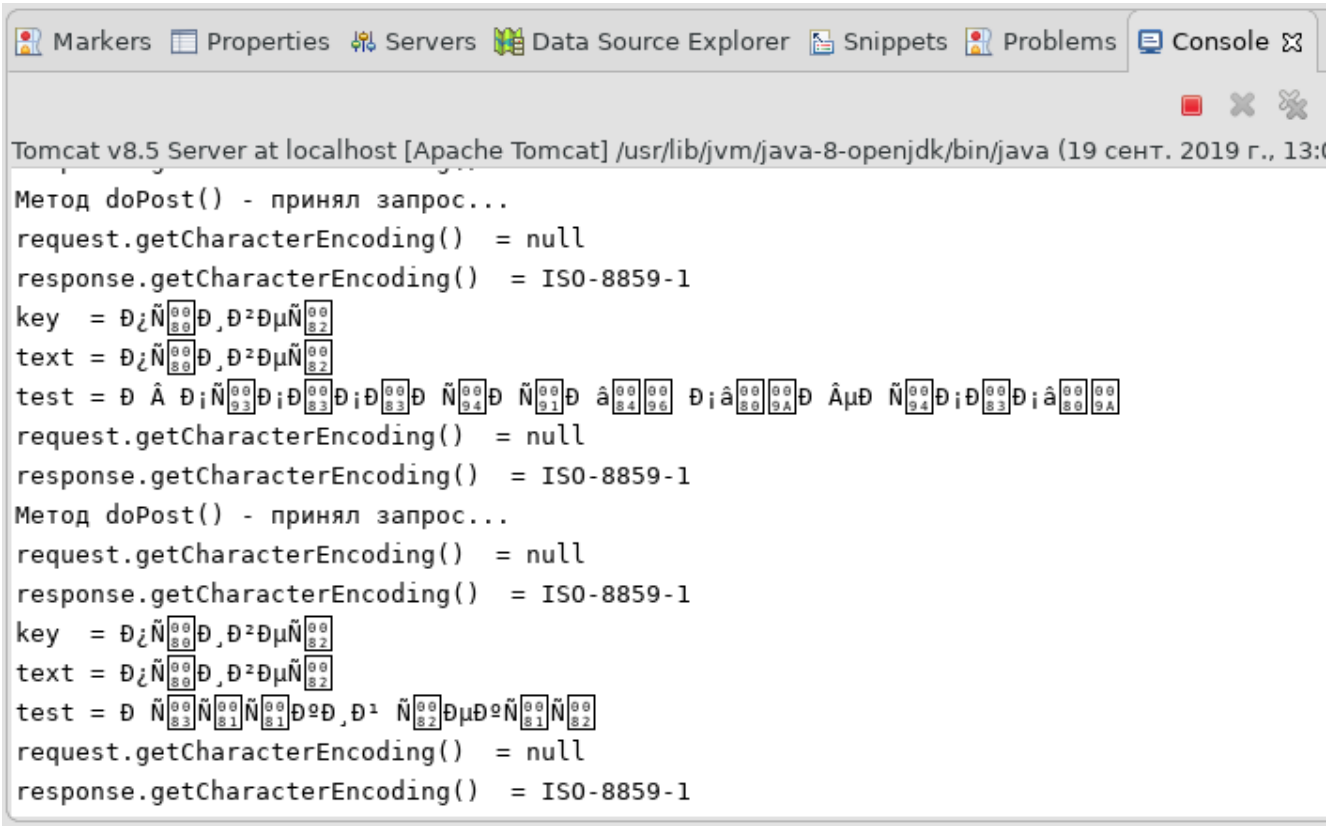

Рисунок 3.4 — Отображения кодировок объектов request и response методом doPost(…) сервлета Example14a

Таким образом, мы наглядно убедились, что правильное отображение страницы HTML в окне браузера не гарантирует правильную передачу запросов в сервер Apache Tomcat. Программист обязан убедиться, что параметры к серверу передаются правильно и в нужной кодировке. Для этого, можно воспользоваться приёмом, который демонстрирует сервлет ***Example14a.java***. В частности, для нашего случая:

- проблема правильного отображения символов в окне браузера решается использованием метода: ***response.setCharacterEncoding("UTF-8")***;
- проблема правильного чтения переданных параметров решается использованием метода: ***request.setCharacterEncoding("UTF-8")***.

Учебное задание

Провести экспериментальные действия по правильному отображению страницы ***post1a.html*** и правильной передачи параметров в рассмотренном выше примере.

Создать в среде MS Windows страницу ***post1b.html***, идентичную по тексту странице ***post1a.html***, разместить ее в каталоге ***WEB-INF*** проекта ***proj14*** и провести соответствующую настройку сервлета ***Example14a.java***.

3.1.2 Обработка контекста запроса

Типичное взаимодействие браузера с сервером Apache Tomcat осуществляется через методы ***doGet(..)*** и ***doPost(...)***:

- метод ***doGet(...)*** используется по умолчанию и обычно не содержит параметров запроса, поскольку это требует знания самой структуры запроса; как правило, указывается только адрес: http://localhost:8080/proj14/Example14a, а принятая браузером страница содержит форму, в которой запрос представлен в виде текстовых полей с нужными подсказками, а также в форме указывается, что запрос передаётся методу сервера ***doPost(...)***;
- метод ***doPost(...)*** наоборот предназначен для формирования и передачи параметров запроса, которые сервер должен принять, провести анализ и выбрать алгоритм ответа.

Поскольку метод ***doGet(...)*** задаёт только начальную форму запроса, то основная нагрузка на обработку запроса ложится на метод ***doPost(...)***. В нашем примере (листинг 3.1), форма запроса содержит три поля:

- поле ***key*** должно содержать целочисленное значение, а метод ***doPost(...)*** должен интерпретировать его в трёх вариантах: пустое поле — завершение работы, целое число — запрос на удаление или добавление записи;
- поле ***text*** — произвольный набор символов, но если оно пустое, то поле ***key*** используется для удаления записи;
- скрытое поле ***test*** имеет начальное значение «***Русский текст***» и имеет некоторую дополнительную семантику, например, сервер может сравнить принятое значение с заданным и протестировать правильность передачи запросов; обычно, скрытые поля вводятся для разрешения каких-либо неоднозначностей; например, если страница содержит несколько отдельных форм, то серверу нужно определить какая из форм была активирована.

Язык HTML содержит множество типов полей ввода. Базовые типы этих полей представлены в следующей таблице:

Тип	**Описание**
button	Кнопка.
checkbox	Флажки. Позволяют выбрать более одного варианта из предложенных.
file	Поле для ввода имени файла, который пересылается на сервер.
hidden	Скрытое поле. Оно никак не отображается на веб-странице.
image	Поле с изображением. При нажатии на рисунок данные формы отправляются на сервер.
password	Обычное текстовое поле, но отличается от него тем, что все символы показываются звёздочками. Предназначено для того, чтобы никто не подглядел вводимый пароль.
radio	Переключатели. Используются, когда следует выбрать один вариант из

	нескольких предложенных.
reset	Кнопка для возвращения данных формы в первоначальное значение.
submit	Кнопка для отправки данных формы на сервер.
text	Текстовое поле. Предназначено для ввода символов с помощью клавиатуры.

В языке HTML 5 добавлены следующие значения типов полей, хотя не все браузеры их поддерживают:

Тип	Описание
color	Виджет для выбора цвета.
date	Поле для выбора календарной даты.
datetime	Указание даты и времени.
datetime-local	Указание местной даты и времени.
email	Для адресов электронной почты.
number	Ввод чисел.
range	Ползунок для выбора чисел в указанном диапазоне.
search	Поле для поиска.
tel	Для телефонных номеров.
time	Для времени.
url	Для веб-адресов.
month	Выбор месяца.
week	Выбор недели.

В общем случае, разработчик должен решить как контролировать правильность введённых параметров: на стороне клиента, на стороне сервера или комбинированным способом. Более того, сам контроль и его результаты существенно зависят от решаемой задачи.

В пределах нашего примера, мы проведём простейший контроль, состоящий в анализе правильности получения сервером параметров ***key*** и ***test***:

- если параметры заданы правильно, то клиенту возвращается страница листинга 3.1 ***post1a.html***;
- иначе — возвращается страница ***end.html***, представленная на листинге 3.3.

Листинг 3.3 — Исходный текст файла end.html для сервлета Example14a

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Спасибо!</title>
</head>
<body>
      <hr>
```

```
        <b>Спасибо, что воспользовались нашим приложением!</b>
        <hr>
        <hr>
        <a href="http://localhost:8080/proj14/Example14a">
                    Повторый запуск сервлета Example14a
        </a>
        <hr>
</body>
</html>
```

В нашем примере, основной контроль связан с семантикой строки ***key***, что влияет на дальнейший предполагаемый запрос к базе данных:

- строка должна отображать целое число больше нуля, тогда параметр задаёт правильное значение;
- если строка ***key*** — пустая или отображает не целое число больше нуля, то она — неправильная и мы завершаем программу.

Для анализа строки ***key*** добавим в сервлет ***Example14a*** метод ***testKey(...)***, показанный на листинге 3.4.

Листинг 3.4 — Исходный текст метода testKey(...) для сервлета Example14a

```
/**
 * Метод анализа параметра key.
 */
protected boolean testKey(String key)
{
      if(key.length() == 0)
            return false;
      try
      {
            if(Integer.parseInt(key) > 0)
            {
                  System.out.println("key -правильное число");
                  return true;
            }
            return false;
      }
      catch(NumberFormatException e) {
            return false;
      }
}
```

Теперь преобразуем метод ***doPost(...)*** сервлета ***Example14a***, как показано на листинге 3.5.

Листинг 3.5 — Исходный текст метода doPost(...) для сервлета Example14a

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 *          HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
```

```
	{
		/**
		 * Сохраненные значения параметров запросов
		 * и их флаги.
		 */
		String key, text, test;
		boolean fkey  = true;
		boolean ftext = true;
		boolean ftest = true;

		/**
		 * Явная установка кодировок объектов запроса и ответа.
		 * Стандартная установка контекста ответа.
		 */
		System.out.println("\nМетод doPost() - принял запрос...");
		request.setCharacterEncoding("UTF-8");
		response.setCharacterEncoding("UTF-8");
		response.setContentType("text/html");

		/**
		 * Чтение и анализ параметров.
		 */
		key  = request.getParameter("key");
		System.out.println("key  = " + key);
		fkey = testKey(key);

		text = request.getParameter("text");
		System.out.println("text = " + text);

		test = request.getParameter("test");
		System.out.println("test = " + test);
		if(!test.equals("Русский текст"))
			ftest = false;

		/**
		 * Подключение ресурса сервлета по выбору.
		 */
		RequestDispatcher disp;
		if(fkey & ftext & ftest)
			disp = request.getRequestDispatcher("/WEB-INF/post1a.html");
		else
			disp = request.getRequestDispatcher("/WEB-INF/end.html");

		disp.forward(request, response);
	}
}
```

Учебное задание

Разобраться с примерами листингов 3.3-3.5.

Реализовать, в пределах проекта ***proj14***, указанные изменения в тестовом сервлете ***Example14a.java***.

3.2 Практическое занятие №8 — лабораторная №8. HTML и технология JSP-страниц

Описание технологии JSP-страниц изложено в учебнике [1, пункт 4.3.4, стр. 179-188]. Этот материал учебника [1] составляет основу лабораторной работы №8: «Технология JSP для формирования динамических html-страниц».

Учебный материал данного (восьмого) практического занятия дополняет материал учебника [1] в плане обмена информацией между JSP-страницами и сервлетами.

В целом мы понимаем, что JSP-страницы могут содержать кроме текста языка HTML:

- вставки текста на языке Java, которые обрабатываются на стороне сервера;
- вставки текста на других языках, обычно JavaScript, которые будут изменять исходную страницу на стороне клиента (в браузере).

Второй из названных вариантов мы рассматривать не будем, поскольку он выходит за рамки данного учебного курса, вот вставки на языке Java и позволяют нам создавать *динамические web-страницы* практически любой сложности. Соответственно, появляется возможность перенести часть обработки клиентского запроса из текста сервлета в текст JSP-страницы.

В общем случае, обработка запросов в среде JSP-страницы осуществляется на основе двух источников информации:

- объектов ***request***, ***response***, ***session*** и ***out***, доступных странице после ее вызова диспетчером запросов;
- объектов-атрибутов, которые подключаются сервлетом или JSP-страницей к объекту ***request***.

Естественно возникает общий вопрос: «*Что обрабатывать в методах сервлета, а что — в JSP-страницах*»?

Общего ответа не существует, поскольку он связан с особенностями решения прикладной задачи, но общие правила — следующие:

- сервлеты, ориентированные на использование языка Java — более приспособлены к анализу данных и могут использовать различные программные расширения в виде библиотек классов;
- JSP-страницы ориентированы на графическое представление информации средствами языка HTML, поэтому масштабный и многовариантный анализ на языке Java для них — нежелателен; лучше, если в JSP-страницу поступают уже обработанные данные, которые можно поместить в нужные места.

В предыдущем пункте (практическое занятие №7) был рассмотрен вопрос правильного чтения параметров запросов сервлетов и правильного отображения HTML-страниц в браузерах клиентов. Продолжим рассмотрение наших примеров проекта ***proj14***, выделив следующие аспекты:

- адекватная установка кодировки символов в ответах сервлета;

- передача атрибутов JSP-страницам.

3.2.1 Установка кодировки символов объекта response

По умолчанию, все сервлеты сервера Apache Tomcat для объекта ***response*** используют кодировку ISO-8859-1. С другой стороны очевидно, что эту кодировку можно устанавливать в самой JSP-странице на этапе ее программирования, поскольку объект ***response*** доступен этой странице. Рассмотрим в рамках проекта ***proj14*** новый сервлет ***Example14b.java***, который доверяет такую установку своим страницам. Очевидно, что в таком случае сервлет не должен напрямую вызывать страницы HTML, а обязан обращаться только к страницам JSP.

С другой стороны, в пределах занятия №7 мы использовали специальную страницу ***end.html***, которая демонстрировала завершение работы с сервером и предлагала новое соединение. Очевидно, что подобная страница могла бы быть использована в методе ***doGet(...)*** для начала работы с сервлетом, для чего должен быть изменён только текст заголовка. В целом такой подход — вполне приемлем, поскольку отсутствует необходимость на каждый нюанс запроса создавать новую HTML-страницу.

Для реализации указанных изменений, создадим сервлет ***Example14b.java*** на основе текста сервлета ***Example14a.java***, а затем — все вызовы HTML-страниц заменим обращением к JSP-странице ***request.jsp***, показанной на листинге 3.6.

Листинг 3.6 — Исходный текст файла request.jsp для сервлета Example14b

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Практика №8</title>
</head>
<body>
      <%-- Устанавливаем кодировку объекта response
             и типа контента --%>
      <%
            response.setCharacterEncoding("UTF-8");
            response.setContentType("text/html");
      %>
      <hr>
      <%-- Определение метода, вызвавшего страницу --%>
      <%
            String method = request.getMethod();
            out.println("Метод: " + method
                  + "<hr>");

            if(method.equals("POST"))
            { %>

            <b>Спасибо, что воспользовались нашим приложением!</b>
            <hr>
                  <a href="http://localhost:8080/proj14/Example14b">
                        Подключение к сервлету Example14b
```

```
                </a>
            <hr>
        <%  } %>

        <%-- Обработка метода doGet(...) --%>
        <%
        if(request.getMethod().equals("GET"))
        {
            if(request.getParameter("new") == null)
            {
            %>
                <b>Мы рады вновь увидеть Вас!</b>
            <hr>
            <a href="http://localhost:8080/proj14/Example14b?new=">
                 Подключение к сервлету Example14b
            </a>
            <hr>

            <%}else{ %>

                <b>Введите Ваш запрос:</b>
            <hr>
            <form action="Example14b" method="post" accept-charset="UTF-8">
              <p> Введи ключ :
                  <input type="text" size="10" name="key">
              </p>
              <p> Введи текст: <br>
                  <textarea rows="5" cols="40" name="text"></textarea>
              </p>
              <p>
                  <input type="hidden" name="test" value="Русский текст">
                  <input type="submit">
              </p>
            </form>
            <hr>
        <%  }
        }
        %>

</body>
</html>
```

Исходный текст сервлета ***Example14b.java*** представлен на листинге 3.7.

Листинг 3.7 — Исходный текст сервлета Example14b

```
package rvs.servlets;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class Example14b
 */
@WebServlet("/Example14b")
```

```
public class Example14b extends HttpServlet {
      private static final long serialVersionUID = 1L;

      // Стандартные методы класса Example14a.


    /**
     * @see HttpServlet#HttpServlet()
     */
    public Example14b() {
        super();
    }

      /**
       * @see HttpServlet#doGet(HttpServletRequest request,
       *          HttpServletResponse response)
       */
      protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                              throws ServletException, IOException
      {
            /**
             * Явная установка кодировок объектов запроса и ответа.
             * Стандартная установка контекста ответа.
             */
            request.setCharacterEncoding("UTF-8");
            System.out.println("\nМетод doGet() - принял запрос...");
            System.out.println("request.getCharacterEncoding()   = "
                        + request.getCharacterEncoding());
            System.out.println("response.getCharacterEncoding()  = "
                        + response.getCharacterEncoding());

            /**
             * Стандартное подключение ресурса сервлета.
             */
            RequestDispatcher disp =
                        request.getRequestDispatcher("/WEB-INF/request.jsp");

            disp.forward(request, response);

            System.out.println("request.getCharacterEncoding()   = "
                        + request.getCharacterEncoding());
            System.out.println("response.getCharacterEncoding()  = "
                        + response.getCharacterEncoding());
      }

      /**
       * Метод анализа параметра key.
       */
      protected boolean testKey(String key)
      {
            if(key.length() == 0)
                  return false;
            try
            {
                  if(Integer.parseInt(key) > 0)
                  {
                        System.out.println("key -правильное число");
                        return true;
                  }
                  return false;
            }
```

```
            catch(NumberFormatException e) {
                  return false;
            }
      }

      /**
       * @see HttpServlet#doPost(HttpServletRequest request,
       *          HttpServletResponse response)
       */
      protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
                        throws ServletException, IOException
      {
            /**
             * Сохраненные значения параметров запросов
             * и их флаги.
             */
            String key, text, test;
            boolean fkey  = true;
            boolean ftext = true;
            boolean ftest = true;

            /**
             * Явная установка кодировок объектов запроса и ответа.
             * Стандартная установка контекста ответа.
             */
            System.out.println("\nМетод doPost() - принял запрос...");
            request.setCharacterEncoding("UTF-8");

            /**
             * Чтение и анализ параметров.
             */
            key  = request.getParameter("key");
            System.out.println("key  = " + key);
            fkey = testKey(key);

            text = request.getParameter("text");
            System.out.println("text = " + text);

            test = request.getParameter("test");
            System.out.println("test = " + test);
            if(!test.equals("Русский текст"))
                  ftest = false;

            /**
             * Подключение ресурса сервлета по выбору.
             */
            RequestDispatcher disp;
            if(fkey & ftext & ftest)
                  disp = request.getRequestDispatcher("/WEB-INF/request.jsp");
            else
                  disp = request.getRequestDispatcher("/WEB-INF/request.jsp");

            disp.forward(request, response);
      }
}
```

Учебное задание

Реализовать указанные изменения в тестовом сервлете ***Example14b.java***.

3.2.2 Передача атрибутов JSP-страницам

Пример предыдущего пункта, реализующий JSP-страницу (см. листинг 3.6), показывает, что даже несложная логика может сделать код страницы достаточно «*запутанным*» и трудно читаемым. С другой стороны, логику сообщений, отображаемых в JSP-страницах, гораздо проще реализовать в сервлете, поскольку его код не содержит операторов языка HTML, а также уже разделен на два метода, что позволяет статически учесть формируемые сообщения.

Объект сервлета ***request*** имеет два метода:

```
Object getAttribute(String name);
void   setAttribute(String name, Object obj);
```

которые позволяют устанавливать и читать произвольные объекты по имени. Это свойство ***request*** можно с успехом использовать для передачи значений в сервлет, что позволяет во многих случаях избежать управляющих операторов языка Java в JSP-странице.

Чтобы показать эти возможности, проведём анализ отображаемых элементов страницы ***request.jsp***, представленной на листинге 3.6:

- первая часть страницы представлена скриплетом, устанавливающим кодировку символов и тип контента формируемой страницы; эту часть можно рассматривать как постоянную часть шаблона, который должен присутствовать в каждой странице;
- вторая часть демонстрирует чтение и отображение метода сервлета, вызвавшего страницу; она введена исключительно для демонстрационных целей и обычно не отображается в приложениях;
- третья часть связана с общим приветствием, которое имеет три варианта: одно - для метода POST и два — для метода GET; очевидно они могут быть сформированы в сервлете и переданы в JSP-страницу, например, как атрибут с именем ***Title2***;
- четвёртая часть относится к адресу сервлета, который может показываться или нет; эту часть можно передавать, например, как атрибут ***Address2***;
- пятая и последняя часть относится к форме запроса, которая содержит достаточно объёмный и статичный текст языка HTML; основная особенность этой части — разрешение на демонстрацию формы, которая успешно может быть представлена логическим атрибутом ***isForm***.

Исходя из приведённых соображений, JSP-страница ***request.jsp*** может быть преобразована в страницу ***request1b.jsp***, которая представлена на листинге 3.8.

Хорошо видно, что структура новой JSP-страницы выгодно отличается от старой структуры, поскольку она приняла последовательный линейный вид, кроме последней (пятой части), в которой применён управляющий оператор ***if(...)*** языка Java.

Листинг 3.8 — Исходный текст файла request1b.jsp для сервлета Example14b

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Практика №8б</title>
</head>
<body>
      <%-- Устанавливаем кодировку объекта response
             и типа контента --%>
      <%
            response.setCharacterEncoding("UTF-8");
            response.setContentType("text/html");
      %>
      <hr>
      <%-- Определение метода, вызвавшего страницу --%>
      <%
            String method = request.getMethod();
            out.println("Метод: " + method);
      %>

      <%-- Третья часть страницы: атрибут Title2 --%>
            <hr><b>
            <%= request.getAttribute("Title2") %>
            </b><hr>

      <%-- Четвертая часть страницы: атрибут Address2 --%>

            <%= request.getAttribute("Address2") %>

      <%-- Пятая часть страницы: атрибут isForm --%>
      <%
      if((boolean)request.getAttribute("isForm"))
      {
      %>
            <form action="Example14b" method="post" accept-charset="UTF-8">
              <p> Введи ключ :
                  <input type="text" size="10" name="key">
              </p>
              <p> Введи текст: <br>
                  <textarea rows="5" cols="40" name="text"></textarea>
              </p>
              <p>
                  <input type="hidden" name="test" value="Русский текст">
                  <input type="submit">
              </p>
            </form>
            <hr>
      <%
      }
      %>

</body>
</html>
```

Соответствующие модификации методов ***doGet(...)*** и ***doPost(...)*** приведены на отдельных листингах 3.9 и 3.10.

Листинг 3.9 — Исходный текст метода doGet(...) сервлета Example14b

```
/**
 * @see HttpServlet#doGet(HttpServletRequest request,
 *          HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
{
      /**
       * Явная установка кодировок объектов запроса и ответа.
       * Стандартная установка контекста ответа.
       */
      request.setCharacterEncoding("UTF-8");
      System.out.println("\nМетод doGet() - принял запрос...");
      System.out.println("request.getCharacterEncoding()   = "
                  + request.getCharacterEncoding());
      System.out.println("response.getCharacterEncoding()  = "
                  + response.getCharacterEncoding());

      /**
       * Стандартное подключение ресурса сервлета.
       */
      RequestDispatcher disp =
                  request.getRequestDispatcher("/WEB-INF/request1b.jsp");

      if(request.getParameter("new") == null)
      {
            request.setAttribute("Title2",
                        "Мы рады вновь увидеть Вас!");

            request.setAttribute("Address2",
            "<a href=\"http://localhost:8080/proj14/Example14b?new=\">" +
            "Подключение к сервлету Example14b </a><hr>");

            request.setAttribute("isForm", false);
      }
      else
      {
            request.setAttribute("Title2",
                        "Введите Ваш запрос:");
            request.setAttribute("Address2", "");
            request.setAttribute("isForm", true);
      }

      disp.forward(request, response);

      System.out.println("request.getCharacterEncoding()   = "
                  + request.getCharacterEncoding());
      System.out.println("response.getCharacterEncoding()  = "
                  + response.getCharacterEncoding());
}
```

Особенность реализации метода ***doGet(...)*** - реакция на наличие или отсутствие параметра запроса ***new***. Хотя это — не лучшее решение, поскольку в JSP-странице можно было бы вместо ссылки с параметром воспользоваться формой на метод ***doPost(...)***, присвоив кнопке ***submit*** нужное значение имени.

Листинг 3.10 — Исходный текст метода doPost(...) сервлета Example14b

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 *          HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
{
      /**
       * Сохраненные значения параметров запросов
       * и их флаги.
       */
      String key, text, test;
      boolean fkey  = true;
      boolean ftext = true;
      boolean ftest = true;

      /**
       * Явная установка кодировок объектов запроса и ответа.
       * Стандартная установка контекста ответа.
       */
      System.out.println("\nМетод doPost() - принял запрос...");
      request.setCharacterEncoding("UTF-8");

      /**
       * Чтение и анализ параметров.
       */
      key  = request.getParameter("key");
      System.out.println("key  = " + key);
      fkey = testKey(key);

      text = request.getParameter("text");
      System.out.println("text = " + text);

      test = request.getParameter("test");
      System.out.println("test = " + test);
      if(!test.equals("Русский текст"))
            ftest = false;

      /**
       * Подключение ресурса сервлета по выбору.
       */
      RequestDispatcher disp;
      if(fkey & ftext & ftest)
      {
            disp =
                  request.getRequestDispatcher("/WEB-INF/request1b.jsp");
      request.setAttribute("Title2",
                  "Здесь следует использовать другую JSP-страницу!");
      request.setAttribute("Address2", "");

      request.setAttribute("isForm", true);
      }
      else
      {
            disp =
                  request.getRequestDispatcher("/WEB-INF/request1b.jsp");
            request.setAttribute("Title2",
                        "Спасибо, что воспользовались нашим приложением!");
            request.setAttribute("Address2",
```

```
                    "<a href=\"http://localhost:8080/proj14/Example14b\">" +
                    "Подключение к сервлету Example14b </a><hr>");

            request.setAttribute("isForm", false);
        }
        disp.forward(request, response);
    }
```

Представленный метод ***doPost(...)*** явно реализован не до конца, поскольку на него ложится основная нагрузка, связанная с анализом переданных параметров и формирование данных для соответствующей JSP-странице. Тем не менее, приведённые листинги наглядно демонстрируют технологию использования параметров и атрибутов объекта ***request***.

Учебное задание

На основе исходного текста сервлета ***Example14b.java***, его JSP-страницы и листингов 3.8-3.9 реализовать новый сервлет ***Example14c.java*** и соответствующие JSP-страницы, максимально упрощающие логическую структуру приведённого выше примера.

3.3 Практическое задание №9 — лабораторная №9. Шаблон проектирования MVC

Описание технологии шаблона MVC кратко изложено в учебнике [1, пункт 4.3.5, стр. 189-196]. Этот материал учебника [1] составляет основу лабораторной работы №9: «Шаблон проектирования MVC».

Учебный материал девятого (заключительного) практического занятия подводит итог изучения web-технологий распределенных систем. Он касается общих вопросов проектирования приложений с использованием технологии сервлетов и JSP-страниц, которые предоставляются инструментами сервера Apache Tomcat. В этом плане, структура занятия разделена на две части:

- проектирование элементов шаблона MVC;
- реализация проекта ***webpad*** в объёме функциональных возможностей класса ***NotePad***, описанного в учебнике [1].

3.3.1 Проектирование элементов шаблона MVC

Формально, шаблон (модель) MVC, как описано в [1], предлагает разделить приложение на три части:

- **Model** — некое хранилище (поставщик) данных в рамках всего проекта.
- **View** — набор HTML- и JSP-страниц, подготавливающий для сервлета ответ клиенту (браузеру);
- **Control** - сервлет, организующий доступ к приложению модели, принимающий от приложения данные, передающий компоненте ***View*** информацию на подготовку HTML-страниц и возвращающий результат клиенту (браузеру).

В представленной интерпретации шаблон MVC допускает множество вариаций исполнения, что, как показано в материале предыдущего занятия, сильно влияет на сложность реализуемого проекта и, как следствие, на качество его исполнения. В такой ситуации возникает естественный вопрос о последовательности проектных действий, устраняющих указанные негативные последствия.

Чтобы не усложнять изложение учебного материала излишними абстракциями, направим свои рассуждения на реализацию проекта ***webpad***, который обеспечивает полный функционал работы с записями базы данных ***exampleDB*** поддерживаемый методами класса ***NotePad*** и подробно описанный в учебнике [1]. Но поскольку указанный функционал уже частично реализован в проекте ***proj14*** в виде сервлета ***Example14***, то воспользуемся и результатами этого приложения.

В качестве предварительной подготовки для проведения дальнейших рассуждений, откроем ***Dynamic Web Project*** проект с именем ***webpad***, а затем:

- создадим сервлет с именем ***WebPad*** и ***package rvs.servlets***;
- создадим Java класс с именем ***ModelPad*** и ***package rvs.models***;

Проектирование начнём с анализа модуля ***Control***, функции которого испол-

няет сервлет ***WebPad***.

Сервлет ***WebPad*** должен предоставлять клиенту следующие функции:

1. Предоставление клиенту ***формы приветствия*** с предложением клиенту начать работу с функциями приложения ***WebPad***.
2. Предоставление клиенту ***стартовой формы*** для ввода запроса к серверу и два элемента управления, предлагающих *отправить* запрос серверу или *завершить* работу с приложением.
3. Предоставление клиенту ***формы для исправления*** значений запроса к серверу и два элемента управления, предлагающих *отправить* запрос серверу или *завершить* работу с приложением.
4. Предоставление клиенту ***формы результата*** запроса к серверу вместе с формой нового запроса, а также два элемента управления, предлагающих *отправить* запрос серверу или *завершить* работу с приложением.
5. Предоставление клиенту ***формы прощания*** с ним одним элементом управления, предлагающим новое соединение с приложением.

Из перечисленных выше задач сервлета ***WebPad***, первая и пятая решаются тривиально — вызовом соответствующих HTML-страниц, например, ***index.html*** и ***stop.html***, представленных на листингах 3.11 и 3.12.

Листинг 3.11 — Исходный текст HTML-страницы index.html сервлета WebPad

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Практика №9</title>
</head>
<body>
      <h2 align="center">
      РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
      </h2>
      <h4 align="center">
      бакалавриат кафедры АСУ
      </h4>

            <b>Мы рады вновь увидеть Вас!</b>
      <hr>
      <form action="WebPad" method="post" accept-charset="UTF-8">
                  <input type="hidden" name="test" value="start">
                  <input type="submit" value="Выполнить подключение к серверу...">
      </form>
      <hr>
</body>
</html>
```

Обратите внимание, что:

- страница сразу обеспечивает обращение к методу ***doPost(...)*** нового сервлета ***WebPad***, поэтому для метода ***doGet(...)*** не нужно создавать отдельную страницу и проверять различные условия;
- форма содержит скрытое поле с именем ***test***, имеющее начальное значение

start; по этому значению, метод ***doPost(...)*** определяет, что запрос делается первый раз, поэтому нет необходимости анализировать параметры запроса ***key*** и ***text***.

Листинг 3.12 — Исходный текст HTML-страницы stop.html сервлета WebPad

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Практика №9</title>
</head>
<body>
	<h2 align="center">
	РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
	</h2>
	<h4 align="center">
	бакалавриат кафедры АСУ
	</h4>

		<b>Спасибо, что воспользовались нашим приложением!</b>
	<hr>
	<form action="WebPad" method="post" accept-charset="UTF-8">
			<input type="hidden" name="test" value="start">
			<input type="submit" value="Выполнить подключение к серверу...">
	</form>
	<hr>
</body>
</html>
```

Эта страница отличается от предыдущей только выводимым сообщением, но оптимизировать здесь не имеет смысла, потому что тексты файлов — небольшие, а также имеется возможность дальнейших дизайнерских вариаций.

Теперь, если рассмотрим вторую функцию сервлета ***WebPad***, то заметим, что она тоже может быть представлена статической HTML-страницей, поскольку она содержит только форму запроса к базе данных и не предполагает анализ каких-либо параметров. Ее содержание представим исходным текстом файла с именем ***start.html***, приведённым на листинге 3.13.

Листинг 3.13 — Исходный текст HTML-страницы start.html сервлета WebPad

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Практика №9</title>
</head>
<body>
	<h2 align="center">
	РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
	</h2>
	<h4 align="center">
	бакалавриат кафедры АСУ
	</h4>

		<b>Вы подключились к приложению WebPad</b>
```

```
        <hr>
              Ввведите запрос к базе данных:
        <hr>
        <form action="WebPad" method="post" accept-charset="UTF-8">
                Введи ключ :
                    <input type="text" size="10" name="key"><br>

                Введи текст: <br>
                    <textarea rows="3" cols="40" name="text"></textarea><br>

                <input type="hidden" name="test" value="normal">
                <input type="submit" value="Выполнить запрос >">
        </form>
        <form action="WebPad" method="post" accept-charset="UTF-8">
                <input type="hidden" name="test" value="stop">
                <input type="submit" value="Завершить работу с приложением...">
        </form>
        <hr>
</body>
</html>
```

Обратите внимание, что форма этой страницы, отправляющая поля параметров ***key*** и ***text***, содержит скрытый параметр ***test*** со значением ***normal***. Это указывает методу ***doPost(...)***, что необходимо обработать вариант контроля входных параметров и сделать нужный запрос к классу ***Model***.

Таким образом, уже на предварительном этапе проектирования мы имеем реализацию трёх функций сервлета ***WebPad***, которые реализуются простым вызовом статических HTML-страниц, достаточных для полной реализации метода ***doGet(...)*** и частичной реализации метода ***doPost(...)***. При этом, различные варианты использования HTML-страниц точно определяются скрытым параметром с именем ***test***, который может принимать одно из значений: ***start***, ***stop*** или ***normal***.

Теперь обратим внимание на то, что, в процессе обработки запроса, сервлету ***WebPad*** потребуется форма (см. функция №3), которая будет выводить сообщение об обнаруженных ошибках и будет давать возможность исправить параметры запроса ***key*** и ***text***. Такая форма очень похожа на исходный текст HTML-страницы ***start.html***, но должна быть реализована в виде JSP-страницы, поскольку требует подстановки на стороне сервера значений уже введённых ранее параметров. Текст такой формы представлен в виде файла ***error.jsp*** на листинге 3.14.

Листинг 3.14 — Исходный текст JSP-страницы error.jsp сервлета WebPad

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Практика №9</title>
</head>
<body>
      <%-- Устанавливаем кодировку объекта response
             и типа контента --%>
      <%
            response.setCharacterEncoding("UTF-8");
```

```
            response.setContentType("text/html");
        %>
        <h2 align="center">
        РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
        </h2>
        <h4 align="center">
        бакалавриат кафедры АСУ
        </h4>

              <b style="color: red">Ошибка!</b>
        <hr>
              <%=request.getAttribute("msg") %>
        <hr>
        <form action="WebPad" method="post" accept-charset="UTF-8">
                Введи ключ :
                    <input type="text" size="10" name="key"
                          value=<%=request.getAttribute("key") %>><br>

                Введи текст: <br>
                    <textarea rows="3" cols="40"
                    name="text"><%=request.getAttribute("text")%></textarea><br>

                <input type="hidden" name="test" value="normal">
                <input type="submit" value="Выполнить запрос >">
        </form>
        <form action="WebPad" method="post" accept-charset="UTF-8">
                <input type="hidden" name="test" value="stop">
                <input type="submit" value="Завершить работу с приложением...">
        </form>
        <hr>
</body>
</html>
```

Обратите внимание, что, при использовании JSP-страницы ***error.jsp***, должны быть установлены значения атрибутов:

- **msg** — текст сообщения об ошибке;
- **key** — отображаемое в форме поле ***key***;
- **text** — отображаемое в форме поле ***text***.

На основе проведённых рассуждений и представленных на листингах 3.11-3.14 можно выполнить предварительную реализацию сервлета ***WebPad***, которая представлена на листинге 3.15.

Листинг 3.15 — Исходный текст предварительной реализации сервлета WebPad

```
package rvs.servlets;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import rvs.models.ModelPad;
```

```java
/**
 * Servlet implementation class WebPad
 */
@WebServlet("/WebPad")
public class WebPad extends HttpServlet
{
      private static final long serialVersionUID = 1L;

      /**
       * Объект доступа к классу ModelPad
       */
      private ModelPad model = null;


    /**
     * @see HttpServlet#HttpServlet()
     * Конструктор, обеспечивающий доступ
     * к объекту ModelPad.
     */
    public WebPad() {
        super();
        model = new ModelPad();
    }

      /**
       * Стандартные методы doGet(...) и doPost(...)
       *
       * @see HttpServlet#doGet(HttpServletRequest request,
       *          HttpServletResponse response)
       */
      protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                              throws ServletException, IOException
      {
            /**
             * Стандартная установка контекста ответа.
             */
            response.setCharacterEncoding("UTF-8");

            /**
             * Стандартное подключение ресурса сервлета.
             *
             * Функция №1 сервлета WebPad.
             */
            RequestDispatcher disp =
                        request.getRequestDispatcher("/index.html");

            disp.forward(request, response);

            System.out.println("Выполнена функция №1");
      }

      /**
       * @see HttpServlet#doPost(HttpServletRequest request,
       *          HttpServletResponse response)
       */
      protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
                              throws ServletException, IOException
      {
            /**
             * Стандартная установка контекста запроса.
```

```
 */
request.setCharacterEncoding("UTF-8");

RequestDispatcher disp;
String test =
            request.getParameter("test");
/**
 * Последовательность вариантов управления,
 * в зависимости от значения параметра test:
 */
if(test.equals("start"))      // Функция №2
{
      disp =
                  request.getRequestDispatcher("/WEB-INF/start.html");

      disp.forward(request, response);

      System.out.println("Выполнена функция №2 - start");

      return;
}

if(test.equals("stop"))       // Функция №5
{
      disp =
                  request.getRequestDispatcher("/WEB-INF/stop.html");

      disp.forward(request, response);

      System.out.println("Выполнена функция №5 - stop");

      return;
}

/**
 * Главная обработка запроса:
 *
 * Здесь предполагается будущая реализация
 * запросов к базе данных.
 */
if(test.equals("normal"))           // Функция №3-№5
{
      /**
       * Реализация функций №3-№5
       */

      System.out.println("Выполнена функция №4 - normal");
}

/**
 * Случай неизвестной ошибки.
 * Функция №3.
 */
disp =
            request.getRequestDispatcher("/WEB-INF/error.jsp");

request.setAttribute("msg", "Неизвестная ошибка: "
            + "Пожалуйста, введите правильный запрос...");
request.setAttribute("key",  "");
request.setAttribute("text", "");

disp.forward(request, response);
```

```
            System.out.println("Выполнена функция №3 - error");
      }
}
```

Обратите внимание, что представленный предварительный вариант сервлета ***WebPad*** позволяет провести тестирование и отладку проекта ***webpad***, что и необходимо сделать, а затем — перейти к анализу третьего компонента шаблона MVC — класса ***ModelPad***.

Объекты и методы созданного, но ещё не использованного класса ***ModelPad***, должны обеспечить проект ***WebPad*** необходимым функционалом для доступа к базе данных ***exampleDB***. Этот функционал полностью реализован в классе ***NotePad***, который хранится в архивной библиотеке ***$HOME/lib/notepad.jar***. Частично необходимый функционал работы с указанной базой данных уже создан и был использован в сервлете ***Example14*** проекта ***proj14*** (см. [1, листинг 4.8, стр. 192-193]).

Исходя из сказанного выше, в исходный текст класса ***WebPad*** без изменений можно перенести:

- статические объекты: ***notepad***, ***isOpen*** и ***isError***;
- статические методы: ***dbOpen()***, ***dbClose()*** и ***dbList()***.

Дополнительно, в него следует перенести исходный текст метода ***testKey(...)***, представленный на листинге 3.4 данного руководства.

Таким образом, предварительный состав исходного текста класса ***ModelPad*** может быть представлен листингом 3.16.

Листинг 3.16 — Исходный текст предварительной реализации класса ModelPad

```
package rvs.models;

import asu.rvs.NotePad;

public class ModelPad {
      /**
       * Объекты доступа к классу Notepad
       */
      private static NotePad notepad = null;
      private static boolean isOpen  = false;
      private static boolean isError = false;

      /**
       * Открытие объекта класса NotePad.
       * @return
       */
      protected static boolean dbOpen()
      {
            if(isError)
                  return false;
            if(!isOpen)
                  notepad = new NotePad();
            if(notepad == null)
            {
                  isError = true;
```

```
                System.out.println("Не могу открыть NotePad");
                return false;
        }else
        {
                isOpen = notepad.isConnect();
                if(isOpen)
                        System.out.println("NotePad - открыта!");
                else
                        System.out.println("NotePad - не открыта!");

                return isOpen;
        }
}

/**
 * Закрытие объекта класса NotePad.
 */
protected static void dbClose()
{
        if(isError || !isOpen)
                return;
        notepad.setClose();
        isOpen  = false;
        notepad = null;
}

/**
 * Получение списка записей объекта класса NotePad.
 * @return String[]
 */
public String[] dbList()
{
        if(!dbOpen())
                return null;

        Object[] obj = notepad.getList();

        if(obj == null)
        {
                System.out.println("getList() вернул null");
                return null;
        }
        int ns =
                        obj.length;
        System.out.println("Получено " + ns + " строк(и)");
        String[] ss =
                        new String[ns];

        for(int i=0; i < ns; i++)
                ss[i] = obj[i].toString();

        return ss;
}

/**
 * Метод анализа параметра key.
 */
public boolean testKey(String key)
{
        if(key.length() == 0)
                return false;
        try
```

```
            {
                  if(Integer.parseInt(key) > 0)
                  {
                        System.out.println("key  - правильное число");
                        return true;
                  }
                  return false;
            }
            catch(NumberFormatException e) {
                  return false;
            }
      }
}
```

Таким образом, в данном пункте практического занятия проведено проектирование составных частей шаблона MVC, а также реализована часть проекта ***webpad***, касающаяся простейших форм и уже имеющегося функционала. Соответственно, как итог последующей разработки, необходимо реализовать:

1. Функции класса ***ModelPad***, отвечающие за добавление записей в базу данных ***exampleDB*** (таблица ***notepad***), а также удаление записей из неё по заданному ключу ***key***.
2. JSP-страницу, обеспечивающую отображение результатов запросов к базе данных.
3. Часть метода ***doPost(...)***, отвечающего за контроль параметров запроса к сервлету ***WebPad***, формирующего запрос к базе данных и передающего ответ клиенту приложения.

Перечисленные задачи на реализацию проекта решаются в следующем пункте данного методического пособия.

3.3.2 Реализация проекта webpad

В предыдущем пункте проведено предварительное проектирование приложения, управляемого сервлетом ***WebPad***, и намечены задачи по его полной реализации.

Сначала необходимо реализовать все методы класса ***ModelPad***, обеспечивающие добавление записей в базу данных и удаление записей из неё. Такие методы реализованы в классе NotePad и опубликованы в учебнике [1, пункт 2.2.2, листинг 3.1, стр. 108-109]:

- **int *setInsert*(int *key*, String *str*)** — добавляет сообщение ***str*** с уникальным ключом ***key*** в таблицу ***notepad***; в случае успешного выполнения запроса возвращает число добавленных строк в таблицу, иначе — значение -1;
- **int *setDelete*(int *key*)** — удаляет строки из таблицы ***notepad***, которые имеют значение ключа ***key***; в случае успешного выполнения запроса возвращает число удалённых из таблицы строк, иначе — значение -1.

Указанные методы класса ***NotePad*** позволяют нам легко добавить в класс ***ModelPad*** недостающие два метода (***dbInsert(...)*** и ***dbDelete(...)***) для работы с базой

данных. Их исходные тексты представлены на листинге 3.17.

Листинг 3.17 — Добавляемые методы dbInsert(...) и dbDelete(...) класса ModelPad

```
/**
 * Метод добавления записи в базу данных.
 */
public String dbInsert(int key, String text)
{
	if(!dbOpen())
		return "-2"; // База данных не открыта.

	return
			String.valueOf(notepad.setInsert(key, text));
}

/**
 * Метод удаления записи из базы данных.
 */
public String dbDelete(int key)
{
	if(!dbOpen())
		return "-2"; // База данных не открыта.

	return
			String.valueOf(notepad.setDelete(key));
}
```

После реализации исходного теста листинга 3.17, класс ***ModelPad*** будет предоставлять четыре необходимых метода для работы с базой данных:

- ***testKey(...)*** - проверяющий правильность значения ключа ***key***;
- ***dbList()*** - получающий список строк, соответствующий содержимому базы данных;
- ***dbInsert()*** - добавляющий запись по значению ключа ***key***;
- ***dbDelete(...)*** - удаляющий запись по значению ключа ***key***.

Теперь следует реализовать последнюю из JSP-страниц, которая выводит результаты нормального обращения к базе данных.

Общая структура такой страницы должна выводить всю информацию о записях таблицы ***notepad***, поэтому она состоит из следующих последовательно расположенных частей:

- титульная часть;
- сообщение о выполненной операции;
- список содержимого таблицы ***notepad***;
- количество строк таблицы ***notepad***;
- форма для нового запроса к базе данных;
- форма с кнопкой завершения работы с приложением.

Исходный текст такой страницы с именем ***normal.jsp*** приведён на листинге 3.18.

Листинг 3.18 — Исходный текст JSP-страницы normal.jsp сервлета WebPad

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Практика №9</title>
</head>
<body>
      <%-- Устанавливаем кодировку объекта response
             и типа контента --%>
      <%
            response.setCharacterEncoding("UTF-8");
            response.setContentType("text/html");
      %>
      <h2 align="center">
      РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
      </h2>
      <h4 align="center">
      бакалавриат кафедры АСУ
      </h4>

            <b style="color: red"><%=request.getAttribute("msg") %>
            </b>
      <hr>
            Содержимое таблицы notepad:
      <hr>

      <%-- Объявление глобальной переменной --%>
      <%! private int ns = 0;%>
      <ul>
      <%-- Скриплет --%>
      <%
            String[] ss =
                  (String[])request.getAttribute("list");
            if(ss == null)
                  out.println("<li>Нет данных, возможно - ошибка БД!</li>");
            else
            {
                  ns = ss.length;
                  for(int i=0; i<ns; i++)
                        out.println("<li>" + ss[i] + "</li>");
            }
      %>
      </ul><hr>
      Получено <%=ns %> строк(и)<br>
      <hr>
            Ввведите запрос к базе данных:
      <hr>
      <form action="WebPad" method="post" accept-charset="UTF-8">
            Введи ключ :
                <input type="text" size="10" name="key"><br>

            Введи текст: <br>
                <textarea rows="3" cols="40" name="text"></textarea><br>

            <input type="hidden" name="test" value="normal">
            <input type="submit" value="Выполнить запрос >">
      </form>
      <form action="WebPad" method="post" accept-charset="UTF-8">
```

```
            <input type="hidden" name="test" value="stop">
            <input type="submit" value="Завершить работу с приложением...">
      </form>
      <hr>
</body>
</html>
```

Последним этапом разработки приложения ***webpad*** является завершение его функциональной части метода ***doPost(...)*** сервлета ***WebPad***, которая выделена на листинге 3.15 участком:

```
/**
 * Главная обработка запроса:
 *
 * Здесь предполагается будущая реализация
 * запросов к базе данных.
 */
if(test.equals("normal"))           // Функция №3-№5
{
      /**
       * Реализация функций №3-№5
       */

      System.out.println("Выполнена функция №4 - normal");
}
```

Его полная реализация представлена на листинге 3.19.

Листинг 3.19 — Завершающая часть метода doPost(...) класса WebPad

```
/**
 * Главная обработка запроса:
 *
 * Здесь предполагается будущая реализация
 * запросов к базе данных.
 */
if(test.equals("normal"))     // Функция №3-№5
{
      /**
       * Чтение параметров запроса key и text.
       */
      String key  =
                  (String)request.getParameter("key");
      String text =
                  (String)request.getParameter("text");

      System.out.println("key =" + key);
      System.out.println("text=" + text);

      String[] list; // Содержимое таблицы notepad.
      String   ret;  // Результат ответа операций удаления и вставки.
      /**
       * Установка атрибутов "key" и "text".
       */
      request.setAttribute("key",  key.trim());
      request.setAttribute("text", text.trim());

      /**
       * Реализация функций №3-№5
```

```
             */
            if(!model.testKey(key)) // Если ошибка значения key.
            {
                  disp =
                        request.getRequestDispatcher("/WEB-INF/error.jsp");

                  request.setAttribute("msg", "Неправильно задан ключ key: "
                        + "Пожалуйста, введите исправления в запрос...");

                  disp.forward(request, response);

                  System.out.println("Выполнена функция №3 - error");
                  return;
            }

            if(text.length() == 0)  // Запрос на удаление записи.
            {
                  ret = model.dbDelete(key);
                  request.setAttribute("msg",
                              "Удалено " + ret + " строка(и)");
            }
            else                          // Запрос на добаление записи.
            {
                  ret = model.dbInsert(key, text);
                  request.setAttribute("msg",
                              "Добавлено " + ret + " строка(и)");
            }
            if(ret.equals("-2"))
               request.setAttribute("msg", "База данных - не открыта");
            if(ret.equals("-1"))
               request.setAttribute("msg", "Нет соединения с базой данных");

            list = model.dbList();
            if(list == null)
            {
                  list    = new String[1];
                  list[0] = "Нет записей в таблице notepad";
            }
            request.setAttribute("list", list);

            disp =
                  request.getRequestDispatcher("/WEB-INF/normal.jsp");

            disp.forward(request, response);

            System.out.println("Выполнена функция №4 - normal");
            return;
      }
```

Учебное задание

На основе представленных выше примеров — завершить реализацию проекта ***webpad***.

Список использованных источников

1. Резник, В. Г. Распределенные вычислительные сети: Учебное пособие [Электронный ресурс] / В. Г. Резник. — Томск: ТУСУР, 2019. — 211 с. — Режим доступа: https://edu.tusur.ru/publications/9072
2. Ноутон П., Шилдт Г. JAVA 2. Наиболее полное руководство в подлиннике. СПб.: БХВ-Петербург, 2008. - 1072 с. - ISBN 978-5-94157-012-6
3. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux [Электронный ресурс]: Учебно-методическое пособие для студентов направления 09.03.01, Направление подготовки "Программное обеспечение средств вычислительной техники и автоматизированных систем" / В. Г. Резник - 2016. 33 с. — Режим доступа: https://edu.tusur.ru/publications/6238
4. Java DB Reference Manual (Документация по СУБД Derby) Дополнительная документация по дисциплине в файле: refderby.pdf